Rekayasa Perangkat Lunak

Migunani, S.Kom, M.Kom

YAYASAN PRIMA AGUS TEKNIK

Rekayasa Perangkat Lunak

Penulis :
Migunani, S.Kom., M.Kom

ISBN : 9 786235 734934

Editor :
Dr. Mars Caroline Wibowo. S.T., M.Mm.Tech

Penyunting :
Dr. Joseph Teguh Santoso, S.Kom., M.Kom.

Desain Sampul dan Tata Letak :
Irdha Yunianto, S.Ds., M.Kom

Penebit :
Yayasan Prima Agus Teknik Bekerja sama dengan
Universitas Sains & Teknologi Komputer (Universitas STEKOM)

Redaksi :
Jl. Majapahit no 605 Semarang
Telp. (024) 6723456
Fax. 024-6710144
Email : penerbit_ypat@stekom.ac.id

Distributor Tunggal :
Universitas STEKOM
Jl. Majapahit no 605 Semarang
Telp. (024) 6723456
Fax. 024-6710144
Email : info@stekom.ac.id

Hak cipta dilindungi undang-undang
Diarang memperbanyak karya tulis ini dalam bentuk dan dengan cara apapun tanpa ijin tertulis dari penerbit

KATA PENGANTAR

Puji syukur penulis panjatkan atas terselesaikannya buku yang berjudul ***"Rekayasa Perangkat Lunak"*** dengan baik. Penggunaan komputer komersial saat ini terhitung sekitar 60 tahun. Setelah melalui berbagai pengembangan, daya komputasi dan kecanggihannya meningkat pesat, sementara harganya turun drastis. Langkah cepat dalam teknologi komputasi tidak ada bandingannya dengan berbagai bidang usaha manusia lainnya. Dampak kemajuan perangkat lunak yang dibuat pada teknologi hardware sangatlah luar biasa. Semakin kuat spesifikasi sebuah komputer maka akan semakin program yang dapat dijalankannyapun semakin canggih.

Dalam Bab pertama, kita membahas beberapa isu dasar dalam rekayasa perangkat lunak. Dalam membangun dan memperbaiki pemrograman, seorang programmer biasanya mulai menulis program segera setelah ia membentuk pemahaman informal tentang persyaratan. Setelah penulisan program selesai, programmer harus turun untuk memperbaiki apa pun yang tidak memenuhi harapan pengguna. Biasanya, sejumlah besar perbaikan kode diperlukan bahkan untuk program mainan. Bab ini dilanjutkan dengan pembatasan perhatian kita pada beberapa yang penting dan umum digunakan. Dilanjutkan dengan bab 2 yang akan membahas beberapa konsep dasar yang terkait dengan model siklus hidup. Selanjutnya, kegiatan penting yang telah ditentukan untuk dilakukan dalam model air terjun klasik dan dilanjutkan dengan model spiral yang menggeneralisasi berbagai model siklus hidup. Dalam bab 3, kita menyelidiki mengapa manajemen proyek perangkat lunak jauh lebih kompleks daripada mengelola banyak jenis proyek lainnya. Dilanjutkan dengan penguraian tanggung jawab dan aktivitas utama manajer proyek perangkat lunak, bab ini akan ditutup dengan gambaran umum tentang aktivitas manajemen risiko dan konfigurasi.

Bab selanjutnya akan membahas tentang bagaimana cara mendokumentasikan model menggunakan bahasa pemodelan. Dalam dua bab ini akan dibahas berbagai proses desain yang dapat digunakan untuk menyempurnakan model analisis menjadi model desain secara iteratif. Dalam konteks konstruksi model, kita perlu memahami dengan cermat perbedaan antara bahasa pemodelan dan proses desain, karena kita akan sering menggunakan kedua istilah ini dalam diskusi kita. Bab selanjunya adalah membangun konsep pemodelan objek, kita juga akan mempelajari teknik analisis dan desain berorientasi objek (OOAD) yang menganjurkan pendekatan yang sangat berbeda dibandingkan dengan pendekatan desain berorientasi fungsi tradisional.

Bab selanjutnya akan membahas beberapa terminologi dan konsep umum yang terkait dengan pengembangan antarmuka pengguna, lalu mengklasifikasikan berbagai jenis antarmuka yang biasa digunakan. Disini juga diberikan beberapa panduan untuk merancang antarmuka yang baik, dan membahas beberapa alat untuk pengembangan antarmuka pengguna grafis (GUI). Lalu ditutup dengan metodologi pengembangan GUI. Bab 10 dan 11 akan membahas beberapa masalah penting yang terkait dengan kegiatan yang dilakukan dalam fase pengkodean. Selanjutnya, fokus pada berbagai jenis teknik pengujian program

untuk program prosedural dan berorientasi objek. Bab hingga akhir bab akan dipaparkan tentang definisikan ruang lingkup CASE dan memeriksa berbagai konsep yang terkait dengan CASE yang diikuti dengan berbagai fitur dari berbagai jenis tool CASE. Dan ditutup dengan informasi mengenai tren rekayasa perangkat lunak yang semakin berkembang yang jauh berbeda dengan awal mula keumunculan komputer. Akhir kata semoga buku ini berguna bagi para pembaca.

Semarang, Oktober 2022
Penulis

Migunani, S.Kom, M.Kom

DAFTAR ISI

Halaman Judul i
Kata Pengantar iii
Daftar Isi v
BAB 1 PENDAHULUAN 1
1.1 Evolusi – dari Bentuk Seni Menuju Disiplin Teknik 1
1.2 Proyek Pengembangan Perangkat Lunak 4
1.3 Gaya Eksplorasi Pengembangan Perangkat Lunak 7
1.4 Munculnya Teknik Perangkat Lunak 15
1.5 Perubahan Utama dalam Praktik Pengembangan Perangkat Lunak 22
1.6 Teknik Sistem Komputer 24
1.7 Ringkasan 25
1.8 Latihan 26
BAB 2 MODEL SIKLUS HIDUP PERANGKAT LUNAK 29
2.1 Beberapa Konsep Dasar 29
2.2 Model Air Terjun dan Perluasannya 34
2.3 *Rapid Application Development* (RAD) 53
2.4 Model Pengembangan Cepat 57
2.5 Model Spiral 63
2.6 Perbandingan Model Siklus Hidup yang Berbeda 66
2.7 Ringkasan 67
2.8 Latihan 68
BAB 3 MANAJEMEN PROYEK PERANGKAT LUNAK 78
3.1 Kompleksitas Manajemen Proyek Perangkat Lunak 78
3.2 Tanggung Jawab Manajer Proyek Perangkat Lunak 79
3.3 Perancangan Proyek 80
3.4 Metrik untuk Estimasi Ukuran Proyek 83
3.5 Teknik Estimasi Proyek 89
3.6 Teknik Estimasi Empiris 91
3.7 Como – Teknik Estimasi Heuristis 92
3.8 Cocomo 2 98
3.9 Ilmu Perangkat Lunak Halstead – Teknik Analitis 100
3.10 Estimasi Tingkat Staf 103
3.11 Penjadwalan 105
3.12 Organisasi dan Struktur Tim 113
3.13 Staf 119
3.14 Manajemen Risiko 120
3.15 Manajemen Konfigurasi Perangkat Lunak 123
3.16 Rencana Lainnya 127

3.17 Ringkasan 127
3.18 Latihan 128
BAB 4 ANALISIS KEBUTUHAN DAN SPESIFIKASI 135
4.1 Pengumpulan dan Analisis Persyaratan 136
4.2 Spesifikasi Kebutuhan Perangkat Lunak (SRS) 141
4.3 Spesifikasi Sistem Formal 160
4.4 Spesifikasi Axiomatis 164
4.5 Spesifikasi Aljabar 165
4.6 Spesifikasi yang dapat Dilakukan dan 4GL 169
4.7 Ringkasan 169
4.8 Latihan 169
BAB 5 DESAIN SOFTWARE 176
5.1 Tinjauan Proses Desain 176
5.2 Bagaimana Karakteristik Desain Perangkat Lunak yang Baik? 178
5.3 Kohesi dan Coupling 181
5.4 Penyusunan Modul Berlapis 185
5.5 Pendekatan untuk Desain Perangkat Lunak 187
5.6 Ringkasan 191
5.7 Latihan 192
BAB 6 DESAIN PERANGKAT LUNAK BERORIENTASI FUNGSI 195
6.1 Tinjauan Metodologi SA/SD 195
6.2 Analisis Terstruktur 196
6.3 Mengembangkan Model DFD dari Sistem 200
6.4 Desain Terstruktur 216
6.5 Desain yang Detail 221
6.6 Tinauan Desain 222
6.7 Ringkasan 222
6.8 Latihan 222
BAB 7 PEMODELAN OBJEK MENGGUNAKAN UML 242
7.1 Konsep Objek – Orientasi Dasar 243
7.2 *Unified Modelling Language* (UML) 259
7.3 Diagram UML 263
7.4 Gunakan Model Kasus 264
7.5 Gunakan Kemasan Kasus 273
7.6 Diagram Kelas 274
7.7 Diagram Interaksi 279
7.8 Diagram Aktivitas 281
7.9 Diagram Bagan Negara 282
7.10 Nota Bene 284
7.11 UML 2.0 285
7.12 RIngkasan 287
7.13 Latihan 287

BAB 8 PENGEMBANGAN PERANGKAT LUNAK BERORIENTASI OBYEK 295
8.1 Pola 296
8.2 Beberapa Pola Desain Umum 300
8.3 Ahli 300
8.4 Kreator 301
8.5 Pola Pemisahan Tampilan Model 302
8.6 Pola Pengambil 303
8.7 Pola *Model View Controller* (MVC) 304
8.8 Pola Publikasi – Berlangganan 305
8.9 Pola Perantara (atau Proksi) 306
8.10 Metodologi Analisis dan Desain Berorientasi Obyek (OOAD) 307
8.11 Aplikasi Analisis dan Proses Desain 317
8.12 Kriteria Kebaikan 323
8.13 Ringkasan 324
8.14 Latihan 324
BAB 9 DESAIN USER INTERFACE 328
9.1 Karakteristik User Interface yang Baik 328
9.2 Konsep Dasar 331
9.3 Jenis Antarmuka Pengguna 332
9.4 Dasar-dasar Pengembangan GUI Berbasis Komponen 336
9.5 Metodologi Desain Antarmuka Pengguna 342
9.6 Ringkasan 347
9.7 Latihan 347
BAB 10 CODING DAN TESTING F 351
10.1 Koding 351
10.2 Kode Review 354
10.3 Dokumentasi Perangkat Lunak 356
10.4 Pengujian 358
10.5 Pengujian Unit 365
10.6 Pengujian Kotak Hitam 366
10.7 Pengujian Kotak Putih 368
10.8 Debugging 377
10.9 Alat Analisis Program 378
10.10 Tes Integrasi 379
10.11 Menguji Program Berorientasi Objek 381
10.12 Pengujian Sistem 384
10.13 Beberapa Masalah Umum yang Terkait dengan Pengujian 387
10.14 Ringkasan 388
10.15 Latihan 388
BAB 11 REABILITAS SOFTWARE DAN MANAJEMEN KUALITAS 403
11.1 Reabilitas Software 403
11.2 Pengujian Statistik 408

11.3 Kualitas Perangkat Lunik 409
11.4 Sistem Manajemen Kualitas Perangkat Lunak 410
11.5 Model Kematian Kemampuan SEI 416
11.6 Beberapa Standar Kualitas Penting Lainnya 420
11.7 Enam Sigma 422
11.8 Ringkasan 423
11.9 Latihan 423
BAB 12 TEKNIK PERANGKAT LUNAK BERBANTUAN KOMPUTER 428
12.1 Kasus dan Ruang Lingkupnya 428
12.2 Lingkungan Kasus 428
12.3 Dukungan Kasus dalam Siklus Hidup Perangkat Lunak 430
12.4 Karakteristik Alat Kasus Lainnya 431
12.5 Menuju Alat Kasus Generasi Kedua 433
12.6 Arsitektur Lingkungan Kasus 433
12.7 Ringkasan 434
12.8 Latihan 434
BAB 13 PEMELIHARAAN PERANGKAT LUNAK 435
13.1 Karakteristik Pemeliharaan Perangkat Lunak 435
13.2 Teknik Pembalik Perangkat Lunak 437
13.3 Model Proses Pemeliharaan Perangkat Lunak 438
13.4 Estimasi Biaya Perawatan 440
13.5 Ringkasan 441
13.6 Latihan 441
BAB 14 PENGGUNAAN PERANGKAT LUNAK 443
14.1 Apa yang Dapat Digunakan Kembali? 443
14.2 Mengapa Hampir Tidak Ada Reuse Sejauh Ini? 444
14.3 Masalah Dasar dalam Setiap Program Penggunaan Kembali 444
14.4 Pendekatan Penggunaan Kembali 445
14.5 Penggunaan Kembali di Tingkat Organisasi 448
14.6 Ringkasan 450
14.7 Latihan 451
BAB 15 TREN YANG MUNCUL 453
15.1 Perangkat Lunak Client – Server 454
15.2 Arsitektur Client – Server 456
15.3 CORBA 458
15.4 ORB 458
15.5 COM/DCOM 461
15.6 *Service Oriented Architecture* (SOA) 461
15.7 *Software As A Service* (SAAS) 463
15.8 Ringkasan 464
15.9 Latihan 464
Daftar Pustaka 466

BAB 1
PENDAHULUAN

Rekayasa Perangkat Lunak

Definisi populer dari rekayasa perangkat lunak adalah: “Sebuah kumpulan sistematis dari praktik dan teknik pengembangan program yang baik”. Teknik pengembangan program yang baik dihasilkan dari inovasi penelitian dan berbagai pelajaran yang dipetik oleh programmer melalui pengalaman pemrograman selama bertahun-tahun. Definisi alternatif dari rekayasa perangkat lunak adalah: "Sebuah pendekatan rekayasa untuk mengembangkan perangkat lunak". Berdasarkan kedua sudut pandang tersebut, kita dapat mendefinisikan rekayasa perangkat lunak sebagai berikut:

Rekayasa perangkat lunak membahas teknik sistematis dan pengembangan perangkat lunak yang hemat biaya. Teknik-teknik ini membantu mengembangkan perangkat lunak menggunakan pendekatan rekayasa.

Apakah rekayasa perangkat lunak merupakan ilmu atau seni?

Beberapa orang berpendapat bahwa menulis program yang berkualitas adalah seni. Dalam konteks ini, mari kita periksa apakah rekayasa perangkat lunak benar-benar suatu bentuk seni atau mirip dengan disiplin ilmu teknik lainnya. Ada beberapa masalah mendasar yang membedakan disiplin ilmu teknik seperti rekayasa perangkat lunak dan teknik sipil, diantaranya adalah:

- Sama seperti disiplin ilmu teknik lainnya, rekayasa perangkat lunak menggunakan banyak pengetahuan yang diperoleh dari pengalaman sejumlah besar praktisi. Pengalaman masa lalu ini diatur secara sistematis dan sedapat mungkin dasar teoretis untuk pengamatan empiris telah disediakan. Semua solusi ilmiah dibangun melalui penerapan prinsip-prinsip yang dapat dibuktikan secara ketat sehingga tidak akan ada pembenaran teoretis yang masuk akal, pengalaman masa lalu telah diadopsi sebagai aturan praktis.
- Sama juga seperti semua disiplin ilmu teknik, saat memecahkan masalah dalam rekayasa perangkat lunak beberapa tujuan yang saling bertentangan seringkali ditemui. Dalam situasi seperti itu, beberapa solusi alternatif diusulkan. Sebuah solusi yang tepat dipilih dari solusi kandidat berdasarkan berbagai trade-off yang perlu dilakukan karena masalah biaya, perawatan, dan kegunaan, sehingga ketika sampai pada solusi akhir, beberapa iterasi dapat dimungkinkan.
- Disiplin teknik seperti rekayasa perangkat lunak hanya menggunakan prinsip-prinsip yang dipahami dan terdokumentasi dengan baik. Seni, di sisi lain, sering didasarkan pada membuat penilaian subjektif berdasarkan atribut kualitatif dan menggunakan prinsip-prinsip yang kurang dipahami.

Dari penjelasan di atas, kita dapat dengan mudah menyimpulkan bahwa rekayasa perangkat lunak dalam banyak hal mirip dengan disiplin ilmu teknik lainnya seperti teknik sipil atau teknik elektronik.

1.1 EVOLUSI—DARI BENTUK SENI MENUJU DISIPLIN TEKNIK

Di bagian ini, kita akan meninjau bagaimana disiplin rekayasa perangkat lunak berkembang selama bertahun-tahun.

Evolusi Seni menjadi Disiplin Teknik

Prinsip-prinsip rekayasa perangkat lunak telah berkembang selama enam puluh tahun terakhir dengan kontribusi dari banyak peneliti dan profesional perangkat lunak. Komputer muncul dari seni murni menjadi kerajinan, dan akhirnya menjadi disiplin teknik.

Programmer awal menggunakan gaya pemrograman *ad hoc*. Gaya pengembangan program ini sekarang disebut sebagai gaya *eksplorasi, build and fix,* dan *code and fix*. Dalam gaya membangun dan memperbaiki, sebuah program dengan cepat dikembangkan tanpa membuat spesifikasi, rencana, atau desain apa pun. Ketidaksempurnaan yang berbeda yang kemudian diperhatikan diperbaiki.

Gaya pemrograman eksplorasi adalah gaya informal dalam artian bahwa, tidak ada aturan atau rekomendasi yang harus dipatuhi oleh seorang programmer — setiap programmer dapat mengembangkan teknik pengembangan perangkat lunak mereka yang dipandu oleh intuisi, pengalaman, keinginan, dan fantasi mereka sendiri. Gaya eksplorasi datang secara alami untuk semua programmer pemula. Tipikal gaya eksplorasi ini biasanya menghasilkan kualitas yang buruk dan kode yang tidak dapat dipelihara, membuat pengembangan program menjadi sangat mahal dan memakan waktu.

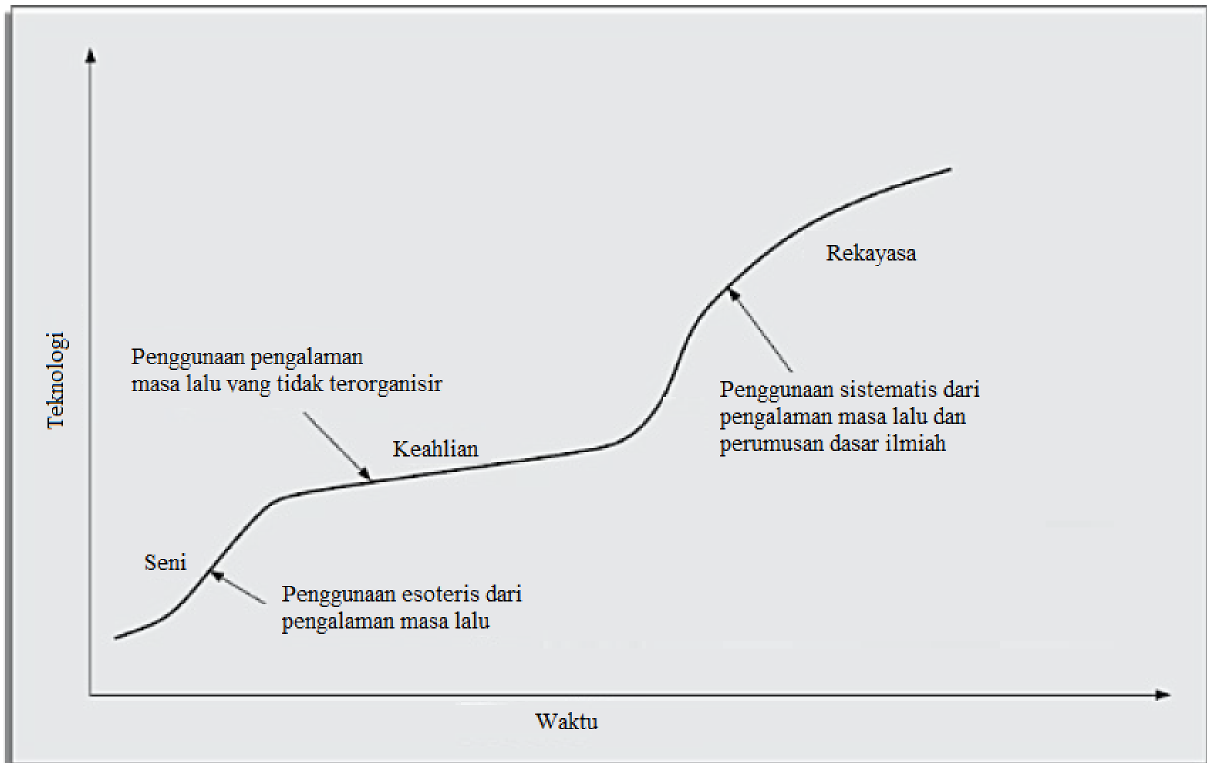
Gaya build and fix diadopsi secara luas oleh para programmer di tahun-tahun awal sejarah komputasi. Kita dapat menganggap gaya pengembangan program eksplorasi sebagai sebuah seni yang mana, gaya ini sebagian besar didapatkan melalui intuisi. Ada banyak cerita tentang programmer di masa lalu bak seniman mahir, mereka mampu menulis program yang baik menggunakan model dasar untuk membangun dan memperbaiki masalah dengan berbagai pengetahuan esoteris. Programmer yang bekerja di industri perangkat lunak modern jarang menggunakan pengetahuan esoterik dan mengembangkan perangkat lunak dengan menerapkan beberapa prinsip yang dipahami dengan baik.

Pola Evolusi untuk Disiplin Teknik

Jika kita menganalisis evolusi gaya pengembangan perangkat lunak selama enam puluh tahun terakhir, kita dapat melihat bahwa, evolusi dari bentuk seni esoteris ke bentuk kerajinan, dan kemudian perlahan-lahan muncul sebagai disiplin teknik. Faktanya, pola evolusi ini tidak jauh berbeda dari yang terlihat dalam disiplin ilmu teknik lainnya. Terlepas dari apakah itu pembuatan besi, pembuatan kertas, pengembangan perangkat lunak, atau konstruksi bangunan; evolusi teknologi telah mengikuti pola yang sangat mirip. Pola perkembangan teknologi ini secara skematis ditunjukkan pada Gambar 1.1, bahwa setiap teknologi pada tahun-tahun awal dimulai sebagai suatu bentuk seni. Seiring dengan berjalannya waktu berubah menjadi kerajinan dan akhirnya muncul sebagai disiplin teknik.

Mari kita ilustrasikan fakta ini menggunakan sebuah contoh. Pertimbangkan evolusi teknologi pembuatan besi. Pada zaman kuno, hanya sedikit orang yang tahu cara membuat besi karena mereka merahasiakan cara membuatnya. Pengetahuan esoteris ini diturunkan dari generasi ke generasi sebagai rahasia keluarga. Dengan berjalannya waktu, perlahan teknologi beralih dari seni ke bentuk kerajinan di mana para pedagang berbagi pengetahuan dengan murid-murid mereka dan kumpulan pengetahuan tersebut terus berkembang, setelah sekian lama teknologi pembuatan baja modern muncul melalui organisasi sistematis, dokumentasi pengetahuan, dan penggabungan dasar ilmiah.

Kisah evolusi disiplin rekayasa perangkat lunak tidak jauh berbeda dengan evolusi teknologi pembuatan besi. Pada awal munculnya pemrograman, terdapat daua jenis programmer, yaitu programmer yang baik dan programmer yang buruk. Programmer yang baik mengetahui prinsip (atau trik) tertentu yang membantu mereka menulis program yang baik, yang jarang mereka bagikan dengan programmer yang buruk. Penulisan program di tahun-tahun berikutnya mirip dengan kerajinan. Selama beberapa tahun berikutnya, semua prinsip (atau trik) diorganisasikan ke dalam kumpulan pengetahuan yang membentuk disiplin rekayasa perangkat lunak.

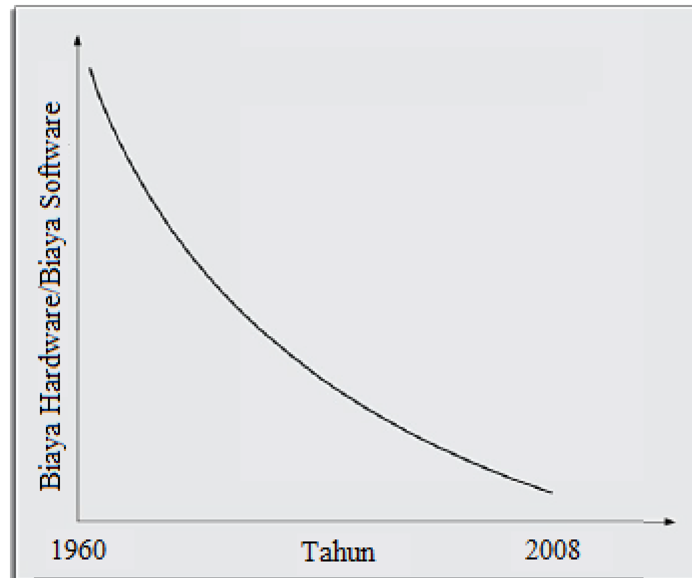


Gambar 1.1 Evolusi teknologi dengan waktu.

Prinsip-prinsip rekayasa perangkat lunak sekarang banyak digunakan di industri, dan prinsip-prinsip baru masih terus muncul dengan kecepatan yang sangat cepat—membuat disiplin ini sangat dinamis. Terlepas dari penerimaannya yang luas, para kritikus menunjukkan bahwa banyak metodologi dan pedoman yang diberikan oleh disiplin rekayasa perangkat lunak tidak memiliki dasar ilmiah, subjektif, dan tidak memadai. Namun, fakta bahwa mengadopsi teknik rekayasa perangkat lunak memfasilitasi pengembangan perangkat lunak berkualitas tinggi dengan cara yang hemat biaya dan tepat waktu tidak dapat disangkal. Praktik rekayasa perangkat lunak telah terbukti sangat diperlukan untuk pengembangan produk perangkat lunak besar—meskipun gaya eksplorasi sering kali berhasil digunakan untuk mengembangkan program kecil seperti yang ditulis oleh mahasiswa sebagai tugas kelas.

Solusi untuk Krisis Perangkat Lunak

Saat ini, rekayasa perangkat lunak tampaknya menjadi salah satu pilihan yang tersedia untuk mengatasi krisis perangkat lunak saat ini. Tapi, apa sebenarnya krisis perangkat lunak saat ini? Apa saja gejala yang muncul, penyebab, dan solusi yang mungkin bisa digunakan? Untuk memahami krisis perangkat lunak saat ini, pertimbangkan fakta berikut. Selama bertahun-tahun biaya yang dikeluarkan oleh organisasi di seluruh dunia untuk pembelian software jauh lebih mengkhawatirkan jika dibandingkan dengan biaya yang dikeluarkan untuk pembelian hardware (lihat Gambar 1.2). Seperti yang terlihat pada gambar, organisasi menghabiskan porsi anggaran yang semakin besar untuk perangkat lunak dibandingkan dengan perangkat keras. Di antara semua gejala krisis perangkat lunak saat ini, tren peningkatan biaya perangkat lunak mungkin yang paling menjengkelkan.



Gambar 1.2 Perubahan relatif dari biaya hardware dan software dari waktu ke waktu.

Tidak hanya produk perangkat lunak menjadi semakin lebih mahal daripada perangkat keras, tetapi mereka juga menghadirkan sejumlah masalah lain pada pelanggan—produk perangkat lunak sulit untuk diubah, di-debug, dan ditingkatkan; menggunakan sumber daya dengan tidak optimal; sering gagal memenuhi persyaratan pengguna; tidak dapat diandalkan; sering crash; dan sering lambat.

Saat ini, banyak organisasi menghabiskan biaya lebih banyak untuk software. Jika tren ini berlanjut, kita mungkin akan segera memiliki skenario yang agak lucu. Gejala krisis perangkat lunak tidak sulit untuk diamati. Beberapa faktor diantaranya ukuran masalah yang meningkat pesat, kurangnya pelatihan yang memadai dalam teknik rekayasa perangkat lunak, kekurangan keterampilan, dan peningkatan produktivitas yang rendah. Lalu, solusi yang memuaskan untuk krisis perangkat lunak saat ini datang dari penyebaran praktik rekayasa perangkat lunak di antara para developer, ditambah dengan kemajuan lebih lanjut pada disiplin rekayasa perangkat lunak itu sendiri. Dengan diskusi singkat tentang evolusi dan dampak disiplin rekayasa perangkat lunak ini, kita dapat memeriksa beberapa konsep dasar yang berkaitan dengan berbagai jenis proyek pengembangan perangkat lunak yang dilakukan oleh perusahaan perangkat lunak.

1.2 PROYEK PENGEMBANGAN PERANGKAT LUNAK

Sebelum membahas tentang berbagai jenis proyek pengembangan yang dilakukan oleh perusahaan pengembangan perangkat lunak, mari kita pahami terlebih dahulu perbedaan penting antara perangkat lunak profesional dengan perangkat lunak mainan seperti yang ditulis oleh seorang mahasiswa dalam tugas pemrograman.

Program versus Produk

Banyak perangkat lunak mainan sedang dikembangkan oleh individu seperti mahasiswa untuk tugas kelas mereka dan para pemliki hobi untuk penggunaan pribadi mereka. Produk seperti ini biasanya berukuran kecil dan mendukung fungsionalitas terbatas. Selanjutnya, pembuat program biasanya merupakan satu-satunya pemelihara kode dan pengguna perangkat lunak tersebut. Oleh karena itu, perangkat lunak mainan ini biasanya tidak memiliki antarmuka pengguna yang baik dan dokumentasi yang tepat. Selain itu, ini mungkin memiliki pemeliharaan, efisiensi, dan keandalan yang buruk. Karena perangkat lunak

mainan ini tidak memiliki dokumen pendukung seperti manual pengguna, manual perawatan, dokumen desain, dokumen uji, dll.

Sebaliknya, perangkat lunak profesional biasanya memiliki banyak pengguna dan, memiliki antarmuka pengguna yang baik, panduan pengguna yang tepat, dan dukungan dokumentasi yang baik. Karena, sebuah produk perangkat lunak memiliki banyak pengguna, maka perangkat tersebt dirancang secara sistematis, diimplementasikan dengan hati-hati, dan diuji secara menyeluruh. Selain itu, perangkat lunak yang ditulis secara profesional biasanya tidak hanya terdiri dari kode program tetapi juga semua dokumen terkait seperti dokumen spesifikasi persyaratan, dokumen desain, dokumen uji, manual pengguna, dll. Perbedaan lebih lanjut adalah perangkat lunak profesional memiliki ukuran yang besar dan kompleks untuk dikembangkan lebih lanjut oleh setiap individu. Biasanya dikembangkan oleh sekelompok developer yang bekerja dalam tim. oftware profesional dikembangkan oleh sekelompok developer software yang bekerja sama dalam sebuah tim, sehingga perlu untuk menggunakan beberapa metodologi pengembangan yang sistematis. Jika tidak, mereka akan merasa sangat sulit untuk berinteraksi dan memahami pekerjaan satu sama lain, dan menghasilkan satu set dokumen yang koheren.

Meskipun prinsip-prinsip rekayasa perangkat lunak terutama dimaksudkan untuk digunakan dalam pengembangan perangkat lunak profesional, banyak hasil rekayasa perangkat lunak dapat secara efektif digunakan untuk pengembangan program kecil juga. Namun, ketika mengembangkan program kecil untuk penggunaan pribadi, kepatuhan yang kaku terhadap prinsip-prinsip rekayasa perangkat lunak seringkali tidak bermanfaat. Ibarat seekor semut bisa dibunuh menggunakan pistol, tapi itu akan sangat tidak efisien dan juga tidak pantas. CAR Hoare [1994] mengamati bahwa menggunakan prinsip-prinsip rekayasa perangkat lunak untuk mengembangkan program mainan secara ketat sangat mirip dengan menerapkan prinsip-prinsip teknik sipil dan arsitektur untuk membangun istana pasir untuk dimainkan anak-anak.

Jenis Proyek Pengembangan Perangkat Lunak

Perusahaan pengembangan perangkat lunak biasanya disusun menjadi sejumlah besar tim yang dapat menangani berbagai jenis proyek pengembangan perangkat lunak. Proyek pengembangan perangkat lunak ini berkaitan dengan pengembangan produk perangkat lunak atau beberapa layanan perangkat lunak.

Produk perangkat lunak

Kita semua tahu berbagai perangkat lunak seperti Microsoft Windows dan Office suite, Oracle DBMS, perangkat lunak yang menyertai camcorder atau printer laser, dll. Perangkat lunak ini tersedia untuk dibeli dan digunakan oleh beragam pelanggan. Ini disebut produk perangkat lunak generik karena banyak pengguna pada dasarnya menggunakan perangkat lunak yang sama. Ini dapat dibeli langsung oleh pelanggan. Ketika perusahaan developer perangkat lunak ingin mengembangkan produk generik, pertama-tama mereka akan menentukan fitur atau fungsi yang akan berguna bagi sebagian besar pengguna. Berdasarkan ini, tim developer menyusun spesifikasi produk sendiri. Tentu saja, mendasarkan kebijaksanaan desainnya pada umpan balik yang dikumpulkan dari sejumlah besar pengguna. Biasanya, setiap produk perangkat lunak ditargetkan untuk beberapa segmen pasar (kumpulan pengguna). Banyak perusahaan merasa menguntungkan untuk mengembangkan lini produk yang menargetkan segmen pasar yang sedikit berbeda berdasarkan variasi perangkat lunak yang pada dasarnya sama. Misalnya, Microsoft menargetkan desktop dan laptop melalui sistem operasi Windows 8, sementara itu menargetkan handset seluler kelas atas melalui sistem operasi seluler Windows, dan menargetkan server melalui sistem operasi server Windows.

Layanan perangkat lunak

Layanan perangkat lunak biasanya melibatkan pengembangan perangkat lunak yang disesuaikan atau pengembangan beberapa bagian tertentu dari perangkat lunak dalam mode outsourcing. Perangkat lunak yang disesuaikan dikembangkan sesuai dengan spesifikasi yang dibuat oleh satu atau paling banyak beberapa pelanggan. Ini perlu dikembangkan dalam jangka waktu yang singkat (biasanya beberapa bulan), dan pada saat yang sama biaya pengembangan harus rendah. Biasanya, perusahaan berkembang mengembangkan perangkat lunak yang disesuaikan dengan menyesuaikan beberapa perangkat lunak yang ada. Misalnya, ketika sebuah institusi akademik ingin memiliki perangkat lunak yang akan mengotomatisasi kegiatan penting seperti pendaftaran siswa, penilaian, dan pengumpulan biaya; perusahaan biasanya akan mengembangkan perangkat lunak seperti itu sebagai produk yang disesuaikan. Ini berarti bahwa untuk mengembangkan perangkat lunak yang disesuaikan, perusahaan yang sedang berkembang biasanya akan menyesuaikan salah satu produk perangkat lunak yang ada yang mungkin telah dikembangkan di masa lalu untuk beberapa institusi akademik lainnya.

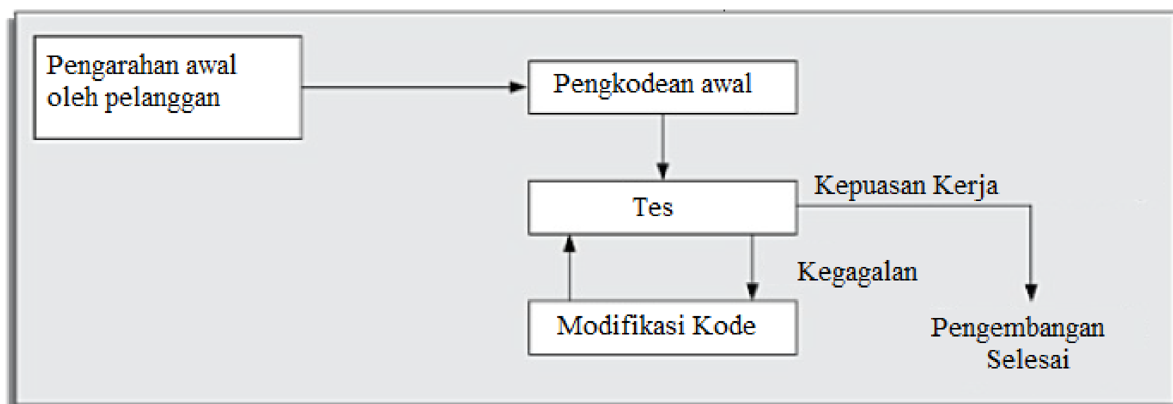
Dalam proyek pengembangan perangkat lunak yang disesuaikan, sebagian besar perangkat lunak digunakan kembali dari kode perangkat lunak terkait yang mungkin telah dikembangkan perusahaan. Biasanya, hanya sebagian kecil dari perangkat lunak yang khusus untuk beberapa klien yang dikembangkan. Misalnya, organisasi pengembangan perangkat lunak telah mengembangkan software otomatisasi akademik yang mengotomatiskan pendaftaran siswa, penilaian, pendirian, asrama, dan aspek lain dari lembaga akademik. Ketika lembaga pendidikan baru meminta untuk mengembangkan perangkat lunak untuk otomatisasi kegiatannya, sebagian besar perangkat lunak yang ada akan digunakan kembali. Namun, sebagian kecil dari kode yang ada dapat dimodifikasi untuk memperhitungkan variasi kecil dalam fitur yang diperlukan. Misalnya, perangkat lunak mungkin telah dikembangkan untuk lembaga akademis yang hanya menawarkan program residensial reguler, lembaga pendidikan yang sekarang meminta perangkat lunak untuk mengotomatisasi aktivitasnya juga menawarkan program pasca sarjana mode jarak jauh di mana pengajaran dan evaluasi sesi dilakukan oleh pusat-pusat lokal.

Jenis lain dari layanan perangkat lunak adalah perangkat lunak outsourcing. Terkadang, masuk akal secara komersial bagi perusahaan yang mengembangkan proyek besar untuk mengalihdayakan beberapa bagian dari pekerjaan pengembangannya ke perusahaan lain. Alasan di balik keputusan seperti itu mungkin banyak. Misalnya, sebuah perusahaan mungkin mempertimbangkan opsi outsourcing, jika merasa tidak memiliki keahlian yang cukup untuk mengembangkan beberapa bagian tertentu dari perangkat lunak; atau jika ditentukan bahwa beberapa bagian dapat dikembangkan dengan biaya yang efektif oleh perusahaan lain. Karena proyek outsourcing adalah bagian kecil dari beberapa proyek yang lebih besar, proyek outsourcing biasanya berukuran kecil dan perlu diselesaikan dalam beberapa bulan atau beberapa minggu.

Jenis proyek pengembangan yang dilakukan oleh suatu perusahaan akan berdampak pada profitabilitasnya. Misalnya, sebuah perusahaan yang telah mengembangkan produk perangkat lunak generik biasanya mendapatkan aliran pendapatan yang tidak terputus yang tersebar selama beberapa tahun. Namun, ini memerlukan investasi awal yang besar dalam mengembangkan perangkat lunak dan pengembalian atas investasi ini tunduk pada risiko penerimaan pelanggan. Di sisi lain, proyek yang dialihdayakan biasanya kurang berisiko, tetapi hanya menghasilkan satu kali pendapatan bagi perusahaan yang sedang berkembang.

1.3 GAYA EKSPLORASI PENGEMBANGAN PERANGKAT LUNAK

Kita telah mempelajari bahwa gaya pengembangan program eksplorasi mengacu pada gaya pengembangan informal di mana programmer menggunakan intuisinya sendiri untuk mengembangkan program, bukan memanfaatkan tubuh pengetahuan sistematis yang dikategorikan di bawah disiplin rekayasa perangkat lunak. Gaya pengembangan eksplorasi memberikan kebebasan penuh kepada programmer untuk memilih aktivitas yang digunakan untuk mengembangkan perangkat lunak. Meskipun gaya eksplorasi tidak memberlakukan aturan, pengembangan tipikal dimulai setelah pengarahan awal dari pelanggan. Berdasarkan pengarahan ini, para developer memulai koding untuk mengembangkan program kerja. Software diuji dan bug yang ditemukan diperbaiki. Siklus pengujian dan perbaikan bug ini berlanjut hingga software dapat bekerja dengan baik dan memuaskan bagi pelanggan. Skema urutan pekerjaan dalam gaya membangun dan memperbaiki ditunjukkan pada Gambar 1.3. Amati bahwa pengkodean dimulai setelah pengarahan pelanggan awal tentang apa yang diperlukan. Setelah pengembangan program selesai, siklus pengujian dan perbaikan berlanjut hingga program dapat diterima oleh pelanggan. Gaya pengembangan eksplorasi dapat berhasil bila digunakan untuk mengembangkan program yang sangat kecil, dan bukan untuk perangkat lunak profesional.



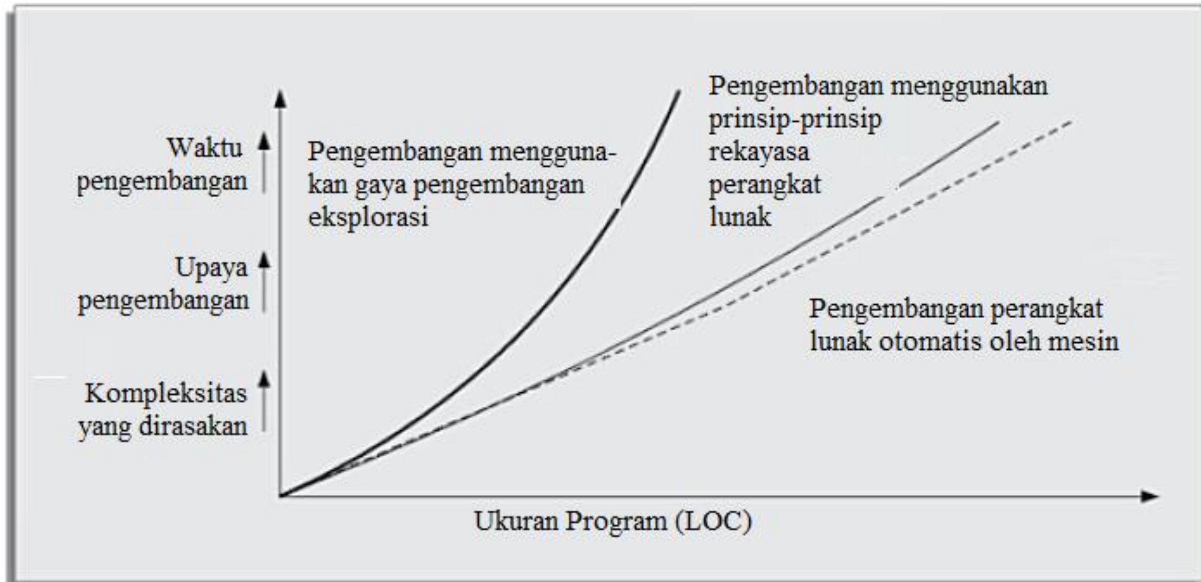
Gambar 1.3 Pengembangan program eksplorasi

Apa yang salah dengan gaya eksplorasi pengembangan perangkat lunak?

Meskipun gaya pengembangan perangkat lunak eksplorasi secara intuitif jelas, tidak ada tim perangkat lunak yang dapat tetap kompetitif jika menggunakan gaya pengembangan perangkat lunak ini. Mari kita selidiki alasan di balik ini. Dalam skenario pengembangan eksplorasi, upaya dan waktu yang dibutuhkan untuk mengembangkan perangkat lunak profesional meningkat seiring dengan peningkatan ukuran program. Mari kita pertimbangkan terlebih dahulu bahwa gaya eksplorasi digunakan untuk mengembangkan perangkat lunak profesional. Peningkatan upaya pengembangan dan waktu dengan ukuran masalah telah ditunjukkan pada Gambar 1.4. Amati plot garis tebal yang mewakili kasus di mana gaya eksplorasi digunakan untuk mengembangkan program. Dapat dilihat bahwa dengan bertambahnya ukuran program, upaya dan waktu yang dibutuhkan akan semakin meningkat, hampir secara eksponensial. Untuk ukuran masalah yang besar, akan memakan waktu terlalu lama dan biaya yang terlalu banyak agar bisa menjadi bermakna secara praktis dalam mengembangkan program menggunakan gaya pengembangan eksplorasi.

Pendekatan pengembangan eksploratif dikatakan rusak setelah ukuran program yang akan dikembangkan meningkat melebihi nilai tertentu. Misalnya, dengan menggunakan gaya eksplorasi, kita dapat dengan mudah memecahkan masalah yang memerlukan penulisan hanya 1000 atau 2000 baris *source code* (kode sumber). Namun, jika diminta untuk

memecahkan masalah yang memerlukan penulisan satu juta baris kode sumber, kita mungkin tidak dapat menyelesaikanya menggunakan gaya eksplorasi; terlepas dari jumlah waktu atau usaha yang mungkin kita investasikan untuk menyelesaikannya. Sekarang amati plot garis tipis padat pada Gambar 1.4 yang mewakili kasus ketika pengembangan dilakukan dengan menggunakan prinsip-prinsip rekayasa perangkat lunak. Dalam hal ini, menjadi mungkin untuk memecahkan masalah dengan usaha dan waktu yang hampir linier dalam ukuran program. Di sisi lain, jika program dapat ditulis secara otomatis oleh mesin, maka peningkatan usaha dan waktu dengan ukuran akan lebih dekat dengan peningkatan linier (garis putus-putus) dengan ukuran.



Gambar 1.4 Peningkatan waktu dan upaya pengembangan dengan ukuran masalah.

Sekarang mari kita coba memahami mengapa upaya yang diperlukan untuk mengembangkan program tumbuh secara eksponensial ketika gaya eksplorasi digunakan dan mengapa pendekatan untuk mengembangkan program ini benar-benar rusak ketika ukuran program menjadi besar? Untuk mendapatkan wawasan dari pertanyaan ini, kita perlu memiliki pengetahuan tentang keterbatasan kognitif manusia. Kompleksitas masalah yang dirasakan (atau psikologis) tumbuh secara eksponensial dengan ukurannya. Harap dicatat bahwa kompleksitas masalah yang dirasakan tidak terkait dengan masalah kompleksitas waktu atau ruang yang mungkin Anda kenal dari kursus dasar tentang algoritme.

Kompleksitas psikologis atau persepsi masalah menyangkut tingkat kesulitan yang dialami oleh seorang programmer saat memecahkan masalah menggunakan gaya pengembangan eksplorasi. Bahkan jika gaya eksplorasi menyebabkan kesulitan yang dirasakan dari suatu masalah tumbuh secara eksponensial karena keterbatasan kognitif manusia, bagaimana prinsip-prinsip rekayasa perangkat lunak membantu menahan kenaikan eksponensial dalam kompleksitas dengan ukuran masalah dan menahannya hingga hampir peningkatan linier? Prinsip rekayasa perangkat lunak membantu mencapai hal ini dengan memanfaatkan teknik abstraksi dan dekomposisi untuk mengatasi keterbatasan kognitif manusia. Anda mungkin masih bertanya-tanya bahwa ketika prinsip-prinsip rekayasa perangkat lunak digunakan, mengapa kurva tidak menjadi sepenuhnya linier? Jawabannya adalah sangat sulit untuk menerapkan prinsip-prinsip dekomposisi dan abstraksi untuk sepenuhnya mengatasi kompleksitas masalah.

Ringkasan kekurangan gaya eksplorasi pengembangan perangkat lunak:

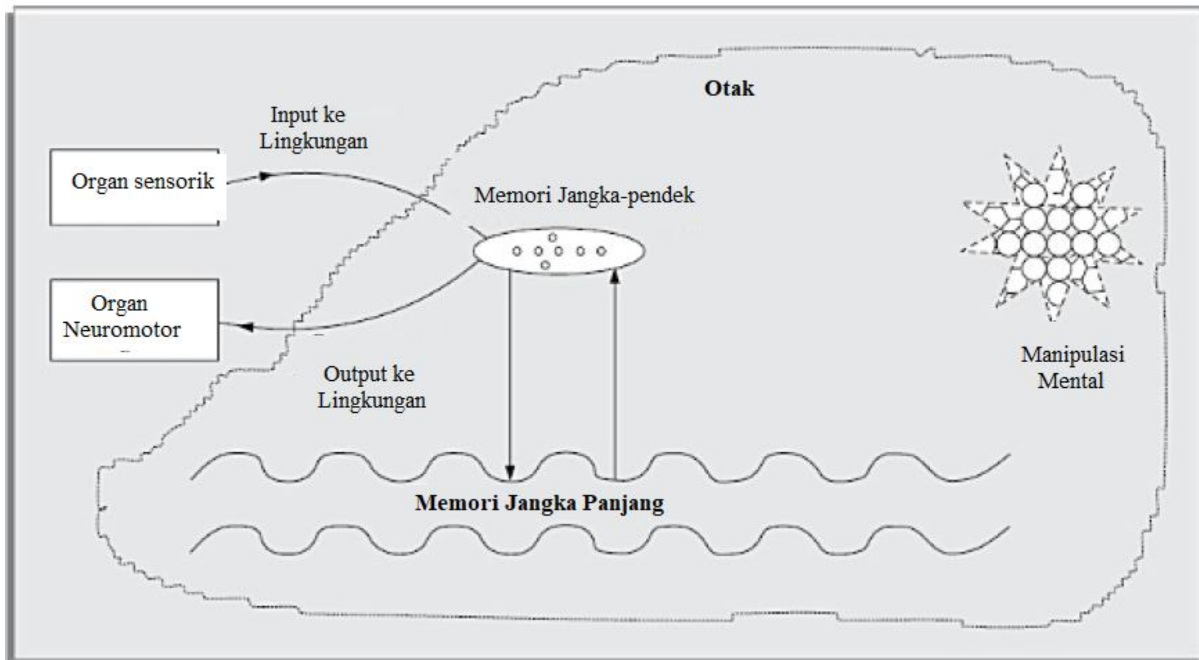
Kekurangan penting dalam menggunakan gaya pengembangan eksplorasi untuk mengembangkan perangkat lunak profesional diantaranya adalah:

- Kesulitan utama adalah pertumbuhan eksponensial waktu dan upaya pengembangan dengan ukuran masalah dan perangkat lunak berukuran besar menjadi hampir tidak mungkin menggunakan gaya pengembangan ini.
- Gaya eksplorasi biasanya menghasilkan kode yang tidak dapat dipelihara. Alasan untuk ini adalah bahwa kode apa pun yang dikembangkan tanpa desain yang tepat akan menghasilkan kode yang sangat tidak terstruktur dan berkualitas buruk.
- Menjadi sangat sulit untuk menggunakan gaya eksplorasi dalam lingkungan pengembangan tim. Dalam gaya eksplorasi, pekerjaan pengembangan dilakukan tanpa desain dan dokumentasi yang tepat. Oleh karena itu menjadi sangat sulit untuk mempartisi pekerjaan secara bermakna di antara sekumpulan developer yang dapat bekerja secara bersamaan. Di sisi lain, pengembangan tim sangat diperlukan untuk mengembangkan perangkat lunak modern—sebagian besar perangkat lunak mengamanatkan upaya pengembangan yang besar, yang memerlukan upaya tim untuk mengembangkannya. Selain kode berkualitas buruk, kurangnya dokumentasi yang tepat membuat pemeliharaan kode selanjutnya menjadi sangat sulit.

Kompleksitas Masalah yang Dirasakan:

Sebuah Interpretasi Berdasarkan Mekanisme Kognisi Manusia Peningkatan cepat dari kompleksitas yang dirasakan dari masalah dengan peningkatan ukuran masalah dapat dijelaskan dari interpretasi mekanisme kognisi manusia. Pemahaman sederhana tentang mekanisme kognitif manusia juga akan memberi kita wawasan tentang mengapa gaya pengembangan eksplorasi mengarah pada peningkatan waktu dan upaya yang tidak semestinya yang diperlukan untuk mengembangkan solusi pemrograman. Hal ini juga dapat menjelaskan mengapa menjadi praktis tidak layak untuk memecahkan masalah yang lebih besar dari ukuran tertentu saat menggunakan gaya eksplorasi; sedangkan menggunakan prinsip-prinsip rekayasa perangkat lunak, upaya yang diperlukan tumbuh hampir linier dengan ukuran (seperti yang ditunjukkan oleh garis tipis padat pada Gambar 1.4).

Psikolog mengatakan bahwa memori manusia dapat dianggap terdiri dari dua bagian yang berbeda[Miller 56]: ingatan jangka pendek dan jangka panjang. Representasi skematis dari kedua jenis ingatan ini dan perannya dalam mekanisme kognisi manusia telah ditunjukkan pada Gambar 1.5. Pada Gambar 1.5, blok berlabel organ sensorik mewakili panca indera manusia penglihatan, pendengaran, sentuhan, penciuman, dan rasa. Blok berlabel aktuator mewakili organ neuromotor seperti tangan, jari, kaki, dll.



Gambar 1.5 Model mekanisme kognisi manusia.

Memori jangka pendek: Memori jangka pendek, seperti namanya, dapat menyimpan informasi untuk sementara waktu—biasanya hingga beberapa detik, dan paling lama untuk beberapa menit. Memori jangka pendek juga kadang-kadang disebut sebagai memori kerja. Informasi yang disimpan dalam memori jangka pendek segera dapat diakses untuk diproses oleh otak. Memori jangka pendek rata-rata orang dapat menyimpan hingga tujuh item; tetapi dalam kasus ekstrim dapat bervariasi dari lima sampai sembilan item (7 ± 2). Seperti yang ditunjukkan pada Gambar 1.5, memori jangka pendek berpartisipasi dalam semua interaksi pikiran manusia dengan lingkungannya.

Harus jelas bahwa memori jangka pendek memainkan peran yang sangat penting dalam mekanisme kognisi manusia. Semua informasi yang dikumpulkan melalui organ sensorik pertama kali disimpan dalam memori jangka pendek. Memori jangka pendek juga digunakan oleh otak untuk menggerakkan organ-organ neuromotor. Unit manipulasi mental juga mendapatkan inputnya dari memori jangka pendek dan menyimpan kembali semua output yang dihasilkannya. Selanjutnya, informasi yang diambil dari memori jangka panjang pertama-tama disimpan dalam memori jangka pendek. Misalnya, jika Anda ditanya pertanyaan: "Jika sekarang jam 10 pagi, berapa jam yang tersisa hari ini?" Pertama, jam 10 pagi akan disimpan dalam memori jangka pendek. Selanjutnya, informasi bahwa satu hari berdurasi 24 jam akan diambil dari memori jangka panjang ke dalam memori jangka pendek. Unit manipulasi mental akan menghitung perbedaan (24-10), dan 14 jam akan disimpan dalam memori jangka pendek. Seperti yang Anda perhatikan, model ini sangat mirip dengan organisasi komputer dalam hal cache, memori utama, dan prosesor.

Item yang disimpan dalam memori jangka pendek bisa hilang baik karena pembusukan dengan waktu atau perpindahan oleh informasi yang lebih baru. Ini membatasi durasi item disimpan dalam memori jangka pendek hingga beberapa puluh detik. Namun, item n dapat disimpan lebih lama dalam memori jangka pendek dengan mendaur ulang. Artinya, ketika kita mengulang atau menyegarkan n item secara sadar, kita dapat mengingatnya untuk durasi yang lebih lama. Informasi tertentu disimpan dalam memori jangka pendek, dalam keadaan tertentu akan disimpan dalam memori jangka panjang.

Memori jangka panjang: Berbeda dengan memori jangka pendek, ukuran memori jangka panjang tidak diketahui memiliki batas atas yang pasti. Ukuran memori jangka panjang dapat bervariasi dari beberapa juta item hingga beberapa miliar item, sebagian besar tergantung pada seberapa aktif seseorang melatih kemampuan mentalnya. Suatu barang yang pernah disimpan dalam memori jangka panjang, biasanya disimpan selama beberapa tahun. Tapi, bagaimana item bisa disimpan dalam memori jangka panjang? Item yang ada dalam memori jangka pendek dapat disimpan dalam memori jangka panjang baik melalui sejumlah besar penyegaran (pengulangan) atau dengan membentuk tautan dengan item yang sudah ada dalam memori jangka panjang. Misalnya, Anda mungkin mengingat nomor telepon Anda sendiri karena Anda mungkin telah mengulangi (menyegarkan) nomor itu berkali-kali dalam ingatan jangka pendek Anda. Mari kita ambil contoh situasi di mana Anda dapat membentuk tautan ke item yang ada dalam memori jangka panjang untuk mengingat informasi tertentu. Misalkan Anda ingin mengingat 10 digit nomor ponsel 9433795369. Mengingatnya dengan hafalan mungkin menakutkan. Tapi, misalkan Anda menganggap nomor tersebut dibagi menjadi 9433 7953 69 dan perhatikan bahwa 94 adalah kode untuk BSNL, 33 adalah kode untuk Kolkata, misalkan 79 adalah tahun lahir Anda, dan 53 adalah nomor roll Anda, dan sisanya dua angka adalah setiapsatu kurang dari angka yang sesuai dari angka sebelumnya; Anda telah secara efektif membuat tautan dengan item yang sudah disimpan, membuatnya lebih mudah untuk mengingat nomornya.

Item: Sejauh ini kita hanya menyebutkan jumlah item yang dapat disimpan oleh memori jangka panjang dan jangka pendek. Tapi, apa sebenarnya item itu? Item adalah kumpulan informasi terkait. Menurut definisi ini, karakter seperti a atau digit seperti '5' masing-masing dapat dianggap sebagai item. Sebuah kata, kalimat, cerita, atau bahkan gambar masing-masing dapat menjadi satu item. Setiap item biasanya menempati satu tempat dalam memori. Definisi item sebagai kumpulan informasi terkait menyiratkan bahwa ketika Anda dapat membuat beberapa hubungan sederhana antara beberapa item yang berbeda, informasi yang biasanya menempati beberapa tempat dapat disimpan hanya dengan menggunakan satu tempat dalam memori. Fenomena terbentuknya satu item dari beberapa item inilah yang disebut sebagai chunking oleh para psikolog. Misalnya, jika Anda diberi bilangan biner 110010101001—mungkin terbukti sangat sulit bagi Anda untuk memahami dan mengingatnya. Namun, bentuk oktal dari bilangan 6251 (yaitu, representasi (110)(010)(101)(001)) mungkin lebih mudah untuk dipahami dan diingat karena kita telah berhasil membuat potongan masing-masing tiga item.

Bukti memori jangka pendek: Bukti memori jangka pendek memanifestasikan dirinya dalam banyak pengalaman kita sehari-hari. Sebagai contoh memori jangka pendek, perhatikan situasi berikut. Misalkan, Anda mencari nomor dari direktori telepon dan mulai memutar nomor itu. Jika Anda menemukan nomor tersebut sedang sibuk, Anda akan menghubungi nomor itu lagi setelah beberapa detik—dalam hal ini, Anda akan dapat melakukannya dengan mudah tanpa harus mencari direktori. Namun, setelah beberapa jam atau hari sejak Anda menekan nomor terakhir kali, Anda mungkin tidak mengingat nomor tersebut sama sekali, dan perlu melihat direktori lagi.

Angka ajaib 7: Miller menyebut angka tujuh sebagai angka ajaib [Miller 56] karena jika seseorang berurusan dengan tujuh atau kurang jumlah informasi yang tidak terkait sekaligus, ini akan dengan mudah diakomodasi dalam memori jangka pendek. Jadi, dia bisa dengan mudah memahaminya. Karena jumlah item yang harus ditangani seseorang meningkat melebihi tujuh, menjadi sangat sulit untuk memahaminya. Pengamatan ini dapat dengan mudah diperluas untuk menulis program. Ketika jumlah detail (atau variabel) yang harus dilacak seseorang untuk memecahkan masalah meningkat melebihi tujuh, itu melebihi

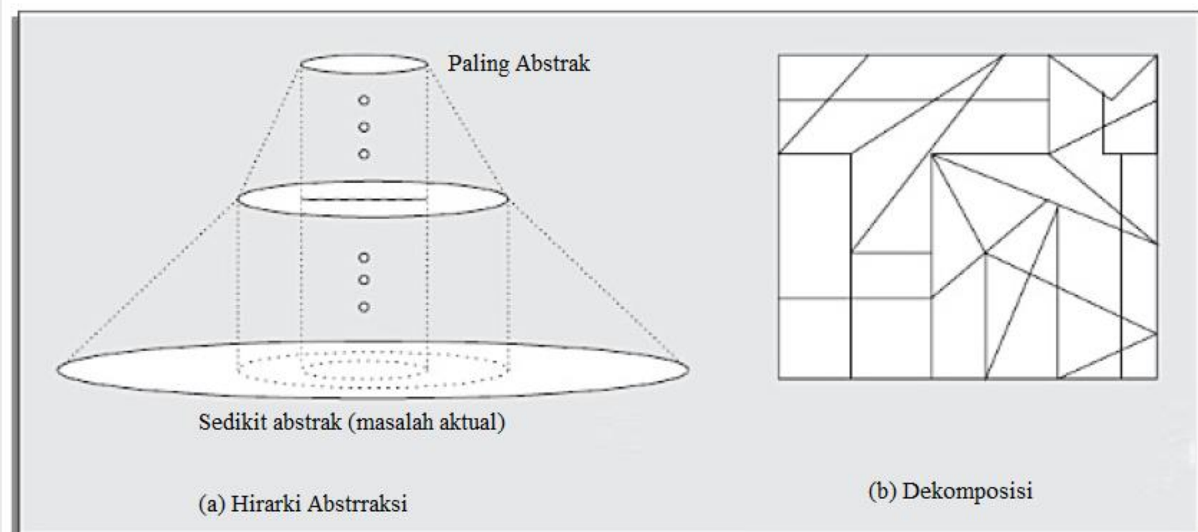
kapasitas memori jangka pendek dan menjadi jauh lebih sulit bagi pikiran manusia untuk memahami masalahnya.

Sebuah program kecil yang hanya memiliki beberapa variabel dapat dengan mudah dipahami oleh seorang individu. Ketika jumlah variabel independen dalam program meningkat, itu dengan cepat melebihi daya tangkap individu dan akan membutuhkan upaya yang terlalu besar untuk menguasai masalah. Ini menguraikan kemungkinan alasan di balik sifat eksponensial dari plot ukuran usaha (garis tebal) yang ditunjukkan pada Gambar 1.4. Harap dicatat bahwa situasi yang digambarkan pada Gambar 1.4 muncul sebagian besar karena keterbatasan kognitif manusia. Alih-alih manusia, jika mesin dapat menulis (menghasilkan) sebuah program, kemiringan kurva akan menjadi linier, karena ukuran cache (memori jangka pendek) komputer cukup besar. Tetapi, mengapa kurva ukuran usaha menjadi hampir linier ketika prinsip-prinsip rekayasa perangkat lunak diterapkan? Ini karena prinsip-prinsip rekayasa perangkat lunak secara ekstensif menggunakan teknik-teknik yang dirancang khusus untuk mengatasi keterbatasan kognitif manusia

Prinsip yang Diterapkan oleh Rekayasa Perangkat Lunak untuk Mengatasi *Keterbatasan Kognitif Manusia*

Kita akan melihat di seluruh buku ini bahwa tema sentral dari sebagian besar prinsip rekayasa perangkat lunak adalah penggunaan teknik untuk secara efektif mengatasi masalah yang muncul karena keterbatasan kognitif manusia. Dua prinsip penting yang digunakan oleh rekayasa perangkat lunak untuk mengatasi masalah yang timbul karena keterbatasan kognitif manusia adalah—abstraksi dan dekomposisi.

Dengan bantuan Gambar 1.6(a) dan (b) saya akan menjelaskan tentang esensi dari dua prinsip penting ini dan bagaimana mereka membantu mengatasi keterbatasan kognitif manusia. Dalam sisa buku ini, kita akan berulang kali menemukan penggunaan dua prinsip dasar ini dalam berbagai bentuk dan rasa dalam aktivitas pengembangan perangkat lunak yang berbeda. Oleh karena itu, pemahaman menyeluruh tentang kedua prinsip ini diperlukan.



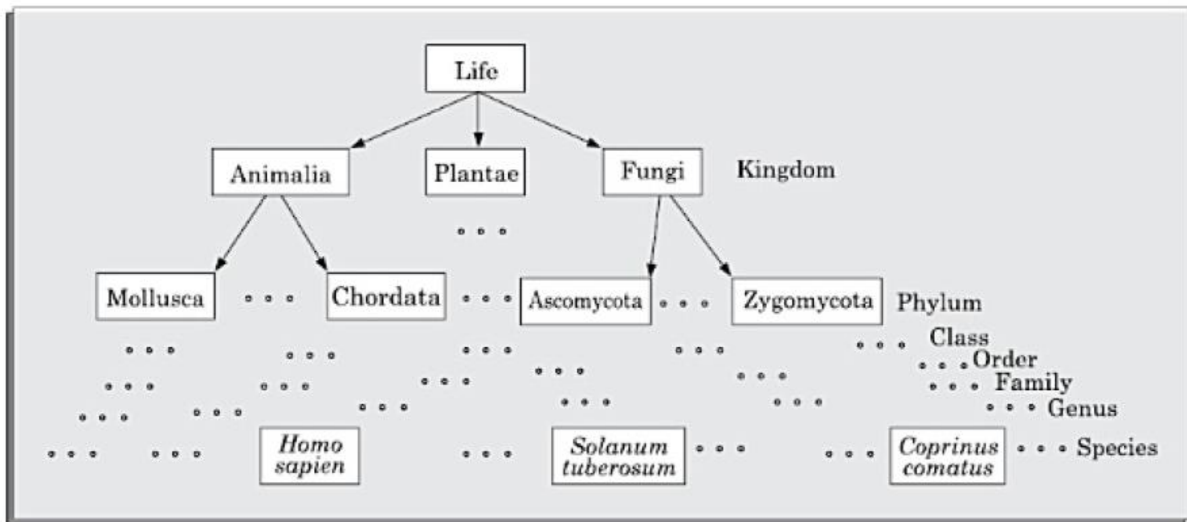
Gambar 1.6 Representasi skematis.

Abstraksi

Abstraksi mengacu pada konstruksi versi masalah yang lebih sederhana dengan mengabaikan detailnya. Prinsip membangun abstraksi dikenal sebagai pemodelan (atau konstruksi model). Abstraksi adalah penyederhanaan suatu masalah dengan memusatkan perhatian hanya pada satu aspek dari masalah sementara mengabaikan semua aspek lainnya. Saat menggunakan prinsip abstraksi untuk memahami masalah yang kompleks, kita

memusatkan perhatian kita hanya pada satu atau dua aspek spesifik dari masalah dan mengabaikan sisanya. Setiap kali menghilangkan beberapa detail masalah untuk membangun abstraksi, kita membangun model masalah. Dalam kehidupan sehari-hari, kita sering menggunakan prinsip abstraksi untuk memahami suatu masalah atau menilai suatu situasi. Perhatikan dua contoh berikut.

- Misalkan Anda diminta untuk mengembangkan pemahaman menyeluruh tentang beberapa negara. Tak seorang pun waras akan memulai tugas ini dengan bertemu semua warga negara, mengunjungi setiap rumah, dan memeriksa setiap pohon negara, dll Anda mungkin akan mengambil bantuan beberapa jenis abstraksi untuk melakukan hal ini. Anda mungkin akan mulai dengan merujuk dan memahami berbagai jenis peta untuk negara itu. Sebuah peta, pada kenyataannya, adalah representasi abstrak dari suatu negara. Ini mengabaikan informasi rinci seperti orang-orang tertentu yang menghuninya, rumah, sekolah, taman bermain, pohon, dll. Sekali lagi, ada dua jenis peta yang penting—peta fisik dan peta politik. Peta fisik menunjukkan ciri-ciri fisik suatu daerah; seperti gunung, danau, sungai, garis pantai, dan sebagainya. Di sisi lain, peta politik menunjukkan negara bagian, ibu kota, dan batas negara, dll. Peta fisik adalah model abstrak negara dan mengabaikan batas negara bagian dan distrik. Peta politik, di sisi lain, adalah abstraksi lain dari negara yang mengabaikan karakteristik fisik seperti ketinggian tanah, vegetasi, dll. Dapat dilihat bahwa, untuk objek yang sama (misalnya negara), beberapa abstraksi dimungkinkan. Dalam setiap abstraksi, beberapa aspek objek diabaikan. Kita memahami masalah dengan mengabstraksikan berbagai aspek masalah (membangun berbagai jenis model) dan memahaminya. Tidak terlalu sulit untuk menyadari bahwa penggunaan yang tepat dari prinsip abstraksi dapat menjadi bantuan yang sangat efektif untuk menguasai bahkan masalah yang menakutkan.
- Pertimbangkan situasi berikut. Misalkan Anda diminta untuk mengembangkan pemahaman tentang semua makhluk hidup yang menghuni bumi. Jika Anda menggunakan pendekatan naif, Anda akan mulai mengambil satu demi satu makhluk hidup yang menghuni bumi dan mulai memahami mereka. Bahkan setelah melakukan upaya yang luar biasa, Anda akan membuat sedikit kemajuan dan dibiarkan bingung karena ada miliaran makhluk hidup di bumi dan informasinya akan terlalu banyak untuk ditangani oleh siapa pun. Sebaliknya, yang dapat dilakukan adalah membangun dan memahami hierarki abstraksi semua makhluk hidup seperti yang ditunjukkan pada Gambar 1.7. Di tingkat atas, kita memahami bahwa pada dasarnya ada tiga jenis makhluk hidup yang berbeda secara mendasar—tumbuhan, hewan, dan jamur. Perlahan-lahan lebih banyak detail ditambahkan tentang setiap jenis pada setiap tingkat berturut-turut, sampai kita mencapai tingkat spesies yang berbeda pada tingkat daun dari pohon abstraksi.



Gambar 1.7 Hirarki abstraksi yang mengklasifikasikan organisme hidup.

Satu tingkat abstraksi bisa cukup untuk masalah yang agak sederhana. Namun, masalah yang lebih kompleks perlu dimodelkan sebagai hierarki abstraksi. Representasi skematis dari hierarki abstraksi telah ditunjukkan pada Gambar 1.6(a). Representasi yang paling abstrak hanya akan memiliki beberapa item dan akan menjadi yang paling mudah untuk dipahami. Setelah seseorang memahami representasi paling sederhana, seseorang akan mencoba memahami tingkat abstraksi berikutnya di mana paling banyak lima atau tujuh informasi baru ditambahkan dan seterusnya sampai tingkat terendah dipahami. Pada saat, seseorang mencapai level terendah, dia akan menguasai seluruh masalah.

Dekomposisi

Dekomposisi adalah prinsip penting lain yang tersedia dalam repertoar seorang insinyur perangkat lunak untuk menangani kompleksitas masalah. Prinsip ini banyak digunakan oleh beberapa teknik rekayasa perangkat lunak untuk menampung pertumbuhan eksponensial dari kompleksitas masalah yang dirasakan. Prinsip dekomposisi dikenal sebagai prinsip membagi dan menaklukkan. Prinsip dekomposisi menganjurkan penguraian masalah menjadi banyak bagian kecil yang independen. Bagian-bagian kecil kemudian diambil satu per satu dan diselesaikan secara terpisah. Idenya adalah bahwa setiap bagian kecil akan mudah dipahami dan dipahami dan dapat dengan mudah dipecahkan. Masalah penuh terpecahkan ketika semua bagian diselesaikan.

Cara yang populer untuk mendemonstrasikan prinsip dekomposisi adalah dengan mencoba mematahkan sekelompok besar tongkat yang diikat menjadi satu dan kemudian mematahkannya satu per satu. Gambar 1.6(b) menunjukkan dekomposisi masalah besar menjadi banyak bagian kecil. Namun, sangat penting untuk dipahami bahwa penguraian masalah yang sewenang-wenang menjadi bagian-bagian kecil tidak akan membantu. Bagian yang berbeda setelah dekomposisi harus lebih atau kurang independen satu sama lain. Artinya, untuk menyelesaikan satu bagian tidak harus mengacu dan memahami bagian lain. Jika untuk menyelesaikan satu bagian Anda harus memahami bagian lain, maka ini akan bermuara pada pemahaman semua bagian bersama-sama. Ini akan secara efektif mengurangi masalah ke masalah awal sebelum dekomposisi (kasus ketika semua tongkat diikat menjadi satu). Oleh karena itu, tidak cukup hanya menguraikan masalah dengan cara apa pun, tetapi penguraian harus sedemikian rupa sehingga bagian-bagian berbeda yang terurai harus lebih atau kurang independen satu sama lain.

Sebagai contoh penggunaan prinsip dekomposisi, perhatikan hal berikut. Anda akan memahami sebuah buku dengan lebih baik ketika isinya diuraikan (diatur) menjadi bab-bab yang kurang lebih independen. Artinya, setiap bab berfokus pada topik yang terpisah, bukan ketika buku mencampur semua topik bersama-sama di seluruh halaman. Demikian pula, setiap bab harus diuraikan menjadi beberapa bagian sehingga setiap bagian membahas masalah yang berbeda. Setiap bagian harus didekomposisi menjadi subbagian dan seterusnya. Jika berbagai subbagian hampir tidak tergantung satu sama lain, subbagian dapat dipahami satu per satu daripada terus merujuk silang ke berbagai subbagian di seluruh buku untuk memahaminya.

Mengapa mempelajari rekayasa perangkat lunak?

Mari kita periksa keterampilan yang dapat Anda peroleh dari mempelajari prinsip-prinsip rekayasa perangkat lunak. Dua keterampilan berikut ini mungkin merupakan keterampilan terpenting yang dapat Anda peroleh setelah menyelesaikan studi rekayasa perangkat lunak:

- Keterampilan untuk berpartisipasi dalam pengembangan perangkat lunak besar. Anda dapat berpartisipasi secara bermakna dalam upaya tim untuk mengembangkan perangkat lunak besar hanya setelah mempelajari teknik sistematis yang digunakan dalam industri.
- Anda akan belajar bagaimana menangani kompleksitas secara efektif dalam masalah pengembangan perangkat lunak. Secara khusus, Anda akan belajar bagaimana menerapkan prinsip-prinsip abstraksi dan dekomposisi untuk menangani kompleksitas selama berbagai tahap dalam pengembangan perangkat lunak seperti spesifikasi, desain, konstruksi, dan pengujian.

Selain dua keterampilan penting di atas, Anda juga akan mempelajari teknik pengembangan antarmuka pengguna spesifikasi persyaratan perangkat lunak, jaminan kualitas, pengujian, manajemen proyek, pemeliharaan, dll. Program kecil juga dapat ditulis tanpa menggunakan rekayasa perangkat lunak. prinsip. Namun bahkan jika Anda berniat untuk menulis program kecil, prinsip-prinsip rekayasa perangkat lunak dapat membantu Anda untuk mencapai produktivitas yang lebih tinggi dan pada saat yang sama memungkinkan Anda untuk menghasilkan program dengan kualitas yang lebih baik.

1.4 MUNCULNYA TEKNIK PERANGKAT LUNAK

Teknik rekayasa perangkat lunak telah berkembang selama bertahun-tahun di masa lalu. Evolusi ini merupakan hasil dari serangkaian inovasi dan akumulasi pengalaman tentang penulisan program yang berkualitas. Karena inovasi dan pengalaman pemrograman ini terlalu banyak, mari kita periksa secara singkat hanya beberapa dari inovasi dan pengalaman pemrograman ini yang telah berkontribusi pada pengembangan disiplin rekayasa perangkat lunak.

Pemrograman Komputer Awal

Komputer komersial awal sangat lambat dan terlalu dasar dibandingkan dengan standar saat ini. Bahkan tugas pemrosesan sederhana membutuhkan waktu komputasi yang cukup lama pada komputer tersebut. Tidak heran jika program-program pada waktu itu berukuran sangat kecil dan kurang canggih. Program-program tersebut biasanya ditulis dalam bahasa assembly. Panjang program biasanya terbatas pada sekitar beberapa ratus baris kode rakitan monolitik. Setiap programmer mengembangkan gaya individualistisnya sendiri dalam menulis program sesuai dengan intuisinya dan menggunakan gaya ini ad hoc saat menulis program yang berbeda. Dengan kata sederhana, programmer menulis program tanpa merumuskan strategi solusi yang tepat, merencanakan, atau merancang lompatan ke terminal dan mulai mengkode segera setelah mengetahui masalahnya. Mereka kemudian terus

memperbaiki masalah yang mereka amati sampai mereka memiliki program yang bekerja dengan cukup baik.

Pemrograman Bahasa Tingkat Tinggi

Komputer menjadi lebih cepat dengan diperkenalkannya teknologi semikonduktor pada awal 1960-an. Transistor semikonduktor yang lebih cepat menggantikan sirkuit berbasis tabung vakum yang lazim di komputer. Dengan ketersediaan komputer yang lebih kuat, menjadi mungkin untuk memecahkan masalah yang lebih besar dan lebih kompleks. Pada saat ini, bahasa tingkat tinggi seperti FORTRAN, ALGOL, dan COBOL diperkenalkan. Ini sangat mengurangi upaya yang diperlukan untuk mengembangkan perangkat lunak dan membantu programmer untuk menulis program yang lebih besar (mengapa?). Menulis setiap konstruksi pemrograman tingkat tinggi yang berlaku memungkinkan programmer untuk menulis beberapa instruksi mesin. Juga, detail mesin (register, flag, dll.) diabstraksi dari programmer. Namun, programmer masih menggunakan gaya pengembangan perangkat lunak eksplorasi. Program tipikal terbatas pada ukuran sekitar beberapa ribu baris kode sumber.

Desain Berbasis Aliran Kontrol

Karena ukuran dan kompleksitas program terus meningkat, gaya pemrograman eksplorasi terbukti tidak cukup. Programmer merasa semakin sulit tidak hanya untuk menulis program yang hemat biaya dan benar, tetapi juga untuk memahami dan memelihara program yang ditulis oleh orang lain. Untuk mengatasi masalah ini, programmer berpengalaman menyarankan programmer lain untuk memberi perhatian khusus pada desain struktur aliran kontrol program.

Struktur aliran kontrol program menunjukkan urutan di mana instruksi program dieksekusi. Untuk membantu mengembangkan program yang memiliki struktur aliran kontrol yang baik, dikembangkan teknik diagram alir. Bahkan saat ini, teknik flow charting digunakan untuk merepresentasikan dan merancang algoritma; meskipun popularitas diagram alir mewakili dan program desain sangat ingin karena munculnya teknik yang lebih maju.

Gambar 1.8 mengilustrasikan dua cara alternatif penulisan kode program untuk masalah yang sama. Representasi diagram alir untuk dua segmen program pada Gambar 1.8 digambarkan pada Gambar 1.9. Perhatikan bahwa struktur aliran kontrol segmen program pada Gambar 1.9(b) jauh lebih sederhana daripada Gambar 1.9(a). Dengan memeriksa kode, dapat dilihat bahwa Gambar 1.9(a) jauh lebih sulit untuk dipahami dibandingkan dengan Gambar 1.9(b). Contoh ini menguatkan fakta bahwa jika representasi diagram alur sederhana, maka kode yang sesuai harus sederhana. Anda dapat menggambar representasi diagram alir dari beberapa masalah lain untuk meyakinkan diri sendiri bahwa program dengan representasi diagram alir yang kompleks memang lebih sulit untuk dipahami dan dipelihara.

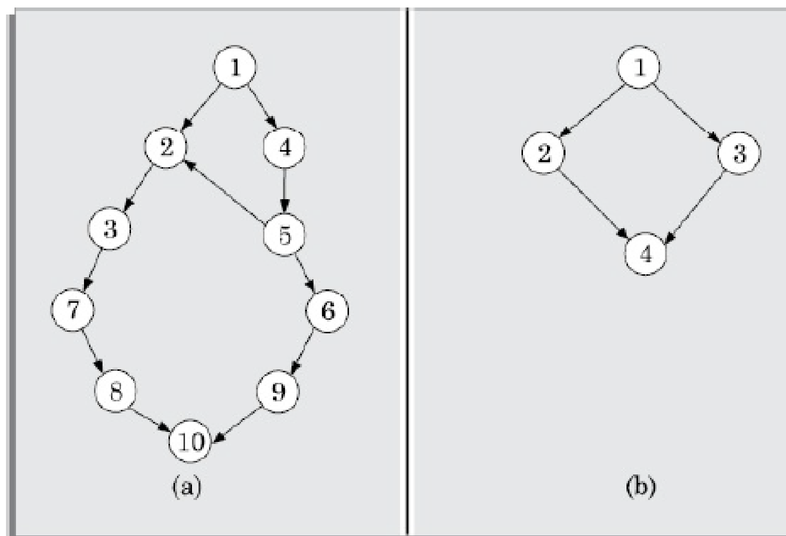
```
1        if(customer_savings_balance>withdrawal_request) {
2   100:      issue_money=TRUE;
3             GOTO 110;
         }
4        else if(privileged_customer==TRUE)
5             GOTO 100;
6        else GOTO 120;
7   110: activate_cash_dispenser(withdrawal_request);
8        GOTO 130;
9   120:      print(error);
10  130:      end-transaction();
```

(a) Contoh program tidak terstruktur

```
1   if(privileged_customer||(customer_savings_balance>withdrawal_request)){
2           activate_cash_dispenser(withdrawal_request);
    }
3   else print(error);
4   end-transaction();
```

(b) Program struktur yang sesuai

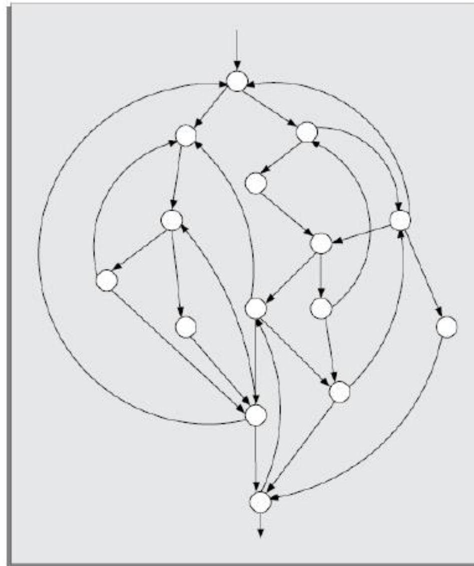
Gambar 1.8 Contoh (a) Program tidak terstruktur (b) Program terstruktur yang sesuai.



Gambar 1.9 Grafik aliran kendali program pada Gambar 1.8(a) dan (b).

Sekarang mari kita coba memahami mengapa program yang memiliki struktur aliran kontrol yang baik akan lebih mudah untuk dikembangkan dan dipahami. Dengan kata lain, mari kita pahami mengapa program dengan representasi diagram alir yang kompleks sulit dipahami? Alasan utama di balik situasi ini adalah bahwa biasanya seseorang memahami program dengan secara mental menelusuri urutan eksekusinya (yaitu urutan pernyataan) untuk memahami bagaimana output dihasilkan dari nilai input. Artinya, kita bisa mulai dari pernyataan yang menghasilkan output, dan menelusuri kembali pernyataan dalam program dan memahami bagaimana mereka menghasilkan output dengan mentransformasikan data input.

Sebagai alternatif, kita dapat memulai dengan data input dan memeriksa dengan menjalankan program bagaimana setiap pernyataan memproses (mengubah) data input hingga output dihasilkan. Misalnya, untuk program pada Gambar 1.9(a) Anda harus memahami eksekusi program sepanjang jalur 1-2-3-7-8-10, 1-4-5-6-9-10, dan 1- 4-5-2-3-7-8-10. Sebuah program yang memiliki struktur aliran kontrol yang berantakan (yaitu diagram alir), akan memiliki sejumlah besar jalur eksekusi (lihat Gambar 1.10). Akibatnya, akan menjadi sangat sulit untuk menentukan semua jalur eksekusi, dan menelusuri urutan eksekusi di sepanjang semua jalur yang mencoba memahaminya bisa menjadi mimpi buruk. Oleh karena itu jelas bahwa program yang memiliki representasi diagram alur yang berantakan memang akan sulit untuk dipahami dan di-debug.



Gambar 1.10 CFG dari sebuah program yang memiliki terlalu banyak pernyataan GO TO.

Apakah pernyataan GO TO pelakunya?

Dalam sebuah makalah penting, Dijkstra [1968] menerbitkan artikelnya (sekarang terkenal) "GO TO Statements Dianggap Berbahaya". Dia menunjukkan bahwa penggunaan pernyataan GO TO yang tidak terkendali adalah penyebab utama dalam membuat struktur kontrol suatu program menjadi berantakan. Untuk memahami argumennya, periksa Gambar 1.10 yang menunjukkan diagram alir representasi program di mana programmer telah menggunakan terlalu banyak pernyataan GO TO. Pernyataan GO TO mengubah aliran kontrol secara sewenang-wenang, menghasilkan terlalu banyak jalur. Tapi, lalu mengapa penggunaan pernyataan GO TO yang terlalu banyak membuat program sulit dimengerti?

Seorang programmer yang mencoba memahami sebuah program harus secara mental melacak dan memahami pemrosesan yang terjadi di sepanjang jalur program membuat pemahaman dan debugging program menjadi sangat rumit. Segera menjadi diterima secara luas bahwa program yang baik harus memiliki struktur kontrol yang sangat sederhana. Dimungkinkan untuk membedakan program yang baik dari program yang buruk hanya dengan memeriksa representasi diagram alirnya secara visual. Penggunaan diagram alir untuk merancang struktur aliran kontrol yang baik dari program menjadi tersebar luas.

Pemrograman terstruktur—ekstensi logis

Kebutuhan untuk membatasi penggunaan pernyataan GO TO diakui oleh semua orang. Namun, banyak programmer masih menggunakan bahasa assembly. Instruksi JUMP sering digunakan untuk percabangan program dalam bahasa assembly. Oleh karena itu, programmer dengan latar belakang pemrograman bahasa assembly menganggap penggunaan pernyataan GO TO dalam program tidak dapat dihindari. Namun, secara meyakinkan dibuktikan oleh Bohm dan Jacopini bahwa hanya tiga konstruksi pemrograman — urutan, seleksi, dan iterasi — yang cukup untuk mengekspresikan logika pemrograman apa pun. Ini adalah hasil yang penting—hal ini dianggap penting bahkan hingga hari ini. Contoh pernyataan urutan adalah pernyataan penugasan dalam bentuk a=b;. Contoh pernyataan seleksi dan iterasi masing-masing adalah pernyataan if-then-else dan do-while. Secara bertahap, semua orang menerima bahwa memang mungkin untuk memecahkan masalah pemrograman apa pun tanpa menggunakan pernyataan GO TO dan penggunaan pernyataan GO TO yang sembarangan harus dihindari. Ini membentuk dasar metodologi pemrograman terstruktur.

Program terstruktur menghindari aliran kontrol tidak terstruktur dengan membatasi penggunaan pernyataan GO TO. Pemrograman terstruktur difasilitasi, jika bahasa pemrograman yang digunakan mendukung konstruksi program single-entry, single-exit seperti if-then-else, do-while, dll. Dengan demikian, fitur penting dari program terstruktur adalah desain struktur kontrol yang baik. Contoh yang menggambarkan perbedaan utama antara program terstruktur dan tidak terstruktur ini ditunjukkan pada Gambar 1.8. Program pada Gambar 1.8(a) menggunakan terlalu banyak pernyataan GO TO, sedangkan program pada Gambar 1.8(b) tidak menggunakan satu pun. Bagan alir program yang menggunakan pernyataan GO TO jelas jauh lebih kompleks seperti yang dapat dilihat pada Gambar 1.9.

Selain aspek struktur kontrol, istilah program terstruktur juga digunakan untuk menunjukkan beberapa fitur program lainnya. Sebuah program terstruktur harus modular. Program modular adalah program yang didekomposisi menjadi satu set modul1 sehingga modul-modul tersebut harus memiliki saling ketergantungan yang rendah satu sama lain.

Tapi, apa kelebihan utama menulis program terstruktur dibandingkan dengan yang tidak terstruktur? Pengalaman penelitian telah menunjukkan bahwa programmer melakukan lebih sedikit kesalahan saat menggunakan pernyataan terstruktur if-then-else dan do-while daripada saat menggunakan konstruksi kode uji-dan-cabang. Selain kurang rawan kesalahan, program terstruktur biasanya lebih mudah dibaca, lebih mudah dirawat, dan membutuhkan lebih sedikit usaha untuk dikembangkan dibandingkan dengan program tidak terstruktur. Keutamaan pemrograman terstruktur menjadi diterima secara luas dan konsep pemrograman terstruktur digunakan bahkan sampai hari ini. Namun, pelanggaran terhadap fitur pemrograman terstruktur biasanya diizinkan dalam situasi pemrograman tertentu tertentu, seperti penanganan pengecualian, dll.

Segera beberapa bahasa seperti *PASCAL, MODULA, C*, dll, menjadi tersedia yang secara khusus dirancang untuk mendukung pemrograman terstruktur. Bahasa pemrograman ini memfasilitasi penulisan program modular dan program yang memiliki struktur kontrol yang baik. Oleh karena itu, struktur kontrol yang berantakan tidak lagi menjadi masalah besar. Jadi, fokusnya bergeser dari merancang struktur kontrol yang baik ke merancang struktur data yang baik untuk program.

Desain Berorientasi Struktur Data

Komputer menjadi lebih kuat dengan munculnya sirkuit terpadu (IC) di awal tahun tujuh puluhan. Ini sekarang dapat digunakan untuk memecahkan masalah yang lebih kompleks. developer perangkat lunak ditugaskan untuk mengembangkan perangkat lunak yang lebih besar dan lebih rumit. yang seringkali membutuhkan penulisan lebih dari beberapa puluh ribu baris kode sumber. Teknik pengembangan program berbasis aliran kontrol tidak dapat digunakan dengan memuaskan lagi untuk menulis program tersebut, dan diperlukan teknik pengembangan program yang lebih efektif.

Segera diketahui bahwa ketika mengembangkan sebuah program, jauh lebih penting untuk memperhatikan desain struktur data penting dari program daripada desain struktur kontrolnya. Teknik desain berdasarkan prinsip ini disebut teknik desain berorientasi struktur data. Menggunakan teknik desain berorientasi struktur data, pertama-tama struktur data program dirancang. Struktur kode dirancang berdasarkan struktur data.

Pada langkah selanjutnya, desain program diturunkan dari struktur data. Contoh teknik desain berorientasi struktur data adalah teknik Jackson's Structured Programming (JSP) yang dikembangkan oleh Michael Jackson [1975]. Dalam metodologi JSP, struktur data program pertama dirancang menggunakan notasi untuk urutan, seleksi, dan iterasi. Metodologi JSP menyediakan teknik yang menarik untuk mendapatkan struktur program dari representasi struktur datanya. Beberapa teknik desain berbasis struktur data lainnya juga dikembangkan.

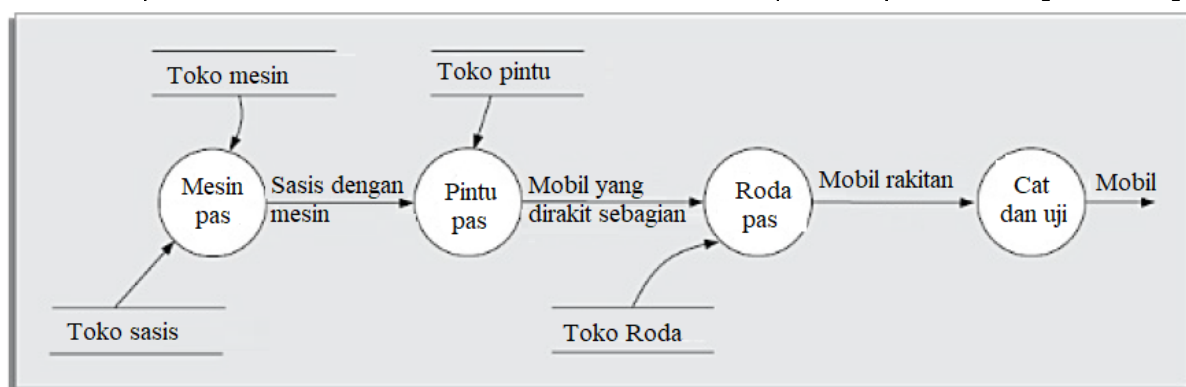
Beberapa teknik ini menjadi sangat populer dan banyak digunakan. Teknik lain yang perlu disebutkan secara khusus adalah Metodologi Warnier-Orr [1977, 1981]. Disini kita tidak akan membahas teknik ini dalam teks ini karena sekarang teknik ini jarang digunakan di industri dan telah digantikan oleh teknik berbasis aliran data dan berorientasi objek.

Desain Berorientasi Aliran Data

Ketika komputer menjadi lebih cepat dan lebih kuat dengan pengenalan Sirkuit terintegrasi skala sangat besar (VLSI) dan beberapa konsep arsitektur baru, perangkat lunak yang lebih kompleks dan canggih diperlukan untuk memecahkan masalah yang lebih menantang. Oleh karena itu, developer perangkat lunak mencari teknik yang lebih efektif untuk merancang perangkat lunak dan segera diusulkan teknik berorientasi aliran. Teknik berorientasi aliran data menganjurkan bahwa item data utama yang ditangani oleh suatu sistem harus diidentifikasi dan pemrosesan yang diperlukan pada item data ini untuk menghasilkan output yang diinginkan harus ditentukan. Fungsi (juga disebut sebagai proses) dan item data yang dipertukarkan antara fungsi yang berbeda direpresentasikan dalam diagram yang dikenal sebagai diagram aliran data (DFD). Struktur program dapat dirancang dari representasi masalah DFD.

DFD: Representasi program penting untuk desain program prosedural

DFD telah terbukti menjadi teknik generik yang digunakan untuk memodelkan semua jenis sistem, dan bukan hanya sistem perangkat lunak. Misalnya, Gambar 1.11 menunjukkan representasi aliran data dari pabrik perakitan mobil otomatis. Jika Anda belum pernah mengunjungi pabrik perakitan mobil otomatis, deskripsi singkat tentang pabrik perakitan mobil otomatis akan diperlukan. Di pabrik perakitan mobil otomatis, ada beberapa stasiun pemrosesan (juga disebut stasiun kerja) yang terletak di sepanjang sisi ban berjalan (juga disebut jalur perakitan). Setiap stasiun kerja dikhususkan untuk melakukan pekerjaan seperti pemasangan roda, pemasangan mesin, pengecatan mobil, dll. Jika program yang dirakit sebagian bergerak di sepanjang jalur perakitan, stasiun kerja yang berbeda melakukan pekerjaan masing-masing pada perangkat lunak yang dirakit sebagian. Setiap lingkaran dalam model DFD pada Gambar 1.11 mewakili sebuah workstation (disebut proses atau gelembung).



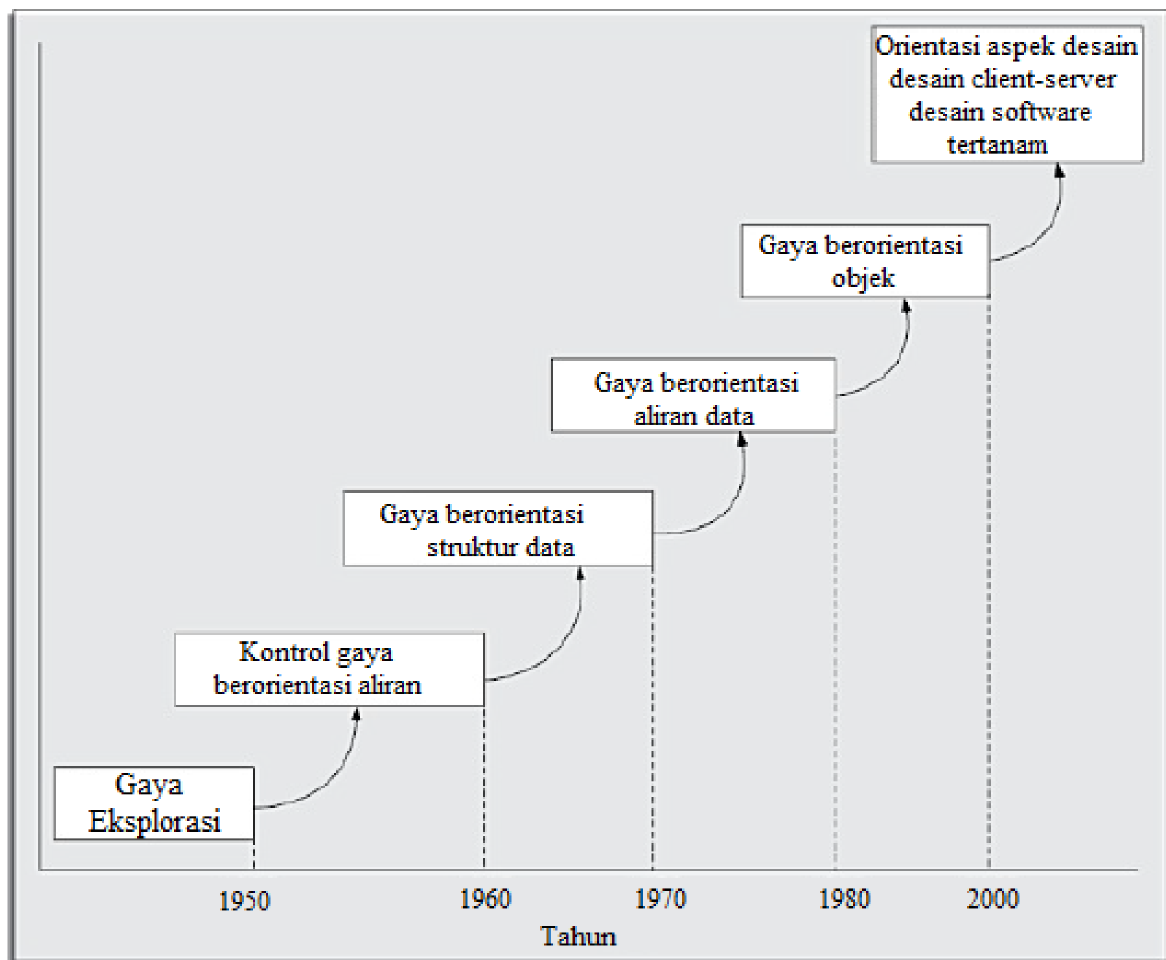
Gambar 1.11 Model aliran data pabrik perakitan mobil.

Setiap workstation mengkonsumsi item input tertentu dan menghasilkan item output tertentu. Saat mobil yang sedang dirakit tiba di stasiun kerja, ia mengambil barang-barang yang diperlukan untuk dipasang dari toko yang sesuai (diwakili oleh dua garis horizontal paralel), dan segera setelah pekerjaan pemasangan selesai diteruskan ke stasiun kerja berikutnya. Sangat mudah untuk memahami model DFD dari pabrik perakitan mobil yang ditunjukkan pada Gambar 1.11 bahkan tanpa mengetahui apa pun tentang DFD. Dalam hal ini, kita dapat mengatakan bahwa keuntungan utama dari DFD adalah kesederhanaannya. Dalam Bab 6, kita akan mempelajari bagaimana membangun model DFD dari sistem perangkat lunak.

Setelah Anda mengembangkan model DFD masalah, teknik desain berorientasi aliran data menyediakan metodologi yang agak lurus ke depan untuk mengubah representasi DFD masalah menjadi desain perangkat lunak yang sesuai. Kita akan mempelajari teknik desain berbasis aliran data di Bab 6.

Desain berorientasi objek

Teknik berorientasi aliran data berkembang menjadi teknik desain berorientasi objek (OOD) di akhir tahun tujuh puluhan. Teknik desain berorientasi objek adalah pendekatan yang menarik secara intuitif, di mana objek alami (seperti karyawan, daftar gaji, dll.) yang relevan dengan masalah pertama kali diidentifikasi dan kemudian hubungan di antara objek seperti komposisi, referensi, dan warisan ditentukan. Setiap objek pada dasarnya bertindak sebagai entitas penyembunyi data (juga dikenal sebagai abstraksi data). Teknik berorientasi objek telah diterima secara luas karena kesederhanaannya, ruang lingkup untuk penggunaan kembali kode dan desain, janji waktu pengembangan yang lebih rendah, biaya pengembangan yang lebih rendah, kode yang lebih kuat, dan perawatan yang lebih mudah. Teknik OOD dibahas dalam Bab 7 dan 8.



Gambar 1.12 Evolusi teknik desain perangkat lunak.

Apa selanjutnya?

Pada bagian ini, sejauh ini kita telah membahas bagaimana teknik desain perangkat lunak telah berkembang sejak awal pemrograman. Saya meringkas evolusi teknik desain perangkat lunak ini dalam Gambar 1.12. Dapat diamati bahwa hampir setiap dekade yang berlalu, ide-ide revolusioner diajukan untuk merancang program yang lebih besar dan lebih canggih, dan pada saat yang sama kualitas solusi desain meningkat. Tapi, apa perbaikan teknik

desain selanjutnya? Sangat sulit untuk berspekulasi tentang perkembangan yang mungkin terjadi di masa depan. Namun, kita telah melihat bahwa di masa lalu, teknik desain telah berkembang setiap saat untuk memenuhi tantangan yang dihadapi dalam mengembangkan perangkat lunak kontemporer. Oleh karena itu, perkembangan selanjutnya kemungkinan besar akan terjadi untuk membantu memenuhi tantangan yang dihadapi oleh para perancang perangkat lunak modern.

Untuk mendapatkan indikasi teknik-teknik yang mungkin muncul, pertama-tama mari kita telaah apa saja tantangan saat ini dalam mendesain perangkat lunak. Pertama, ukuran program semakin meningkat dibandingkan dengan apa yang dikembangkan satu dekade lalu. Kedua, banyak perangkat lunak saat ini diperlukan untuk bekerja di lingkungan client-server melalui akses berbasis web browser (disebut perangkat lunak berbasis web). Pada saat yang sama, perangkat tertanam mengalami pertumbuhan yang belum pernah terjadi sebelumnya dan penerimaan pelanggan yang cepat dalam dekade terakhir. Itu ada untuk mengembangkan aplikasi untuk perangkat genggam kecil dan prosesor tertanam. Bagaimana pemrograman berorientasi aspek, desain berbasis client-server, dan teknik desain perangkat lunak tertanam telah muncul dengan cepat. Dalam dekade saat ini, orientasi layanan telah muncul sebagai arah rekayasa perangkat lunak baru-baru ini karena popularitas aplikasi berbasis web dan cloud publik.

Perkembangan Lainnya

Dapat dilihat bahwa perbaikan luar biasa pada teknik desain perangkat lunak yang lazim terjadi hampir setiap dekade. Perbaikan metodologi desain perangkat lunak selama lima dekade terakhir memang luar biasa. Selain kemajuan yang dibuat untuk teknik desain perangkat lunak, beberapa konsep dan teknik baru lainnya untuk pengembangan perangkat lunak yang efektif juga diperkenalkan. Teknik-teknik baru ini termasuk model siklus hidup, teknik spesifikasi, teknik manajemen proyek, teknik pengujian, teknik debugging, teknik jaminan kualitas, teknik pengukuran perangkat lunak, alat rekayasa perangkat lunak berbantuan komputer (CASE), dll. Perkembangan teknik ini mempercepat pertumbuhan perangkat lunak rekayasa sebagai disiplin. Kita akan membahas teknik-teknik ini di bab-bab selanjutnya.

1.5 PERUBAHAN UTAMA DALAM PRAKTIK PENGEMBANGAN PERANGKAT LUNAK

Sebelum kita membahas rincian berbagai prinsip rekayasa perangkat lunak, ada baiknya untuk memeriksa perbedaan mencolok yang akan Anda perhatikan ketika Anda mengamati gaya eksplorasi pengembangan perangkat lunak dan upaya pengembangan lain berdasarkan praktik rekayasa perangkat lunak modern. Perbedaan penting berikut antara kedua pendekatan pengembangan perangkat lunak ini akan segera dapat diamati.

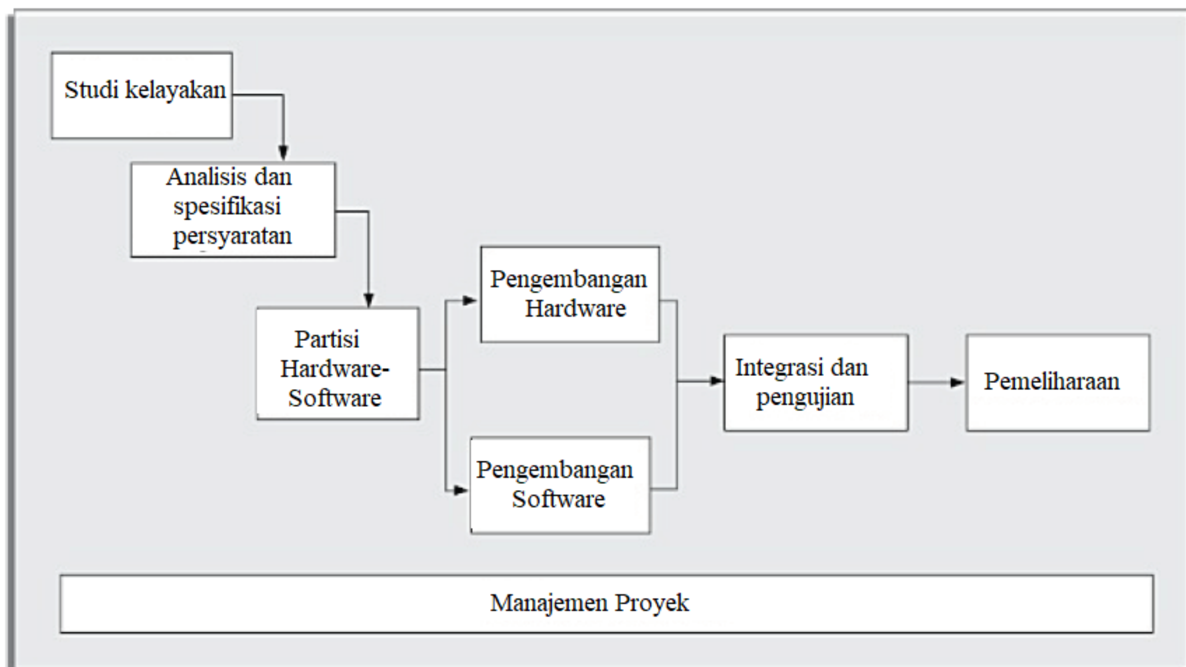
- Perbedaan penting adalah bahwa gaya pengembangan perangkat lunak eksplorasi didasarkan pada koreksi kesalahan (membangun dan memperbaiki) sedangkan teknik rekayasa perangkat lunak didasarkan pada prinsip-prinsip pencegahan kesalahan. Inheren dalam prinsip-prinsip rekayasa perangkat lunak adalah kesadaran bahwa jauh lebih hemat biaya untuk mencegah kesalahan terjadi daripada memperbaikinya ketika dan ketika kesalahan itu terdeteksi. Bahkan ketika kesalahan dilakukan selama pengembangan, prinsip-prinsip rekayasa perangkat lunak menekankan deteksi kesalahan seperti yang terdeteksi hanya selama pengujian produk akhir. Sebaliknya, praktik modern pengembangan perangkat lunak adalah mengembangkan perangkat lunak melalui beberapa tahap yang terdefinisi dengan baik seperti spesifikasi persyaratan, desain, pengkodean, pengujian, dll., Dan upaya dilakukan untuk mendeteksi dan memperbaiki kesalahan sebanyak mungkin dalam hal yang sama. fase di mana mereka dibuat.

- Dalam gaya eksplorasi, pengkodean dianggap identik dengan pengembangan perangkat lunak. Misalnya, cara pengembangan perangkat lunak yang naif ini diyakini dapat mengembangkan sistem kerja secepat mungkin dan kemudian secara berurutan memodifikasinya hingga kinerjanya memuaskan. Programmer eksplorasi benar-benar terjun ke komputer untuk memulai program mereka bahkan sebelum mereka sepenuhnya mempelajari masalahnya!!! Diakui bahwa pemrograman eksplorasi tidak hanya menjadi sangat mahal untuk masalah non-sepele, tetapi juga menghasilkan program yang sulit dipelihara. Bahkan modifikasi kecil pada program semacam itu nantinya bisa menjadi mimpi buruk. Dalam gaya pengembangan perangkat lunak modern, pengkodean dianggap hanya sebagai bagian kecil dari keseluruhan kegiatan pengembangan perangkat lunak. Ada beberapa aktivitas pengembangan seperti desain dan pengujian yang mungkin menuntut lebih banyak upaya daripada pengkodean.
- Banyak perhatian sekarang diberikan pada spesifikasi persyaratan. Upaya yang signifikan sedang dicurahkan untuk mengembangkan spesifikasi masalah yang jelas dan benar sebelum aktivitas pengembangan dimulai. Kecuali jika spesifikasi persyaratan dapat dengan benar menangkap persyaratan pelanggan yang tepat, sejumlah besar pengerjaan ulang akan diperlukan pada tahap selanjutnya. Pengerjaan ulang seperti itu akan menghasilkan biaya pengembangan yang lebih tinggi dan ketidakpuasan pelanggan.
- Sekarang ada fase desain yang berbeda di mana teknik desain standar digunakan untuk menghasilkan model desain yang koheren dan lengkap.
- Tinjauan berkala sedang dilakukan selama semua tahap proses pengembangan. Tujuan utama dari melakukan tinjauan adalah fase penahanan kesalahan, yaitu mendeteksi dan memperbaiki kesalahan sesegera mungkin. Fase penahanan kesalahan adalah prinsip rekayasa perangkat lunak yang penting. Kita akan membahas teknik ini di Bab 2.
- Saat ini, pengujian perangkat lunak telah menjadi sangat sistematis dan teknik pengujian standar tersedia. Aktivitas pengujian juga telah mencakup semua, karena kasus uji sedang dikembangkan langsung dari tahap spesifikasi persyaratan.
- Ada visibilitas yang lebih baik dari perangkat lunak melalui berbagai kegiatan pengembangan.
- Visibilitas adalah produksi dokumen yang berkualitas baik, konsisten, dan ditinjau oleh rekan sejawat di akhir setiap aktivitas pengembangan perangkat lunak.
- Di masa lalu, sangat sedikit perhatian yang diberikan untuk menghasilkan dokumen yang berkualitas baik dan konsisten. Dalam gaya eksplorasi, aktivitas desain dan pengujian, bahkan jika dilakukan (dengan cara apa pun), tidak didokumentasikan dengan memuaskan. Saat ini, dokumen berkualitas baik secara sadar sedang dikembangkan selama pengembangan perangkat lunak. Ini telah membuat diagnosis dan pemeliharaan kesalahan jauh lebih lancar. Kita akan melihat di Bab 3 bahwa selain memfasilitasi pemeliharaan produk, peningkatan visibilitas membuat pengelolaan proyek perangkat lunak menjadi lebih mudah.
- Sekarang, proyek sedang direncanakan secara menyeluruh. Tujuan utama dari perencanaan proyek adalah untuk memastikan bahwa berbagai kegiatan pembangunan berlangsung pada waktu yang tepat dan tidak ada kegiatan yang dihentikan karena kekurangan sumber daya. Perencanaan proyek biasanya mencakup persiapan berbagai jenis perkiraan, penjadwalan sumber daya, dan pengembangan rencana pelacakan proyek. Beberapa teknik dan alat otomatisasi untuk tugas-tugas seperti manajemen konfigurasi, estimasi biaya, penjadwalan, dll., digunakan untuk manajemen proyek perangkat lunak yang efektif. Beberapa metrik (pengukuran kuantitatif) produk dan

aktivitas pengembangan produk dikumpulkan untuk membantu dalam manajemen proyek perangkat lunak dan jaminan kualitas perangkat lunak.

1.6 TEKNIK SISTEM KOMPUTER

Dalam semua diskusi sejauh ini, kita berasumsi bahwa perangkat lunak yang sedang dikembangkan akan berjalan pada beberapa platform perangkat keras untuk keperluan umum seperti komputer desktop atau server. Namun, dalam beberapa situasi mungkin perlu untuk mengembangkan perangkat keras khusus yang akan menjalankan perangkat lunak tersebut. Contoh sistem semacam itu sangat banyak, dan termasuk robot, sistem otomasi pabrik, dan telepon seluler. Di dalam telepon seluler, terdapat prosesor khusus dan perangkat khusus lainnya seperti speaker dan mikrofon. Saya hanya dapat menjalankan program yang ditulis khusus untuk itu. Pengembangan sistem tersebut memerlukan pengembangan perangkat lunak dan perangkat keras khusus yang akan menjalankan perangkat lunak. Rekayasa sistem komputer membahas pengembangan sistem semacam itu yang membutuhkan pengembangan perangkat lunak dan perangkat keras khusus untuk menjalankan perangkat lunak. Dengan demikian, rekayasa sistem mencakup rekayasa perangkat lunak.



Gambar 1.13 Rekayasa sistem komputer.

Model umum rekayasa sistem ditunjukkan secara skematis pada Gambar 1.13. Salah satu tahapan penting dalam rekayasa sistem adalah tahap di mana keputusan dibuat mengenai bagian-bagian dari masalah yang akan diimplementasikan dalam perangkat keras dan yang akan diimplementasikan dalam perangkat lunak. Ini telah diwakili oleh kotak yang diberi judul partisi perangkat keras-perangkat lunak pada Gambar 1.13. Saat mempartisi fungsi antara perangkat keras dan perangkat lunak, beberapa pertukaran seperti fleksibilitas, biaya, kecepatan operasi, dll., perlu dipertimbangkan. Fungsionalitas yang diimplementasikan dalam perangkat keras berjalan lebih cepat. Di sisi lain, fungsionalitas yang diimplementasikan dalam perangkat lunak lebih mudah diperluas. Selanjutnya, sulit untuk mengimplementasikan fungsi kompleks dalam perangkat keras. Selain itu, fungsi yang diterapkan dalam perangkat keras memerlukan ruang ekstra, bobot, biaya produksi, dan overhead daya.

Setelah tahap partisi perangkat keras-perangkat lunak, pengembangan perangkat keras dan perangkat lunak dilakukan secara bersamaan (ditunjukkan sebagai cabang

bersamaan pada Gambar 1.13). Dalam rekayasa sistem, pengujian perangkat lunak selama pengembangan menjadi masalah yang rumit, perangkat keras tempat perangkat lunak akan dijalankan dan diuji masih dalam pengembangan—ingat bahwa perangkat keras dan perangkat lunak sedang dikembangkan pada waktu yang sama. Untuk menguji perangkat lunak selama pengembangan, biasanya perlu mengembangkan simulator yang meniru fitur perangkat keras yang sedang dikembangkan. Perangkat lunak diuji menggunakan simulator ini. Setelah pengembangan perangkat keras dan perangkat lunak selesai, keduanya diintegrasikan dan diuji. Aktivitas manajemen proyek diperlukan selama durasi pengembangan sistem seperti yang ditunjukkan pada Gambar 1.13. Dalam teks ini, kita telah membatasi perhatian kita pada rekayasa perangkat lunak saja.

1.7 RINGKASAN

- Dua definisi alternatif ruang lingkup rekayasa perangkat lunak:
  - o Kumpulan sistematis pengalaman pemrograman selama beberapa dekade bersama dengan inovasi yang dibuat oleh para peneliti untuk mengembangkan perangkat lunak berkualitas tinggi dengan cara yang hemat biaya.
  - o Pendekatan rekayasa untuk mengembangkan perangkat lunak.
- Gaya pengembangan program eksplorasi (juga disebut *build and fix*) digunakan oleh programmer pemula. Gaya eksplorasi dicirikan dengan mengembangkan kode program dengan cepat dan kemudian memodifikasinya secara berurutan hingga program berfungsi. Pendekatan tidak hanya menjadi cara yang sangat mahal dan tidak efisien untuk mengembangkan perangkat lunak, tetapi juga menghasilkan produk yang tidak dapat diandalkan dan sulit untuk dipelihara. Gaya eksplorasi sangat sulit digunakan ketika perangkat lunak dikembangkan melalui upaya tim. Kelemahan yang lebih besar dari gaya pemrograman eksplorasi ini bisa rusak ketika digunakan untuk mengembangkan program besar.
- Peningkatan upaya dan waktu dengan ukuran program akan menjadi eksponensial kecuali seseorang menggunakan prinsip-prinsip rekayasa perangkat lunak,—membuat hampir tidak mungkin bagi seseorang untuk mengembangkan program besar.
- Untuk menangani kompleksitas dalam suatu masalah, semua prinsip rekayasa perangkat lunak menggunakan dua teknik berikut secara ekstensif:
  - o Abstraksi (pemodelan), dan
  - o Dekomposisi (Membagi dan menaklukkan).
- Teknik rekayasa perangkat lunak sangat penting untuk pengembangan produk perangkat lunak besar di mana sekelompok insinyur bekerja dalam tim untuk mengembangkan produk. Namun, sebagian besar prinsip rekayasa perangkat lunak berguna bahkan saat mengembangkan program kecil.
- Sebuah program disebut terstruktur, ketika didekomposisi menjadi satu set modul dan setiap modul pada gilirannya didekomposisi menjadi fungsi.
- Program terstruktur menghindari penggunaan pernyataan GO TO dan hanya menggunakan konstruksi pemrograman terstruktur.
- Rekayasa sistem komputer berkaitan dengan pengembangan sistem yang lengkap, yang memerlukan pengembangan terintegrasi dari bagian perangkat lunak dan perangkat keras. Rekayasa sistem komputer mencakup rekayasa perangkat lunak.
- Mereka yang telah menulis program berukuran besar, dapat lebih menghargai banyak prinsip rekayasa perangkat lunak.

1.8 LATIHAN

1. Pilih opsi yang benar:
   a. Manakah dari berikut ini yang bukan merupakan gejala dari krisis perangkat lunak saat ini:
      i. Perangkat lunak itu mahal.
      ii. Membutuhkan waktu terlalu lama untuk membangun produk perangkat lunak.
      iii. Perangkat lunak dikirimkan terlambat.
      iv. Produk perangkat lunak diperlukan untuk melakukan tugas yang sangat kompleks.
   b. Tujuan dari pemrograman terstruktur adalah:
      i. Memiliki program yang berindentasi baik.
      ii. Dapat menyimpulkan aliran kontrol dari kode yang dikompilasi.
      iii. Mampu menyimpulkan alur kendali dari teks program.
      iv. Untuk menghindari penggunaan pernyataan GO TO.
   c. Penggunaan pernyataan GO TO yang tidak dibatasi biasanya dihindari saat menulis program, karena:
      i. Meningkatkan waktu berjalan program.
      ii. Meningkatkan kebutuhan memori program.
      iii. Ini menghasilkan ukuran kode yang dapat dieksekusi lebih besar.
      iv. ini membuat debugging menjadi sulit.
   d. Mengapa menulis kode yang mudah dimodifikasi itu penting?
      i. Kode yang mudah dimodifikasi menghasilkan loading yang lebih cepat.
      ii. Sebagian besar program dunia nyata memerlukan perubahan pada beberapa titik waktu atau lainnya.
      iii. Sebagian besar editor teks mewajibkan penulisan kode yang dapat dimodifikasi.
      iv. Beberapa orang mungkin menulis bagian kode yang berbeda secara bersamaan.
2. Apa tujuan utama dari disiplin rekayasa perangkat lunak? Apa yang dibahas oleh disiplin rekayasa perangkat lunak?
3. Menurut Anda mengapa pengembangan perangkat lunak sistematis menggunakan prinsip rekayasa perangkat lunak berbeda dari seni atau kerajinan?
4. Bedakan antara program dan perangkat lunak yang dikembangkan secara profesional.
5. Membedakan antara program, produk perangkat lunak, dan layanan perangkat lunak. Berikan satu contoh masing-masing. Diskusikan perbedaan karakteristik proyek pembangunan untuk masing-masingnya.
6. Apa yang dimaksud dengan lini produk perangkat lunak? Berikan contoh lini produk perangkat lunak. Bagaimana pengembangan lini produk perangkat lunak berbeda dari pengembangan produk perangkat lunak.
7. Apa jenis proyek utama yang dilakukan oleh perusahaan pengembangan perangkat lunak? Berikan contoh jenis proyek ini dan tunjukkan perbedaan karakteristik penting antara jenis proyek ini.
8. Apakah Anda setuju dengan pernyataan berikut: *Fokus program eksplorasi adalah koreksi kesalahan sedangkan prinsip-prinsip rekayasa perangkat lunak menekankan pencegahan kesalahan"?* Mengapa? Jelaskan alasan Anda.

9. Kesulitan apa yang akan dihadapi perusahaan pengembangan perangkat lunak, jika mencoba menggunakan gaya pengembangan program eksplorasi (membangun dan memperbaiki) dalam proyek pengembangannya? Jelaskan jawaban Anda.
10. Apa saja gejala krisis perangkat lunak saat ini? Faktor-faktor apa yang telah berkontribusi pada pembuatan krisis perangkat lunak saat ini? Apa solusi yang mungkin untuk krisis perangkat lunak saat ini?
11. Jelaskan mengapa upaya, waktu, dan biaya yang diperlukan dalam mengembangkan program menggunakan gaya *build and fix* meningkat secara eksponensial dengan ukuran program? Bagaimana prinsip-prinsip rekayasa perangkat lunak membantu mengatasi peningkatan pesat dalam waktu dan biaya pengembangan ini?
12. Apa perbedaan produk dan layanan perangkat lunak. Berikan contoh masing-masing.
13. Apa saja jenis proyek yang dilakukan oleh rumah pengembangan perangkat lunak? Manakah dari jenis proyek ini yang merupakan keahlian organisasi pengembangan perangkat lunak India? Identifikasi kemungkinan alasan mengapa yang lain tidak difokuskan oleh organisasi pengembangan perangkat lunak India.
14. Sebutkan teknik dasar yang digunakan oleh teknik rekayasa perangkat lunak untuk menangani kompleksitas dalam suatu masalah.
15. Apa yang Anda pahami dengan gaya pengembangan perangkat lunak eksplorasi (juga dikenal sebagai *build and fix)?* Secara grafis menggambarkan aktivitas yang biasanya dilakukan oleh programmer saat mengembangkan solusi pemrograman menggunakan gaya eksplorasi. Dalam diagram Anda juga tunjukkan urutan kegiatan yang dilakukan. Apa kekurangan dari gaya pengembangan program ini?
16. Buat daftar perbedaan utama antara praktik pengembangan perangkat lunak eksplorasi dan modern.
17. Apa perbedaan antara kompleksitas aktual dari pemecahan masalah dan kompleksitas yang dirasakan? Apa yang menyebabkan perbedaan antara keduanya muncul?
18. Apa yang Anda pahami dengan istilah kompleksitas masalah yang dirasakan? Apa bedanya dengan kompleksitas komputasi? Bagaimana kompleksitas masalah yang dirasakan dapat dikurangi?
19. Mengapa angka 7 dianggap sebagai angka ajaib dalam rekayasa perangkat lunak? Bagaimana rekayasa perangkat lunak yang berguna?
20. Apa yang Anda pahami dengan prinsip abstraksi dan dekomposisi? Mengapa kedua prinsip ini dianggap penting dalam rekayasa perangkat lunak? Jelaskan masalah yang ingin dipecahkan oleh kedua prinsip ini? Dukung jawaban Anda dengan menggunakan contoh yang sesuai.
21. Apa yang Anda pahami dengan struktur aliran kontrol suatu program? Mengapa sulit untuk memahami program yang memiliki struktur aliran kontrol yang berantakan? Bagaimana struktur aliran kontrol yang baik untuk suatu program dapat dirancang?
22. Apa itu diagram alur? Bagaimana teknik diagram alur berguna selama pengembangan perangkat lunak?
23. Apa yang Anda pahami dengan visibilitas desain dan kode? Bagaimana peningkatan visibilitas membantu dalam pengembangan perangkat lunak yang sistematis? (Kita akan meninjau kembali pertanyaan ini di Bab 3)
24. Apa yang Anda pahami dengan istilah—pemrograman terstruktur? Bagaimana bahasa pemrograman modern seperti PASCAL dan C memfasilitasi penulisan program terstruktur? Apa keuntungan menulis program terstruktur dibandingkan dengan program tidak terstruktur?

25. Apa itu bahasa pemrograman tingkat tinggi? Mengapa seorang programmer yang menggunakan bahasa pemrograman tingkat tinggi memiliki produktivitas yang lebih tinggi dibandingkan dengan ketika menggunakan bahasa mesin untuk pengembangan aplikasi?
26. Apa tiga tipe dasar konstruksi program yang diperlukan untuk mengembangkan program untuk setiap masalah yang diberikan? Berikan contoh ketiga konstruksi ini dari bahasa tingkat tinggi apa pun yang Anda ketahui.
27. Apa yang Anda pahami dengan modul program? Apa karakteristik penting dari modul program?
28. Jelaskan bagaimana penggunaan prinsip-prinsip rekayasa perangkat lunak membantu mengembangkan produk perangkat lunak dengan biaya yang efektif dan tepat waktu. Uraikan jawaban Anda dengan menggunakan contoh-contoh yang sesuai.
29. Apa perbedaan mendasar antara teknik desain berorientasi aliran kontrol dan aliran data? Dapatkah Anda memikirkan alasan mengapa teknik desain berorientasi aliran data cenderung menghasilkan desain yang lebih baik daripada teknik desain berorientasi aliran kontrol?.
30. Sebutkan dua prinsip dasar yang digunakan secara luas dalam rekayasa perangkat lunak untuk mengatasi kompleksitas dalam mengembangkan program besar? Jelaskan kedua prinsip tersebut. Dengan menggunakan contoh yang sesuai, jelaskan bagaimana kedua prinsip ini membantu mengatasi kompleksitas yang terkait dengan pengembangan program besar.
31. Apa yang diwakili oleh grafik aliran kontrol (CFG) dari suatu program? Gambarkan CFG dari program berikut:

```
main(){
   int y=1;
      if(y<0)
         if(y>0) y=3;
         else y=0;
   printf("%d\n",y);
}
```

32. Diskusikan kemungkinan alasan di balik penggantian metode desain berorientasi struktur data dengan metode desain berorientasi aliran kontrol.
33. Apa yang dimaksud dengan metodologi desain perangkat lunak berorientasi struktur data? Apa bedanya dengan metodologi desain berorientasi aliran data?
34. Diskusikan keuntungan utama dari metodologi desain berorientasi objek (OOD) atas metodologi desain berorientasi aliran data.
35. Jelaskan bagaimana teknik desain perangkat lunak telah berkembang di masa lalu. Bagaimana menurut Anda teknik desain perangkat lunak akan berkembang dalam waktu dekat?
36. Apa itu rekayasa sistem komputer? Apa bedanya dengan rekayasa perangkat lunak? Berikan contoh beberapa jenis proyek pengembangan produk yang rekayasa sistemnya sesuai.
37. Apakah yang Anda maksud: layanan perangkat lunak Jelaskan perbedaan penting antara karakteristik proyek pengembangan layanan perangkat lunak dan proyek pengembangan produk perangkat lunak.

BAB 2
MODEL SIKLUS HIDUP PERANGKAT LUNAK

Dalam Bab 1, kita membahas beberapa isu dasar dalam rekayasa perangkat lunak. Beberapa perbedaan penting antara gaya pengembangan program eksplorasi dan pendekatan rekayasa perangkat lunak akan ditunjukkan. Harap ingat kembali dari diskusi di Bab 1 bahwa gaya eksplorasi juga dikenal sebagai pemrograman build and fix. Dalam membangun dan memperbaiki pemrograman, seorang programmer biasanya mulai menulis program segera setelah ia membentuk pemahaman informal tentang persyaratan. Setelah penulisan program selesai, ia turun untuk memperbaiki apa pun yang tidak memenuhi harapan pengguna. Biasanya, sejumlah besar perbaikan kode diperlukan bahkan untuk program mainan. Hal ini mendorong naiknya biaya pengembangan dan menurunkan kualitas program. Selanjutnya, pendekatan ini biasanya menjadi resep untuk kegagalan proyek ketika digunakan untuk mengembangkan program non-sepele yang membutuhkan usaha tim. Berbeda dengan gaya membangun dan memperbaiki, pendekatan rekayasa perangkat lunak menekankan pengembangan perangkat lunak melalui serangkaian aktivitas yang terdefinisi dengan baik dan teratur. Kegiatan-kegiatan ini dimodelkan secara grafis (diwakili) serta dijelaskan secara tekstual dan secara beragam disebut model siklus hidup perangkat lunak, model siklus hidup pengembangan perangkat lunak (SDLC), dan model proses pengembangan perangkat lunak. Beberapa model siklus hidup sejauh ini telah diusulkan. Namun, dalam Bab ini kita membatasi perhatian kita pada beberapa yang penting dan umum digunakan.

Dalam bab ini, pertama-tama kita membahas beberapa konsep dasar yang terkait dengan model siklus hidup. Selanjutnya, kegiatan penting yang telah ditentukan untuk dilakukan dalam model air terjun klasik. Hal ini dimaksudkan untuk memberikan wawasan tentang kegiatan yang dilakukan sebagai bagian dari setiap model siklus hidup. Bahkan, model air terjun klasik dapat dianggap sebagai model dasar dan semua model siklus hidup lainnya sebagai perpanjangan dari model ini untuk memenuhi situasi proyek tertentu. Setelah membahas model waterfall, kita bahas beberapa turunan dari model ini. Selanjutnya kita membahas model spiral yang menggeneralisasi berbagai model siklus hidup. Terakhir, beberapa model siklus hidup yang baru-baru ini diusulkan yang dikategorikan di bawah payung model agile. Akhir-akhir ini, model tangkas menemukan penerimaan yang meningkat di antara developer dan peneliti. Asal-usul model tangkas dapat ditelusuri ke perubahan radikal pada jenis proyek yang sedang dilakukan saat ini, daripada inovasi radikal ke model siklus hidup itu sendiri. Proyek telah berubah dari proyek pengembangan produk multi-tahun yang besar menjadi proyek layanan kecil sekarang.

2.1 BEBERAPA KONSEP DASAR

Pada bagian ini, kita akan mempelajari beberapa konsep dasar tentang model siklus hidup.

Siklus hidup perangkat lunak

Diketahui bahwa semua organisme hidup mengalami siklus hidup. Misalnya, ketika benih ditanam, ia berkecambah, tumbuh menjadi pohon penuh, dan akhirnya mati. Berdasarkan konsep siklus hidup biologis, istilah siklus hidup perangkat lunak telah didefinisikan untuk menyiratkan tahapan yang berbeda (atau fase) di mana perangkat lunak berkembang dari permintaan pelanggan awal untuk itu, ke perangkat lunak yang dikembangkan sepenuhnya, dan akhirnya ke tahap di mana itu tidak lagi berguna bagi pengguna mana pun, dan kemudian dibuang.

Siklus hidup setiap perangkat lunak dimulai dengan permintaan oleh satu atau lebih pelanggan. Pada tahap ini, pelanggan biasanya tidak jelas tentang semua fitur yang akan dibutuhkan, mereka juga tidak dapat sepenuhnya menggambarkan fitur yang diidentifikasi secara konkret, dan hanya dapat secara samar menggambarkan apa yang dibutuhkan. Tahap ini di mana pelanggan merasakan kebutuhan akan perangkat lunak dan membentuk gagasan kasar tentang fitur yang diperlukan dikenal sebagai tahap awal. Dimulai dengan tahap awal, perangkat lunak berkembang melalui serangkaian tahap yang dapat diidentifikasi (juga disebut fase) karena aktivitas pengembangan yang dilakukan oleh pengembang, hingga sepenuhnya dikembangkan dan dirilis ke pelanggan.

Setelah diinstal dan tersedia untuk digunakan, pengguna mulai menggunakan perangkat lunak. Ini menandakan dimulainya fase operasi (juga disebut pemeliharaan). Saat pengguna menggunakan perangkat lunak, mereka tidak hanya meminta untuk memperbaiki kegagalan yang mungkin mereka temui, tetapi mereka juga terus menyarankan beberapa perbaikan dan modifikasi pada perangkat lunak. Dengan demikian, fase pemeliharaan biasanya melibatkan terus-menerus membuat perubahan pada perangkat lunak untuk mengakomodasi perbaikan bug dan permintaan perubahan dari pengguna. Fase operasi biasanya yang terpanjang dari semua fase dan merupakan masa manfaat perangkat lunak. Akhirnya perangkat lunak dihentikan, ketika pengguna tidak merasa berguna lagi karena alasan seperti skenario bisnis yang berubah, ketersediaan perangkat lunak baru yang memiliki fitur dan kerja yang lebih baik, platform komputasi yang berubah, dll. Ini membentuk inti dari siklus hidup dari setiap perangkat lunak. Berdasarkan uraian tersebut, kita dapat mendefinisikan siklus hidup perangkat lunak sebagai berikut:

Siklus hidup perangkat lunak mewakili serangkaian tahapan yang dapat diidentifikasi di mana ia berkembang selama masa hidupnya. Dengan pengetahuan tentang siklus hidup perangkat lunak ini, konsep model siklus hidup perangkat lunak dan mengeksplorasi mengapa perlu mengikuti model siklus hidup dalam lingkungan pengembangan perangkat lunak profesional.

Model siklus hidup pengembangan perangkat lunak (SDLC)

Dalam skenario pengembangan perangkat lunak yang sistematis, aktivitas tertentu yang terdefinisi dengan baik perlu dilakukan oleh tim pengembangan dan mungkin juga oleh pelanggan, agar perangkat lunak berkembang dari satu tahap dalam siklus hidupnya ke tahap berikutnya. Misalnya, agar perangkat lunak berkembang dari tahap spesifikasi persyaratan ke tahap desain, developer perlu memperoleh persyaratan dari pelanggan, menganalisis persyaratan tersebut, dan secara formal mendokumentasikan persyaratan dalam bentuk dokumen SRS.

Model siklus hidup pengembangan perangkat lunak (SDLC) (juga disebut model siklus hidup perangkat lunak dan model proses pengembangan perangkat lunak) menjelaskan berbagai aktivitas yang perlu dilakukan agar perangkat lunak berkembang dalam siklus hidupnya. Sepanjang diskusi kita, kita akan menggunakan istilah siklus hidup pengembangan perangkat lunak (SDLC) dan proses pengembangan perangkat lunak secara bergantian. Namun, beberapa penulis membedakan SDLC dari proses pengembangan perangkat lunak. Dalam penggunaannya, proses pengembangan perangkat lunak menggambarkan aktivitas siklus hidup lebih tepat dan rumit, dibandingkan dengan SDLC. Juga, proses pengembangan mungkin tidak hanya menggambarkan berbagai kegiatan yang dilakukan selama siklus hidup, tetapi juga menentukan metodologi khusus untuk melaksanakan kegiatan, dan juga merekomendasikan dokumen spesifik dan artefak lain yang harus dihasilkan pada akhir setiap fase. Dalam pengertian ini, istilah SDLC dapat dianggap sebagai istilah yang lebih umum,

dibandingkan dengan proses pengembangan dan beberapa proses pengembangan mungkin cocok dengan SDLC yang sama.

SDLC direpresentasikan secara grafis dengan menggambar berbagai tahapan siklus hidup dan menunjukkan transisi antar fase. Model grafis ini biasanya disertai dengan deskripsi tekstual dari berbagai kegiatan yang perlu dilakukan selama fase sebelum fase itu dapat dianggap selesai. Dengan kata sederhana, kita dapat mendefinisikan SDLC sebagai berikut: SDLC secara grafis menggambarkan fase-fase yang berbeda di mana perangkat lunak berkembang. Biasanya disertai dengan deskripsi tekstual tentang berbagai kegiatan yang perlu dilakukan selama setiap fase.

Proses versus metodologi

Meskipun istilah proses dan metodologi kadang-kadang digunakan secara bergantian, ada perbedaan tipis antara keduanya. Pertama, istilah proses memiliki cakupan yang lebih luas dan membahas semua aktivitas yang terjadi selama pengembangan perangkat lunak, atau aktivitas tertentu yang kasar seperti desain (misalnya proses desain), pengujian (proses pengujian), dll. Selanjutnya, proses perangkat lunak tidak hanya mengidentifikasi aktivitas spesifik yang perlu dilakukan, tetapi juga dapat menentukan metodologi tertentu untuk melaksanakan setiap aktivitas. Misalnya, proses desain dapat merekomendasikan bahwa pada tahap desain, aktivitas desain tingkat tinggi dilakukan menggunakan analisis terstruktur dan metodologi desain Hatley dan Pirbhai.

Metodologi, di sisi lain, mengatur serangkaian langkah untuk melakukan aktivitas siklus hidup tertentu. Ini juga dapat mencakup alasan dan asumsi filosofis di balik serangkaian langkah-langkah di mana aktivitas tersebut dicapai. Proses pengembangan perangkat lunak memiliki cakupan yang jauh lebih luas dibandingkan dengan metodologi pengembangan perangkat lunak. Sebuah proses biasanya menggambarkan semua aktivitas mulai dari awal perangkat lunak hingga tahap pemeliharaan dan penghentiannya, atau setidaknya sebagian aktivitas dalam siklus hidup. Ini juga merekomendasikan metodologi khusus untuk melaksanakan setiap kegiatan. Metodologi, sebaliknya, menggambarkan langkah-langkah untuk melakukan hanya satu atau paling banyak beberapa kegiatan.

Mengapa menggunakan proses pengembangan?

Keuntungan utama menggunakan proses pengembangan adalah mendorong pengembangan perangkat lunak secara sistematis dan disiplin. Mengikuti suatu proses sangat penting untuk pengembangan perangkat lunak profesional yang membutuhkan upaya tim. Ketika perangkat lunak dikembangkan oleh tim daripada oleh programmer individu, penggunaan model siklus hidup menjadi sangat diperlukan untuk keberhasilan penyelesaian proyek. Organisasi pengembangan perangkat lunak telah menyadari bahwa kepatuhan terhadap model siklus hidup yang sesuai membantu menghasilkan perangkat lunak berkualitas baik dan membantu meminimalkan kemungkinan pembengkakan waktu dan biaya.

Misalkan seorang programmer tunggal sedang mengembangkan sebuah program kecil. Misalnya, seorang mahasiswa mungkin mengembangkan kode untuk tugas ruang kelas. mahasiswa mungkin berhasil bahkan ketika dia tidak secara ketat mengikuti proses pengembangan tertentu dan mengadopsi gaya pengembangan dan perbaikan. Namun, ini adalah permainan bola yang berbeda ketika perangkat lunak profesional sedang dikembangkan oleh tim pemrogram. Sekarang mari kita memahami kesulitan yang mungkin timbul jika tim tidak menggunakan proses pengembangan apa pun, dan anggota tim diberikan kebebasan penuh untuk mengembangkan bagian perangkat lunak yang ditugaskan sesuai dengan kebijaksanaan mereka sendiri.

Beberapa jenis masalah mungkin muncul. Salah satu masalah digambarkan melalui contoh. Misalkan, masalah pengembangan perangkat lunak telah dibagi menjadi beberapa bagian dan bagian-bagian ini ditugaskan ke anggota tim. Sejak saat itu, anggaplah anggota tim diberi kebebasan untuk mengembangkan bagian-bagian yang ditugaskan kepada mereka dengan cara apa pun yang mereka suka. Ada kemungkinan bahwa satu anggota mungkin mulai menulis kode untuk bagiannya sambil membuat asumsi tentang hasil input yang diperlukan dari bagian lain, yang lain mungkin memutuskan untuk menyiapkan dokumen uji terlebih dahulu, dan beberapa developer lain mungkin mulai melaksanakan desain untuk bagian tersebut. bagian yang diberikan kepadanya. Dalam hal ini, masalah berat dapat muncul dalam menghubungkan bagian-bagian yang berbeda dan dalam mengelola pembangunan secara keseluruhan. Oleh karena itu, pengembangan ad hoc ternyata merupakan cara yang pasti untuk mendapatkan proyek yang gagal. Percaya atau tidak, inilah yang menyebabkan banyak proyek gagal di masa lalu!

Ketika sebuah perangkat lunak dikembangkan oleh sebuah tim, penting untuk memiliki pemahaman yang tepat di antara anggota tim tentang—kapan melakukan apa. Dengan tidak adanya pemahaman seperti itu, jika setiap anggota setiap saat akan melakukan aktivitas apa pun yang dia ingin lakukan. Ini akan menjadi undangan terbuka untuk kekacauan perkembangan dan kegagalan proyek. Penggunaan model siklus hidup yang sesuai sangat penting untuk keberhasilan penyelesaian proyek pengembangan berbasis tim. Namun, apakah kita memerlukan model SDLC untuk mengembangkan program kecil. Dalam konteks ini, kita perlu membedakan antara programming-in-the-small dan programming-in-the-large. Pemrograman-dalam-kecil mengacu pada pengembangan program mainan oleh seorang programmer tunggal. Sedangkan programming-in-the-large mengacu pada pengembangan perangkat lunak profesional melalui upaya tim. Sementara pengembangan perangkat lunak dari tipe sebelumnya dapat berhasil bahkan ketika seorang programmer individu menggunakan gaya pengembangan dan perbaikan, penggunaan SDLC yang sesuai sangat penting untuk proyek pengembangan perangkat lunak profesional yang melibatkan upaya tim untuk berhasil.

Mengapa mendokumentasikan proses pengembangan?

Tidaklah cukup bagi sebuah organisasi untuk hanya memiliki proses pengembangan yang terdefinisi dengan baik, tetapi proses pengembangan perlu didokumentasikan dengan baik. Untuk memahami alasannya, mari kita pertimbangkan bahwa organisasi pengembangan tidak mendokumentasikan proses pengembangannya. Dalam hal ini, pengembangnya hanya mengembangkan pemahaman informal tentang proses pengembangan. Pemahaman informal tentang proses pengembangan di antara anggota tim dapat menimbulkan beberapa masalah selama pengembangan. Ientifikasi beberapa masalah penting yang mungkin muncul ketika proses pengembangan tidak didokumentasikan secara memadai. Masalah-masalah tersebut adalah sebagai berikut:

- Model proses terdokumentasi memastikan bahwa setiap aktivitas dalam siklus hidup didefinisikan secara akurat. Juga, di mana diperlukan metodologi untuk melaksanakan kegiatan masing-masing dijelaskan. Tanpa dokumentasi, aktivitas dan urutannya cenderung didefinisikan secara longgar, yang menyebabkan kebingungan dan salah tafsir oleh tim yang berbeda dalam organisasi. Misalnya, tinjauan kode dapat dilakukan secara informal dan tidak memadai karena tidak ada metodologi yang terdokumentasi tentang bagaimana tinjauan kode harus dilakukan. Kesulitan lain adalah bahwa untuk aktivitas yang didefinisikan secara longgar, developer cenderung menggunakan penilaian subjektif mereka. Sebagai contoh, kecuali jika ditentukan secara eksplisit, anggota tim akan secara subyektif memutuskan apakah kasus uji harus dirancang tepat setelah fase persyaratan,

setelah fase desain, atau setelah fase pengkodean. Juga, mereka akan memperdebatkan apakah kasus uji harus didokumentasikan sama sekali dan ketelitiannya harus didokumentasikan.

- Proses yang tidak terdokumentasi memberikan indikasi yang jelas kepada anggota tim pengembangan tentang kurangnya keseriusan pihak manajemen organisasi untuk mengikuti proses tersebut. Oleh karena itu, proses yang tidak berdokumen berfungsi sebagai petunjuk bagi developer untuk mengikuti proses secara longgar. Gejala dari proses yang tidak terdokumentasi mudah terlihat—desain dilakukan dengan lusuh, tinjauan tidak dilakukan secara ketat, dll.
- Sebuah tim proyek mungkin sering harus menyesuaikan model proses standar untuk digunakan dalam proyek tertentu. Lebih mudah untuk menyesuaikan model proses yang terdokumentasi, ketika diperlukan untuk memodifikasi aktivitas atau fase tertentu dari siklus hidup. Misalnya, pertimbangkan situasi proyek yang mengharuskan aktivitas pengujian dialihdayakan ke organisasi lain. Pada kasus ini,
- Model proses yang terdokumentasi akan membantu mengidentifikasi di mana tepatnya penyesuaian yang diperlukan harus dilakukan. Model proses terdokumentasi merupakan persyaratan wajib standar jaminan kualitas modern seperti ISO 9000 dan SEI CMM. Ini berarti bahwa kecuali organisasi perangkat lunak memiliki proses terdokumentasi, itu tidak akan memenuhi syarat untuk akreditasi dengan standar kualitas apa pun. Dengan tidak adanya sertifikasi mutu untuk organisasi, pelanggan akan curiga terhadap kemampuannya dalam mengembangkan perangkat lunak berkualitas dan organisasi mungkin akan kesulitan memenangkan tender untuk pengembangan perangkat lunak.

Proses pengembangan yang terdokumentasi membentuk pemahaman umum tentang kegiatan yang akan dilakukan di antara para developer perangkat lunak dan membantu mereka mengembangkan perangkat lunak secara sistematis dan disiplin. Model proses pengembangan yang terdokumentasi, selain mencegah salah tafsir yang mungkin terjadi ketika proses pengembangan tidak didokumentasikan secara memadai, juga membantu mengidentifikasi inkonsistensi, redundansi, dan kelalaian dalam proses pengembangan. Saat ini, organisasi pengembangan perangkat lunak yang baik biasanya mendokumentasikan proses pengembangan mereka dalam bentuk buklet. Mereka mengharapkan para developer yang baru direkrut untuk organisasi mereka untuk terlebih dahulu menguasai proses pengembangan perangkat lunak mereka selama pelatihan induksi singkat yang harus mereka jalani.

Kriteria masuk dan keluar fase

SDLC yang baik selain secara jelas mengidentifikasi fase-fase yang berbeda dalam siklus hidup, harus secara jelas mendefinisikan kriteria masuk dan keluar untuk setiap fase. Kriteria masuk (atau keluar) fase biasanya dinyatakan sebagai serangkaian kondisi yang harus dipenuhi agar fase dapat dimulai (atau diselesaikan). Sebagai contoh, kriteria keluar fase untuk fase spesifikasi kebutuhan perangkat lunak, dapat berupa dokumen spesifikasi persyaratan perangkat lunak (SRS) sudah siap, telah ditinjau secara internal, dan juga telah ditinjau dan disetujui oleh pelanggan. Hanya setelah kriteria ini terpenuhi, fase berikutnya dapat dimulai.

Jika kriteria masuk dan keluar untuk berbagai fase tidak didefinisikan dengan baik, maka itu akan meninggalkan ruang yang cukup untuk ambiguitas dalam memulai dan mengakhiri berbagai fase, dan menyebabkan banyak kebingungan di antara para pengembang. Kadang-kadang mereka mungkin menghentikan aktivitas dalam suatu fase sebelum waktunya, dan di lain waktu mereka mungkin terus mengerjakan suatu fase jauh setelah fase itu seharusnya berakhir. Keputusan mengenai apakah suatu fase selesai atau tidak menjadi subyektif dan menjadi sulit bagi manajer proyek untuk secara akurat

mengatakan berapa banyak perkembangan telah berkembang. Ketika kriteria masuk dan keluar fase tidak didefinisikan dengan baik, developer mungkin menutup aktivitas fase jauh sebelum mereka benar-benar selesai, memberikan kesan palsu tentang kemajuan pesat. Dalam hal ini, menjadi sangat sulit bagi manajer proyek untuk menentukan status pasti pengembangan dan melacak kemajuan proyek. Ini biasanya mengarah pada masalah yang biasanya diidentifikasi sebagai sindrom lengkap 99 persen. Sindrom ini muncul ketika manajer proyek perangkat lunak tidak memiliki cara yang pasti untuk menilai kemajuan suatu proyek, anggota tim yang optimis merasa bahwa pekerjaan mereka 99 persen selesai bahkan ketika pekerjaan mereka jauh dari penyelesaian—membuat semua proyeksi yang dibuat oleh proyek. manajer tentang waktu penyelesaian proyek menjadi sangat tidak akurat.

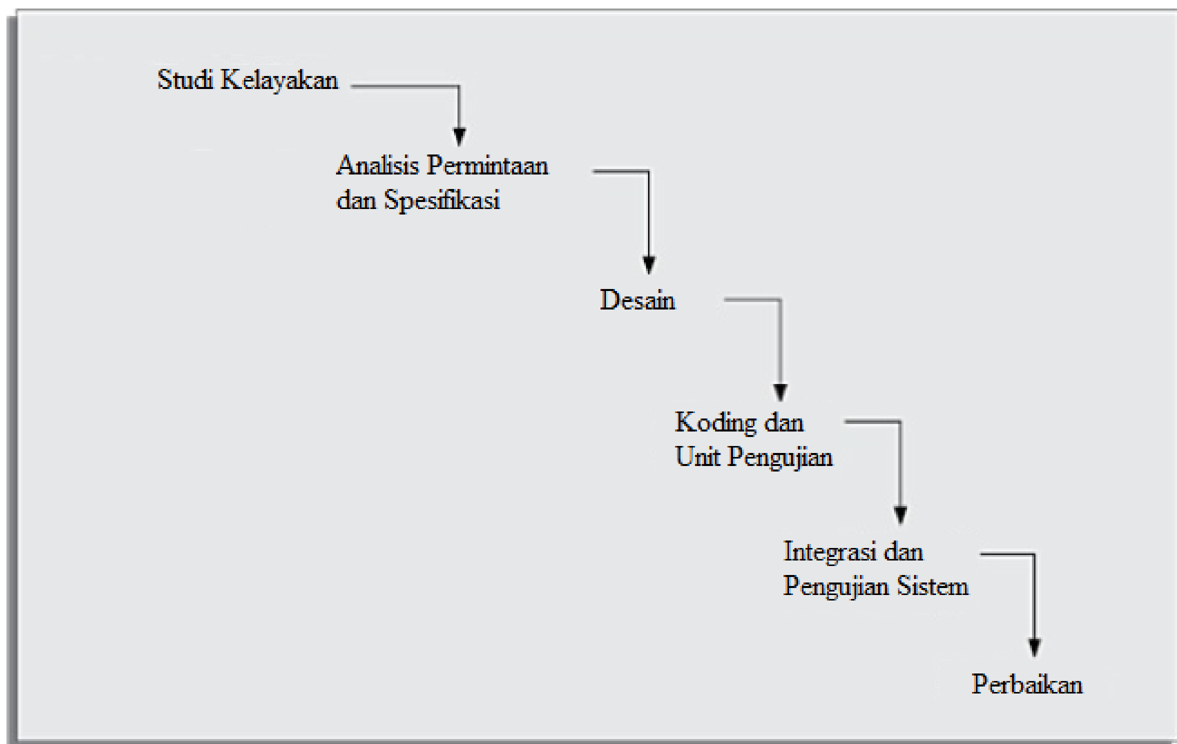
2.2 MODEL AIR TERJUN DAN PERLUASANNYA

Model air terjun dan turunannya sangat populer di tahun 1970-an dan masih banyak digunakan di banyak proyek pembangunan. Model air terjun mungkin merupakan cara yang paling jelas dan intuitif di mana perangkat lunak dapat dikembangkan melalui upaya tim. Kita dapat menganggap model air terjun sebagai model generik yang telah diperluas dalam banyak cara untuk memenuhi situasi pengembangan perangkat lunak tertentu tertentu untuk mewujudkan semua model siklus hidup perangkat lunak lainnya.

Model Air Terjun Klasik

Model air terjun klasik secara intuitif merupakan cara paling jelas untuk mengembangkan perangkat lunak. Sederhana tapi idealis. Faktanya, sulit untuk menerapkan model ini dalam proyek pengembangan perangkat lunak non-sepele. Orang mungkin bertanya-tanya apakah model ini sulit digunakan dalam proyek pengembangan praktis, lalu mengapa mempelajarinya sama sekali? Alasannya adalah bahwa semua model siklus hidup lainnya dapat dianggap sebagai perpanjangan dari model air terjun klasik.

Oleh karena itu, masuk akal untuk terlebih dahulu memahami model air terjun klasik, agar dapat mengembangkan pemahaman yang tepat tentang model siklus hidup lainnya. Selain itu, kita akan melihat nanti dalam teks ini bahwa model ini meskipun tidak digunakan untuk pengembangan perangkat lunak; secara implisit digunakan saat mendokumentasikan perangkat lunak. Model air terjun klasik membagi siklus hidup menjadi satu set fase seperti yang ditunjukkan pada Gambar 2.1. Dapat dengan mudah diamati dari gambar ini bahwa representasi diagram model air terjun klasik menyerupai air terjun multi-level. Kemiripan ini membenarkan nama model.

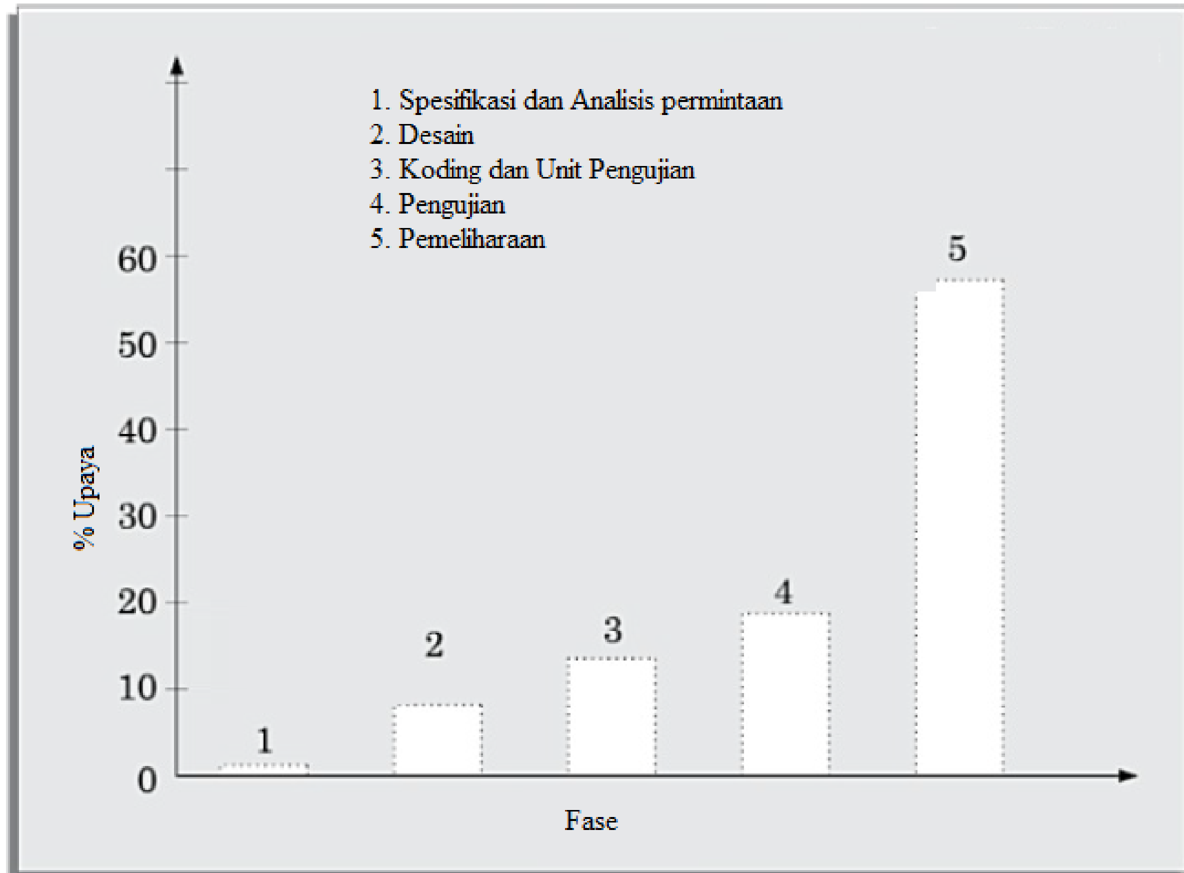


Gambar 2.1 Model air terjun klasik.

Fase model air terjun klasik

Fase yang berbeda dari model air terjun klasik telah ditunjukkan pada Gambar 2.1. Seperti yang ditunjukkan pada Gambar 2.1, fase yang berbeda adalah—studi kelayakan, analisis dan spesifikasi kebutuhan, desain, pengkodean dan pengujian unit, integrasi dan pengujian sistem, dan pemeliharaan. Fase mulai dari studi kelayakan hingga integrasi dan fase pengujian sistem dikenal sebagai fase pengembangan. Perangkat lunak dikembangkan selama fase pengembangan, dan pada penyelesaian fase pengembangan, perangkat lunak dikirimkan ke pelanggan. Setelah pengiriman perangkat lunak, pelanggan mulai menggunakan perangkat lunak yang menandakan dimulainya fase operasi. Ketika pelanggan mulai menggunakan perangkat lunak, perubahan menjadi perlu karena perbaikan bug dan ekstensi fitur, menyebabkan pekerjaan pemeliharaan harus dilakukan. Oleh karena itu, fase terakhir juga dikenal sebagai fase pemeliharaan siklus hidup.

Perlu diingat bahwa beberapa buku teks memiliki nomor dan nama fase yang berbeda. Aktivitas yang mencakup semua fase pengembangan perangkat lunak adalah manajemen proyek. Karena mencakup seluruh durasi proyek, tidak ada fase khusus yang dinamai menurut namanya. Manajemen proyek, bagaimanapun, merupakan kegiatan penting dalam siklus hidup dan berhubungan dengan pengelolaan pengembangan perangkat lunak dan kegiatan pemeliharaan. Dalam model air terjun, fase siklus hidup yang berbeda biasanya memerlukan jumlah upaya yang relatif berbeda untuk dilakukan oleh tim pengembangan. Jumlah relatif dari upaya yang dihabiskan pada fase yang berbeda untuk perangkat lunak yang khas telah ditunjukkan pada Gambar 2.2. Perhatikan dari Gambar 2.2 bahwa di antara semua fase siklus hidup, fase pemeliharaan biasanya membutuhkan upaya maksimal. Rata-rata, sekitar 60 persen dari total upaya yang dilakukan oleh tim pengembangan di seluruh siklus hidup dihabiskan untuk kegiatan pemeliharaan saja.



Gambar 2.2 Distribusi usaha relatif di antara berbagai fase produk tipikal.

Namun, di antara fase pengembangan, fase integrasi dan pengujian sistem memerlukan upaya maksimal dalam proyek pengembangan yang khas. Bagian selanjutnya adalah kegiatan yang dilakukan dalam fase yang berbeda dari model air terjun klasik.

Studi kelayakan

Fokus utama dari tahap studi kelayakan adalah untuk menentukan apakah akan layak secara finansial dan teknis untuk mengembangkan perangkat lunak. Studi kelayakan melibatkan pelaksanaan beberapa kegiatan seperti pengumpulan informasi dasar yang berkaitan dengan perangkat lunak seperti item data yang berbeda yang akan dimasukkan ke sistem, pemrosesan yang harus dilakukan pada data ini, data keluaran yang diperlukan untuk diproduksi. oleh sistem, serta berbagai kendala dalam pengembangannya. Data yang dikumpulkan ini dianalisis untuk melakukan hal-hal berikut:

Pengembangan pemahaman masalah secara keseluruhan: Pertama-tama perlu dikembangkan pemahaman menyeluruh tentang apa yang dibutuhkan pelanggan untuk dikembangkan. Untuk ini, hanya persyaratan penting pelanggan yang perlu dipahami dan rincian berbagai persyaratan seperti tata letak layar yang diperlukan dalam antarmuka pengguna grafis (GUI), formula atau algoritma khusus yang diperlukan untuk menghasilkan hasil yang diperlukan, dan basis data skema yang akan digunakan diabaikan.

Perumusan berbagai kemungkinan strategi untuk memecahkan masalah: Dalam kegiatan ini, berbagai kemungkinan skema solusi tingkat tinggi untuk masalah tersebut ditentukan. Misalnya, solusi dalam kerangka klien-server dan kerangka aplikasi mandiri dapat dieksplorasi.

Evaluasi strategi solusi yang berbeda: Skema solusi yang diidentifikasi berbeda dianalisis untuk mengevaluasi manfaat dan kekurangannya. Evaluasi semacam itu seringkali

memerlukan perkiraan perkiraan sumber daya yang dibutuhkan, biaya pengembangan, dan waktu pengembangan yang diperlukan. Solusi yang berbeda dibandingkan berdasarkan estimasi yang telah dikerjakan. Setelah solusi terbaik diidentifikasi, semua kegiatan di fase selanjutnya dilakukan sesuai solusi ini. Pada tahap ini, juga dapat ditentukan bahwa tidak ada solusi yang layak karena biaya tinggi, kendala sumber daya, atau beberapa alasan teknis. Skenario ini tentu saja mengharuskan proyek tersebut ditinggalkan.

Kita dapat meringkas hasil dari fase studi kelayakan dengan mencatat bahwa selain memutuskan apakah akan mengambil sebuah proyek atau tidak, pada tahap ini keputusan tingkat yang sangat tinggi mengenai strategi solusi didefinisikan. Oleh karena itu, studi kelayakan merupakan tahap yang sangat penting dalam pengembangan perangkat lunak. Berikut ini adalah studi kasus studi kelayakan yang dilakukan oleh suatu organisasi. Hal ini dimaksudkan untuk memberikan nuansa kegiatan dan isu-isu yang terlibat dalam tahap studi kelayakan proyek perangkat lunak yang khas.

Studi Kasus 2.1

Sebuah perusahaan pertambangan bernama Galaxy Mining Company Ltd. (GMC Ltd.) memiliki tambang yang berlokasi di berbagai tempat di India. Ini memiliki sekitar lima puluh lokasi tambang berbeda yang tersebar di delapan negara bagian. Perusahaan mempekerjakan sejumlah besar penambang di setiap lokasi tambang. Pertambangan sebagai profesi berisiko, perusahaan bermaksud untuk mengoperasikan dana cadangan khusus, yang akan ada di samping dana cadangan standar yang sudah dinikmati para penambang. Tujuan utama dari adanya dana khusus (SPF) adalah untuk segera mendistribusikan sejumlah kompensasi sebelum jumlah PF dibayarkan.

Menurut skema ini, setiap lokasi tambang akan memotong angsuran SPF dari setiap penambang setiap bulan dan menyetorkannya ke komisioner dana khusus pusat (CSPFC). CSPFC akan menyimpan semua detail mengenai cicilan SPF yang dikumpulkan dari para penambang. GMC Ltd. meminta vendor perangkat lunak terkenal Adventure Software Inc. untuk melakukan tugas mengembangkan perangkat lunak untuk mengotomatisasi pemeliharaan catatan SPF semua karyawan. GMC Ltd. telah menyadari bahwa selain menghemat tenaga kerja pada pekerjaan pembukuan, perangkat lunak ini akan membantu penyelesaian kasus klaim dengan cepat. GMC Ltd. mengindikasikan bahwa jumlah yang paling bisa ia beli adalah Rp. 1 juta untuk software ini untuk dikembangkan dan diinstal.

Adventure Software Inc. menugaskan manajer proyek mereka untuk melakukan studi kelayakan. Manajer proyek berdiskusi dengan manajer puncak GMC Ltd. untuk mendapatkan gambaran umum proyek. Dia juga berdiskusi dengan petugas PF lapangan di berbagai lokasi tambang untuk menentukan detail pasti dari proyek tersebut. Manajer proyek mengidentifikasi dua pendekatan luas untuk memecahkan masalah. Salah satunya adalah memiliki database pusat yang akan diakses dan diperbarui melalui koneksi satelit ke berbagai lokasi tambang. Pendekatan lainnya adalah memiliki database lokal di setiap lokasi tambang dan memperbarui database pusat secara berkala melalui koneksi dial-up. Pembaruan berkala ini dapat dilakukan setiap hari atau setiap jam tergantung pada penundaan yang dapat diterima oleh GMC Ltd. dalam menjalankan berbagai fungsi perangkat lunak. Dia menemukan bahwa pendekatan kedua sangat terjangkau dan lebih toleran terhadap kesalahan karena lokasi tambang lokal dapat beroperasi bahkan ketika hubungan komunikasi untuk sementara gagal. Dalam pendekatan ini, ketika tautan gagal, hanya pembaruan basis data pusat yang tertunda. Sedangkan pada pendekatan pertama, semua pekerjaan SPF terhenti di situs tambang selama seluruh durasi kegagalan tautan. Manajer proyek dengan cepat menganalisis keseluruhan fungsionalitas basis data yang diperlukan, masalah antarmuka pengguna, dan perangkat lunak yang menangani komunikasi dengan lokasi tambang. Dari analisis ini, ia

memperkirakan perkiraan biaya untuk mengembangkan perangkat lunak. Dia menemukan bahwa solusi yang melibatkan pemeliharaan database lokal di lokasi tambang dan secara berkala memperbarui database pusat layak secara finansial dan teknis. Manajer proyek mendiskusikan solusi ini dengan presiden GMC Ltd., yang menunjukkan bahwa solusi yang diusulkan dapat diterima oleh mereka.

Analisis dan spesifikasi persyaratan

Tujuan dari fase analisis dan spesifikasi kebutuhan adalah untuk memahami kebutuhan yang tepat dari pelanggan dan untuk mendokumentasikannya dengan benar. Fase ini terdiri dari dua aktivitas yang berbeda, yaitu pengumpulan dan analisis kebutuhan, dan spesifikasi kebutuhan. Berikut ini adalah gambaran umum tentang dua kegiatan:

- Pengumpulan dan analisis persyaratan: Tujuan dari aktivitas pengumpulan persyaratan adalah untuk mengumpulkan semua informasi yang relevan mengenai perangkat lunak yang akan dikembangkan dari pelanggan dengan maksud untuk memahami persyaratan dengan jelas. Untuk ini, persyaratan pertama dikumpulkan dari pelanggan dan kemudian persyaratan yang dikumpulkan dianalisis. Tujuan dari aktivitas analisis kebutuhan adalah untuk menghilangkan ketidaklengkapan dan inkonsistensi dalam persyaratan yang dikumpulkan ini. Perhatikan bahwa persyaratan n tidak konsisten adalah persyaratan di mana beberapa bagian dari persyaratan bertentangan dengan beberapa bagian lainnya. Di sisi lain, persyaratan n tidak lengkap adalah persyaratan di mana beberapa bagian dari persyaratan aktual telah dihilangkan.
- Spesifikasi persyaratan: Setelah pengumpulan persyaratan dan kegiatan analisis selesai, persyaratan yang diidentifikasi didokumentasikan. Ini disebut dokumen spesifikasi persyaratan perangkat lunak (SRS). Dokumen SRS ditulis menggunakan terminologi pengguna akhir. Hal ini membuat dokumen SRS dapat dimengerti oleh pelanggan. Oleh karena itu, pemahaman terhadap dokumen SRS merupakan hal yang penting. Dokumen SRS biasanya berfungsi sebagai kontrak antara tim pengembangan dan pelanggan. Setiap perselisihan di masa depan antara pelanggan dan developer dapat diselesaikan dengan memeriksa dokumen SRS. Oleh karena itu, dokumen SRS merupakan dokumen penting yang harus dipahami secara menyeluruh oleh tim pengembangan, dan ditinjau bersama dengan pelanggan. Dokumen SRS tidak hanya menjadi dasar untuk melakukan semua kegiatan pengembangan, tetapi beberapa dokumen seperti manual pengguna, rencana pengujian sistem, dll disiapkan langsung berdasarkan itu. Dalam Bab 4, kita akan memeriksa aktivitas analisis kebutuhan dan berbagai isu yang terlibat dalam mengembangkan dokumen SRS yang baik secara lebih rinci.

Desain

Tujuan dari fase desain adalah untuk mengubah persyaratan yang ditentukan dalam dokumen SRS menjadi struktur yang cocok untuk implementasi dalam beberapa bahasa pemrograman. Dalam istilah teknis, selama fase desain rancangan roftware berasal dari dokumen SRS. Dua pendekatan desain yang sangat berbeda sedang populer digunakan saat ini—pendekatan desain prosedural dan berorientasi objek.

- **Pendekatan desain prosedural**: Pendekatan desain tradisional digunakan dalam banyak proyek pengembangan perangkat lunak saat ini. Teknik desain tradisional ini didasarkan pada pendekatan desain berorientasi aliran data. Ini terdiri dari dua kegiatan penting; analisis terstruktur pertama dari spesifikasi persyaratan dilakukan di mana struktur rinci masalah diperiksa. Ini diikuti oleh langkah desain terstruktur di mana hasil analisis terstruktur ditransformasikan ke dalam desain perangkat lunak.

  Selama analisis terstruktur, persyaratan fungsional yang ditentukan dalam dokumen SRS didekomposisi menjadi subfungsi dan aliran data di antara subfungsi ini dianalisis dan

direpresentasikan secara diagram dalam bentuk DFD. Teknik DFD dibahas pada Bab 6. Desain terstruktur dilakukan setelah aktivitas analisis terstruktur selesai. Desain terstruktur terdiri dari dua aktivitas utama—desain arsitektur (juga disebut desain tingkat tinggi) dan desain detail (juga disebut desain tingkat rendah). Desain tingkat tinggi melibatkan penguraian sistem ke dalam modul, dan mewakili antarmuka dan hubungan doa di antara modul. Desain perangkat lunak tingkat tinggi kadang-kadang disebut sebagai rancangan roftware. Selama aktivitas desain terperinci, internal modul individual seperti struktur data dan algoritma modul dirancang dan didokumentasikan.

- **Pendekatan desain berorientasi objek:** Dalam teknik ini, berbagai objek yang terjadi di domain masalah dan domain solusi pertama kali diidentifikasi dan hubungan berbeda yang ada di antara objek-objek ini diidentifikasi. Struktur objek lebih disempurnakan untuk mendapatkan desain detail. Pendekatan OOD dikreditkan memiliki beberapa manfaat seperti waktu dan upaya pengembangan yang lebih rendah, dan pemeliharaan perangkat lunak yang lebih baik. Teknik desain berorientasi objek dibahas dalam Bab 7 dan 8.

  Tujuan dari fase pengkodean dan pengujian unit adalah untuk menerjemahkan desain perangkat lunak ke dalam kode sumber dan untuk memastikan bahwa masing-masing fungsi bekerja dengan benar. Fase pengkodean juga kadang-kadang disebut fase implementasi, karena desain diimplementasikan ke dalam solusi yang bisa diterapkan dalam fase ini. Setiap komponen desain diimplementasikan sebagai modul program. Produk akhir dari fase ini adalah satu set modul program yang telah diuji unit secara individual. Tujuan utama dari pengujian unit adalah untuk menentukan kerja yang benar dari masing-masing modul. Kegiatan khusus yang dilakukan selama pengujian unit termasuk merancang kasus uji, pengujian, debugging untuk memperbaiki masalah, dan pengelolaan kasus uji. Kita akan membahas teknik pengkodean dan pengujian unit di Bab 10.

Integrasi dan pengujian sistem Integrasi modul yang berbeda dilakukan segera setelah mereka dikodekan dan diuji unit. Selama fase integrasi dan pengujian sistem, modul yang berbeda diintegrasikan secara terencana. Berbagai modul yang menyusun perangkat lunak hampir tidak pernah terintegrasi dalam satu bidikan (dapatkah Anda menebak alasannya?). Integrasi berbagai modul biasanya dilakukan secara bertahap melalui beberapa langkah. Selama setiap langkah integrasi, modul yang direncanakan sebelumnya ditambahkan ke sistem yang terintegrasi sebagian dan sistem yang dihasilkan diuji. Akhirnya, setelah semua modul berhasil diintegrasikan dan diuji, sistem kerja penuh diperoleh. Pengujian sistem dilakukan pada sistem yang berfungsi penuh ini. Pengujian integrasi dilakukan untuk memverifikasi bahwa antarmuka di antara unit yang berbeda bekerja dengan memuaskan. Di sisi lain, tujuan pengujian sistem adalah untuk memastikan bahwa sistem yang dikembangkan sesuai dengan persyaratan yang telah ditetapkan dalam dokumen SRS.

Pengujian sistem biasanya terdiri dari tiga jenis aktivitas pengujian yang berbeda:

- Developer-testing: Ini adalah pengujian sistem yang dilakukan oleh tim pengembang.
- customer-testing: Ini adalah pengujian sistem yang dilakukan oleh sekelompok pelanggan yang ramah.
- Pengujian penerimaan: Setelah perangkat lunak dikirimkan, pelanggan melakukan pengujian sistem untuk menentukan apakah akan menerima perangkat lunak yang dikirimkan atau menolaknya.

Kita membahas berbagai integrasi dan teknik pengujian sistem secara lebih rinci di Bab 10.

Pemeliharaan

Upaya total yang dihabiskan untuk pemeliharaan perangkat lunak biasa selama fase operasinya jauh lebih banyak daripada yang diperlukan untuk mengembangkan perangkat lunak itu sendiri. Banyak penelitian yang dilakukan di masa lalu mengkonfirmasi hal ini dan menunjukkan bahwa rasio upaya relatif untuk mengembangkan produk perangkat lunak yang khas dan total upaya yang dihabiskan untuk pemeliharaannya kira-kira 40:60.
Pemeliharaan diperlukan dalam tiga jenis situasi berikut:

- Pemeliharaan korektif: Jenis pemeliharaan ini dilakukan untuk memperbaiki kesalahan yang tidak ditemukan selama fase pengembangan produk.
- Perawatan sempurna: Jenis perawatan ini dilakukan untuk meningkatkan kinerja sistem, atau untuk meningkatkan fungsionalitas sistem berdasarkan permintaan pelanggan.
- Pemeliharaan adaptif: Pemeliharaan adaptif biasanya diperlukan untuk mem-porting perangkat lunak agar berfungsi di lingkungan baru. Misalnya, porting mungkin diperlukan agar perangkat lunak dapat bekerja pada platform komputer baru atau dengan sistem operasi baru.

Kekurangan model air terjun klasik

Model air terjun klasik adalah model yang sangat sederhana dan intuitif. Namun, ia menderita beberapa kekurangan. Mari kita identifikasi beberapa kekurangan penting dari model air terjun klasik:

Tidak ada jalur umpan balik: Dalam model air terjun klasik, evolusi perangkat lunak dari satu fase ke fase berikutnya dianalogikan dengan air terjun. Sama seperti air di air terjun setelah mengalir ke bawah tidak dapat mengalir kembali, setelah fase selesai, aktivitas yang dilakukan di dalamnya dan semua artefak yang dihasilkan dalam fase ini dianggap final dan ditutup untuk pengerjaan ulang. Ini mengharuskan semua aktivitas selama fase dilakukan dengan sempurna.

Model air terjun klasik adalah idealis dalam arti mengasumsikan bahwa tidak ada kesalahan yang pernah dilakukan oleh developer selama salah satu fase siklus hidup, dan oleh karena itu, tidak memasukkan mekanisme untuk koreksi kesalahan. Berlawanan dengan asumsi mendasar yang dibuat oleh model air terjun klasik, dalam lingkungan pengembangan praktis, developer melakukan banyak kesalahan di hampir setiap aktivitas yang mereka lakukan selama berbagai fase siklus hidup. Bagaimanapun, programmer adalah manusia dan seperti pepatah lama mengatakan untuk berbuat salah adalah manusiawi. Penyebab kesalahan bisa banyak—pengawasan, interpretasi yang salah, penggunaan skema solusi yang salah, kesenjangan komunikasi, dll. Cacat ini biasanya terdeteksi jauh di kemudian hari dalam siklus hidup. Misalnya, cacat desain mungkin tidak diketahui sampai tahap pengkodean atau pengujian. Setelah cacat terdeteksi di lain waktu, developer perlu mengulang beberapa pekerjaan yang dilakukan selama fase itu dan juga mengulang pekerjaan fase selanjutnya yang terpengaruh oleh pengerjaan ulang. Oleh karena itu, dalam setiap proyek pengembangan perangkat lunak non-sepele, menjadi hampir tidak mungkin untuk secara ketat mengikuti model pengembangan perangkat lunak air terjun klasik.

Sulit untuk mengakomodasi permintaan perubahan: Model ini mengasumsikan bahwa semua persyaratan pelanggan dapat didefinisikan secara lengkap dan benar di awal proyek. Ada banyak penekanan pada pembuatan serangkaian persyaratan yang jelas dan lengkap. Namun, sulit untuk mencapai ini bahkan dalam skenario proyek yang ideal. Persyaratan pelanggan biasanya terus berubah seiring waktu. Namun, dalam model ini menjadi sulit untuk mengakomodasi permintaan perubahan kebutuhan yang dibuat oleh pelanggan setelah fase spesifikasi persyaratan selesai, dan ini sering menjadi sumber ketidakpuasan pelanggan.

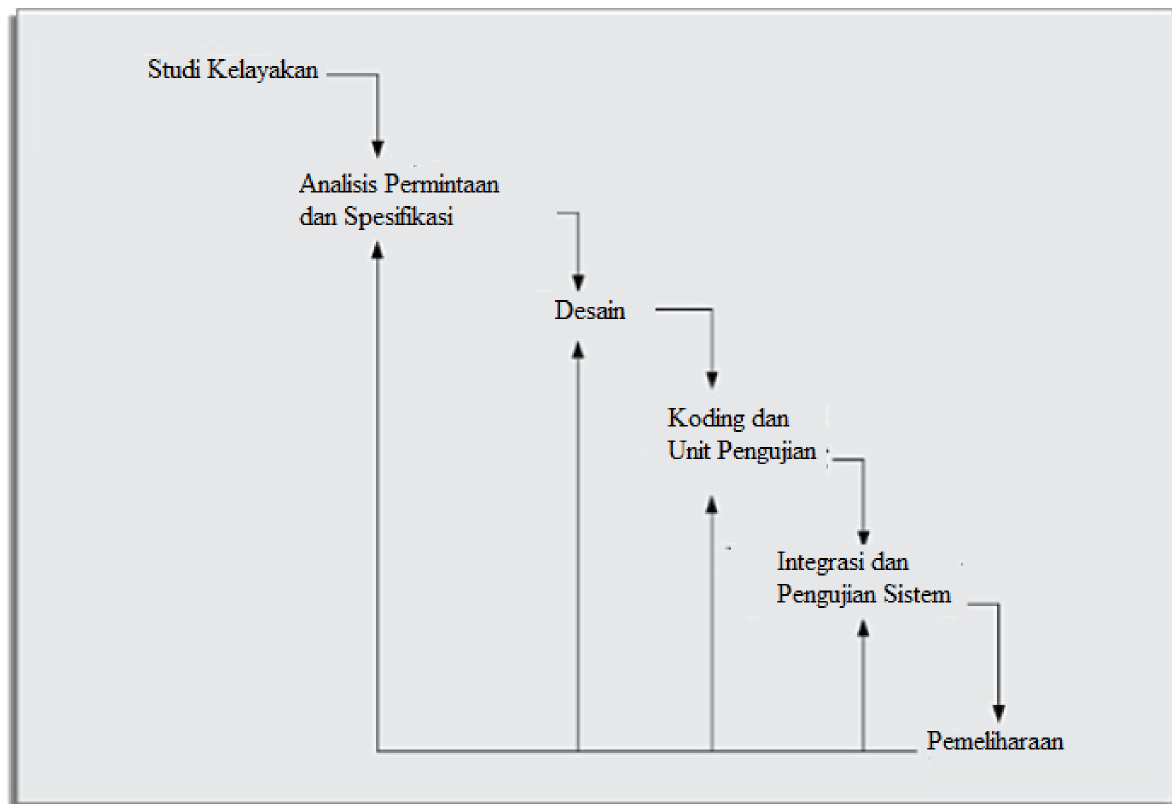
Koreksi kesalahan yang tidak efisien: Model ini menunda integrasi kode dan tugas pengujian sampai sangat terlambat ketika masalah lebih sulit untuk diselesaikan.

Tidak ada fase yang tumpang tindih: Model ini merekomendasikan agar fase dilakukan secara berurutan—fase baru dapat dimulai hanya setelah fase sebelumnya selesai. Namun, jarang mungkin untuk mematuhi rekomendasi ini dan itu menyebabkan sejumlah besar anggota tim menganggur untuk waktu yang lama. Misalnya, untuk pemanfaatan tenaga kerja yang efisien, tim pengujian mungkin perlu merancang kasus uji sistem segera setelah spesifikasi persyaratan selesai. (Kita akan membahas dalam Bab 10 bahwa kasus uji sistem dirancang hanya berdasarkan dokumen SRS). Dalam hal ini, kegiatan fase desain dan pengujian tumpang tindih. Akibatnya, aman untuk mengatakan bahwa dalam skenario pengembangan perangkat lunak praktis, daripada memiliki titik waktu yang tepat di mana transisi fase terjadi, fase yang berbeda perlu tumpang tindih untuk alasan biaya dan efisiensi.

Apakah model air terjun klasik berguna?

Sulit untuk menggunakan model air terjun klasik dalam proyek nyata. Dalam lingkungan pengembangan praktis apa pun, saat perangkat lunak mulai terbentuk, beberapa iterasi melalui tahapan air terjun yang berbeda menjadi perlu untuk koreksi kesalahan yang dilakukan selama berbagai fase. Oleh karena itu, model air terjun klasik hampir tidak dapat digunakan untuk pengembangan perangkat lunak. Tapi, seperti yang disarankan oleh Parnas [1972] dokumen akhir untuk produk harus ditulis seolah-olah produk dikembangkan menggunakan air terjun klasik murni. Terlepas dari model siklus hidup yang sebenarnya diikuti untuk pengembangan produk, dokumen akhir selalu ditulis untuk mencerminkan model pengembangan air terjun klasik, sehingga pemahaman dokumen menjadi lebih mudah bagi siapa pun yang membaca dokumen.

Alasan di balik penyusunan dokumen berdasarkan model air terjun klasik dapat dijelaskan dengan menggunakan metafora Hoare tentang teorema matematika [1994] membuktikan — Seorang ahli matematika menyajikan bukti sebagai satu rantai deduksi, meskipun buktinya mungkin berasal dari satu set berbelit-belit upaya parsial, jalan buntu dan mundur. Bayangkan betapa sulitnya untuk memahami, jika seorang ahli matematika menyajikan bukti dengan mempertahankan semua backtracking, koreksi kesalahan, dan perbaikan solusi yang dia buat saat mengerjakan bukti.



Gambar 2.3 Model air terjun iteratif.

Model Air Terjun Berulang

Dibagian sebelumnya saya sudah menunjukkan bahwa proyek pengembangan perangkat lunak praktis, model air terjun klasik sulit digunakan. Model air terjun klasik sebagai model idealis. Dalam konteks ini, model air terjun berulang dapat dianggap menggabungkan perubahan yang diperlukan pada model air terjun klasik agar dapat digunakan dalam proyek pengembangan perangkat lunak praktis. Perubahan utama yang dibawa oleh model air terjun iteratif ke model air terjun klasik adalah dalam bentuk memberikan jalur umpan balik dari setiap fase ke fase sebelumnya.

Jalur umpan balik yang diperkenalkan oleh model air terjun berulang ditunjukkan pada Gambar 2.3. Jalur umpan balik memungkinkan untuk mengoreksi kesalahan yang dilakukan oleh programmer selama beberapa fase, ketika dan ketika ini terdeteksi di fase selanjutnya. Misalnya, jika selama fase pengujian kesalahan desain diidentifikasi, maka jalur umpan balik memungkinkan desain untuk dikerjakan ulang dan perubahan tersebut tercermin dalam dokumen desain dan semua dokumen berikutnya lainnya. Harap perhatikan bahwa pada Gambar 2.3 tidak ada jalur umpan balik ke tahap kelayakan. Hal ini karena setelah sebuah tim menerima untuk mengambil sebuah proyek, tidak mudah menyerah proyek karena alasan hukum dan moral. Hampir setiap model siklus hidup yang kita bahas bersifat iteratif, kecuali model air terjun klasik dan model V—yang sifatnya sekuensial. Dalam model sekuensial, setelah fase selesai, tidak ada produk kerja dari fase itu yang diubah kemudian.

Fase penahanan kesalahan

Tidak peduli seberapa berhati-hatinya seorang programmer, dia mungkin akan melakukan beberapa kesalahan atau lainnya saat melakukan aktivitas siklus hidup. Kesalahan ini mengakibatkan kesalahan (juga disebut kesalahan atau bug) dalam produk kerja. Adalah menguntungkan untuk mendeteksi kesalahan ini dalam fase yang sama di mana mereka terjadi, karena deteksi dini bug mengurangi upaya dan waktu yang diperlukan untuk

memperbaikinya. Misalnya, jika masalah desain terdeteksi dalam fase desain itu sendiri, maka masalah tersebut dapat ditangani dengan lebih mudah daripada jika kesalahan diidentifikasi, katakanlah, pada akhir fase pengujian. Dalam kasus selanjutnya, akan diperlukan tidak hanya untuk mendesain ulang, tetapi juga untuk mengulang pengkodean yang relevan serta aktivitas pengujian dengan tepat, sehingga menimbulkan biaya yang lebih tinggi. Mungkin tidak selalu mungkin untuk mendeteksi semua kesalahan dalam fase yang sama di mana mereka dibuat. Namun demikian, kesalahan harus dideteksi sedini mungkin. Prinsip mendeteksi kesalahan sedekat mungkin dengan titik komitmen mereka dikenal sebagai fase penahanan kesalahan.

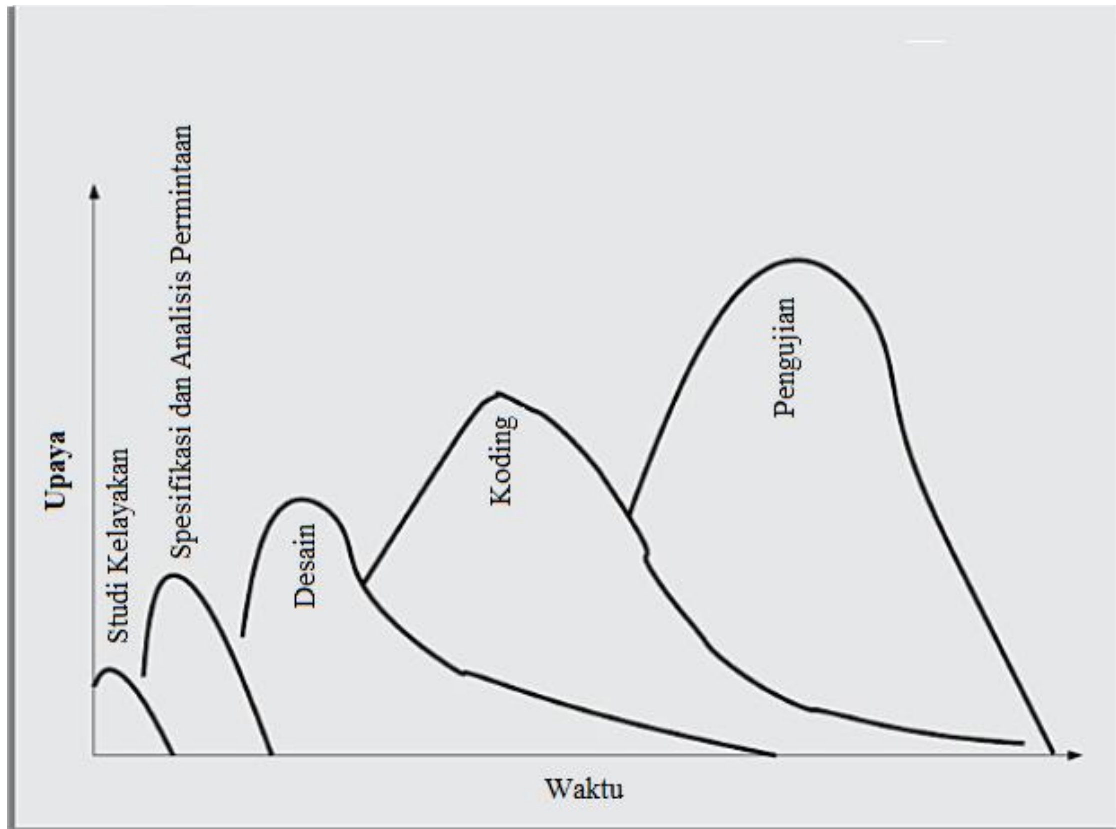
Untuk mencapai fase penahanan kesalahan, bagaimana developer dapat mendeteksi hampir semua kesalahan yang mereka lakukan dalam fase yang sama? Bagaimanapun, produk akhir dari banyak fase adalah dokumen teks atau grafik, mis. Dokumen SRS, dokumen desain, dokumen rencana pengujian, dll. Teknik yang populer adalah meninjau dokumen yang dihasilkan di akhir fase secara ketat.

Fase tumpang tindih

Meskipun model air terjun yang ketat membayangkan transisi yang tajam terjadi dari satu fase ke fase berikutnya (lihat Gambar 2.3), dalam praktiknya kegiatan fase yang berbeda tumpang tindih (seperti yang ditunjukkan pada Gambar 2.4) karena dua alasan utama:

- Terlepas dari upaya terbaik untuk mendeteksi kesalahan dalam fase yang sama di mana kesalahan itu dilakukan, beberapa kesalahan lolos dari deteksi dan terdeteksi pada fase selanjutnya. Kesalahan yang kemudian terdeteksi ini menyebabkan aktivitas dari beberapa fase yang sudah selesai dikerjakan ulang. Jika kita mempertimbangkan pengerjaan ulang tersebut setelah fase selesai, kita dapat mengatakan bahwa kegiatan yang berkaitan dengan fase tidak berakhir pada penyelesaian fase, tetapi tumpang tindih dengan fase lain seperti yang ditunjukkan pada Gambar 2.4.
- Alasan penting untuk fase tumpang tindih adalah bahwa biasanya pekerjaan yang diperlukan untuk dilakukan dalam fase dibagi di antara anggota tim. Beberapa anggota dapat menyelesaikan bagian pekerjaan mereka lebih awal dari anggota lainnya. Jika transisi fase yang ketat dipertahankan, maka anggota tim yang menyelesaikan pekerjaannya lebih awal akan menganggur menunggu fase selesai, dan dikatakan dalam keadaan memblokir. Dengan demikian developer yang menyelesaikan lebih awal akan menganggur sambil menunggu rekan satu timnya menyelesaikan pekerjaan yang ditugaskan. Jelas ini adalah penyebab pemborosan sumber daya dan sumber eskalasi biaya dan inefisiensi. Akibatnya, dalam proyek nyata, fase dibiarkan tumpang tindih. Artinya, setelah developer menyelesaikan tugas pekerjaannya untuk suatu fase, melanjutkan untuk memulai pekerjaan untuk fase berikutnya, tanpa menunggu semua anggota timnya menyelesaikan alokasi pekerjaan masing-masing.

Mempertimbangkan situasi ini, distribusi upaya untuk fase yang berbeda dengan waktu akan seperti yang ditunjukkan pada Gambar 2.4.



Gambar 2.4 Distribusi upaya untuk berbagai fase dalam model air terjun iteratif.

Kekurangan model air terjun berulang

Model air terjun berulang adalah model pengembangan perangkat lunak yang sederhana dan intuitif. Itu digunakan dengan memuaskan selama tahun 1970-an dan 1980-an. Namun, karakteristik proyek pengembangan perangkat lunak telah berubah secara drastis selama bertahun-tahun. Pada 1970-an dan 1960-an, proyek pengembangan perangkat lunak berlangsung beberapa tahun dan sebagian besar melibatkan pengembangan produk perangkat lunak generik. Proyek sekarang lebih pendek, dan melibatkan pengembangan perangkat lunak yang disesuaikan. Selanjutnya, perangkat lunak sebelumnya dikembangkan dari awal. Sekarang penekanannya adalah pada penggunaan kembali kode dan artefak proyek lainnya sebanyak mungkin. Model berbasis air terjun telah bekerja dengan memuaskan selama bertahun-tahun terakhir di masa lalu. Situasinya telah berubah secara substansial sekarang. Seperti yang ditunjukkan dalam bab pertama beberapa dekade yang lalu, setiap perangkat lunak dikembangkan dari awal. Sekarang, tidak hanya perangkat lunak yang menjadi sangat besar dan kompleks, sangat sedikit (jika ada) proyek perangkat lunak yang dikembangkan dari awal. Layanan perangkat lunak (perangkat lunak yang disesuaikan) siap menjadi jenis proyek yang dominan. Dalam proyek pengembangan perangkat lunak saat ini, penggunaan model air terjun menyebabkan beberapa masalah. Dalam konteks ini, model tangkas telah diusulkan sekitar satu dekade yang lalu yang mencoba untuk mengatasi kekurangan penting dari model air terjun dengan menyarankan modifikasi radikal tertentu pada gaya pengembangan perangkat lunak air terjun. Beberapa kekurangan yang mencolok dari model air terjun ketika digunakan dalam proyek pengembangan perangkat lunak saat ini adalah sebagai berikut:

Sulit untuk mengakomodasi permintaan perubahan: Masalah utama dengan model air terjun adalah bahwa persyaratan harus dibekukan sebelum pengembangan dimulai. Berdasarkan persyaratan yang dibekukan, rencana terperinci dibuat untuk kegiatan yang akan

dilakukan selama fase desain, pengkodean, dan pengujian. Karena kegiatan direncanakan untuk seluruh durasi, upaya dan sumber daya yang substansial diinvestasikan dalam kegiatan seperti mengembangkan spesifikasi persyaratan lengkap, desain untuk fungsionalitas lengkap, dan sebagainya. Oleh karena itu, mengakomodasi permintaan perubahan kecil sekalipun setelah kegiatan pembangunan berlangsung tidak hanya membutuhkan perombakan rencana, tetapi juga artefak yang telah dikembangkan. Setelah persyaratan dibekukan, model air terjun tidak menyediakan ruang lingkup untuk modifikasi apa pun pada persyaratan.

Sementara model air terjun tidak fleksibel untuk perubahan persyaratan di kemudian hari, bukti yang dikumpulkan dari beberapa proyek menunjukkan fakta bahwa perubahan persyaratan di kemudian hari hampir tidak dapat dihindari. Bahkan untuk proyek dengan profesional yang sangat berpengalaman di semua tingkatan, serta pelanggan yang paham komputer, persyaratan sering terlewatkan dan juga disalahartikan. Kecuali jika permintaan perubahan didorong, fungsionalitas yang dikembangkan akan tidak sesuai dengan kebutuhan pelanggan yang sebenarnya. Perubahan persyaratan dapat terjadi karena berbagai alasan, antara lain: persyaratan tidak jelas bagi pelanggan, persyaratan disalahpahami, proses bisnis pelanggan mungkin berubah setelah dokumen SRS ditandatangani, dll. Bahkan, pelanggan menjadi lebih jelas pemahaman tentang persyaratan mereka hanya setelah bekerja pada sistem yang dikembangkan dan diinstal sepenuhnya. Asumsi dasar yang dibuat dalam model air terjun berulang bahwa pengumpulan dan analisis persyaratan metodis saja akan secara komprehensif dan benar mengidentifikasi semua persyaratan pada akhir fase persyaratan adalah cacat.

Pengiriman tambahan tidak didukung: Dalam model air terjun berulang, perangkat lunak lengkap dikembangkan dan diuji sepenuhnya sebelum dikirimkan ke pelanggan. Tidak ada ketentuan untuk pengiriman perantara terjadi. Ini bermasalah karena aplikasi yang lengkap mungkin membutuhkan waktu beberapa bulan atau tahun untuk diselesaikan dan dikirimkan ke pelanggan. Pada saat perangkat lunak dikirimkan, diinstal, dan siap digunakan, proses bisnis pelanggan mungkin telah berubah secara substansial. Hal ini membuat aplikasi yang dikembangkan kurang sesuai dengan kebutuhan pelanggan.

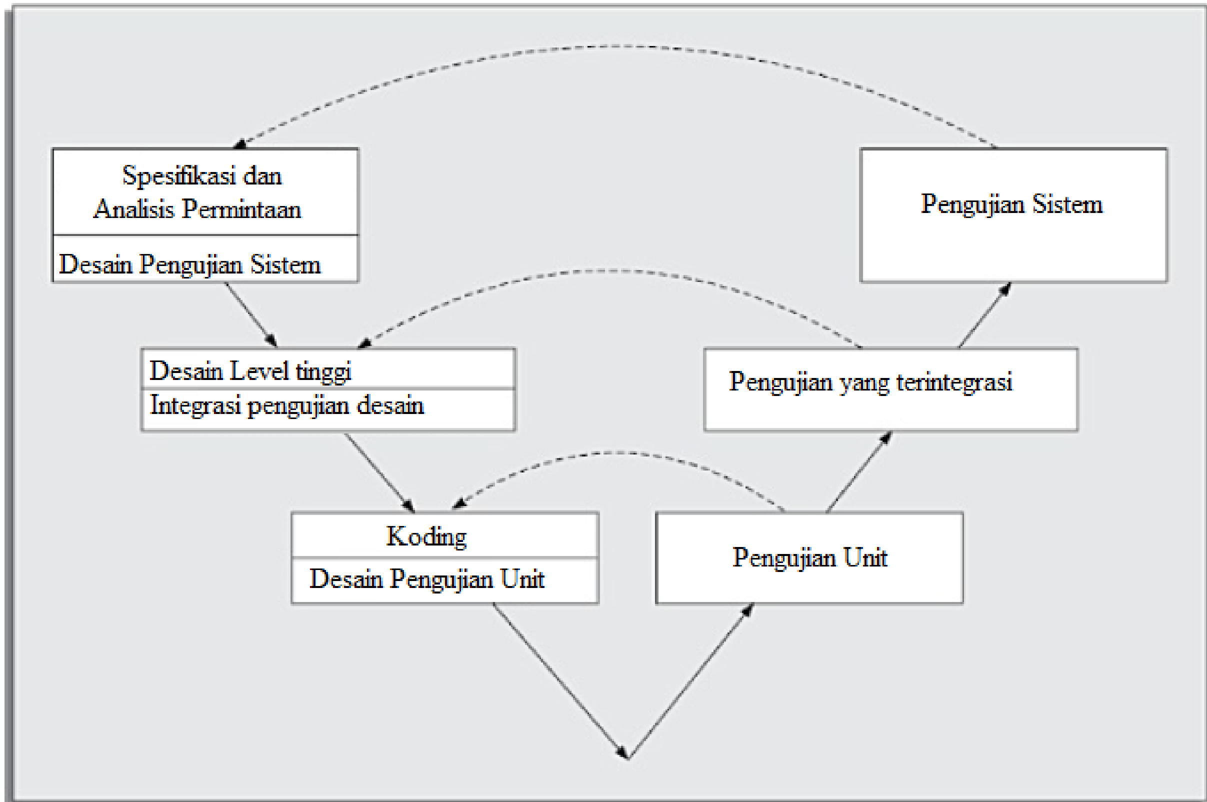
Fase tumpang tindih tidak didukung: Untuk sebagian besar proyek kehidupan nyata, menjadi sulit untuk mengikuti urutan fase kaku yang ditentukan oleh model air terjun. Dengan istilah urutan fase kaku, kita berarti bahwa fase dapat dimulai hanya setelah fase sebelumnya selesai dalam segala hal. Seperti yang telah dibahas, kepatuhan yang ketat pada model air terjun menciptakan status pemblokiran. Model air terjun biasanya diadaptasi untuk digunakan dalam proyek kehidupan nyata dengan memungkinkan tumpang tindih berbagai fase seperti yang ditunjukkan pada Gambar 2.4.

Koreksi kesalahan terlalu mahal: Dalam model air terjun, validasi ditunda hingga pengembangan perangkat lunak selesai. Akibatnya, cacat yang terlihat pada saat validasi menimbulkan pengerjaan ulang yang mahal dan mengakibatkan peningkatan biaya dan pengiriman tertunda.

Interaksi pelanggan terbatas: Model ini mendukung interaksi pelanggan yang sangat terbatas. Secara umum diterima bahwa perangkat lunak yang dikembangkan secara terpisah dari pelanggan adalah penyebab banyak masalah. Faktanya, interaksi hanya terjadi pada awal proyek dan pada penyelesaian proyek. Akibatnya, perangkat lunak yang dikembangkan biasanya tidak sesuai dengan kebutuhan aktual pelanggan.

Bobot berat: Model air terjun terlalu menekankan dokumentasi. Sebagian besar waktu developer dihabiskan untuk menyiapkan dokumen, dan merevisinya saat terjadi perubahan selama siklus hidup. Dokumentasi berat meskipun berguna selama pemeliharaan dan untuk melaksanakan tinjauan, merupakan sumber inefisiensi tim.

Tidak ada dukungan untuk penanganan risiko dan penggunaan kembali kode: Menjadi sulit untuk menggunakan model air terjun dalam proyek yang rentan terhadap berbagai jenis risiko, atau yang melibatkan penggunaan kembali artefak pengembangan yang ada secara signifikan. Harap diingat bahwa jenis proyek layanan perangkat lunak biasanya melibatkan penggunaan kembali yang signifikan.



Gambar 2.5 Model-V.

V-Model

Model proses pengembangan yang populer, model-V adalah varian dari model air terjun. Seperti halnya model air terjun, model ini mendapatkan namanya dari tampilan visualnya (lihat Gambar 2.5). Dalam model ini, aktivitas verifikasi dan validasi dilakukan sepanjang siklus hidup pengembangan, dan oleh karena itu kemungkinan bug dalam produk kerja sangat berkurang. Oleh karena itu, model ini umumnya dianggap cocok untuk digunakan dalam proyek-proyek yang berkaitan dengan pengembangan perangkat lunak yang kritis terhadap keselamatan yang dituntut untuk memiliki keandalan yang tinggi. Seperti yang ditunjukkan pada Gambar 2.5, ada dua fase utama—fase pengembangan dan validasi. Bagian kiri model terdiri dari fase pengembangan dan bagian kanan terdiri dari fase validasi.

- Pada setiap tahap pengembangan, bersamaan dengan pengembangan produk kerja, dilakukan desain kasus uji dan rencana pengujian produk kerja, sedangkan pengujian aktual dilakukan pada tahap validasi. Rencana validasi yang dibuat selama fase pengembangan ini dilakukan dalam fase validasi yang sesuai yang telah ditunjukkan oleh busur putus-putus pada Gambar 2.5.
- Pada fase validasi, pengujian dilakukan dalam tiga langkah—unit, integrasi, dan pengujian sistem. Tujuan dari ketiga langkah pengujian yang berbeda ini selama fase validasi adalah untuk mendeteksi cacat yang muncul pada fase pengembangan perangkat lunak yang sesuai—analisis dan spesifikasi kebutuhan, desain, dan pengkodean masing-masing.

Model-V versus model air terjun

Model-V dapat dianggap sebagai perpanjangan dari model air terjun. Namun, ada perbedaan besar antara keduanya. Seperti yang telah disebutkan, berbeda dengan model air terjun berulang di mana aktivitas pengujian terbatas pada fase pengujian saja, dalam aktivitas pengujian model-V tersebar di seluruh siklus hidup. Seperti yang ditunjukkan pada Gambar 2.5, selama fase spesifikasi persyaratan, aktivitas desain rangkaian pengujian sistem berlangsung. Selama fase desain, kasus uji integrasi dirancang. Selama pengkodean, kasus uji unit dirancang. Dengan demikian, dapat dikatakan bahwa dalam model ini, kegiatan pengembangan dan validasi berjalan beriringan.

Keuntungan dari model-V

Keuntungan penting dari model-V dibandingkan model air terjun berulang adalah sebagai berikut:

- Dalam model-V, sebagian besar aktivitas pengujian (desain kasus pengujian, perencanaan pengujian, dll.) dilakukan secara paralel dengan aktivitas pengembangan. Oleh karena itu, sebelum fase pengujian dimulai, bagian penting dari kegiatan pengujian, termasuk desain kasus uji dan perencanaan pengujian, sudah selesai. Oleh karena itu, model ini biasanya mengarah ke fase pengujian yang lebih pendek dan pengembangan produk secara keseluruhan lebih cepat dibandingkan dengan model iteratif.
- Karena kasus uji dirancang ketika tekanan jadwal belum terbentuk, kualitas kasus uji biasanya lebih baik.
- Tim uji cukup sibuk selama siklus pengembangan berbeda dengan model air terjun di mana penguji hanya aktif selama fase pengujian. Ini mengarah pada pemanfaatan tenaga kerja yang lebih efisien.
- Dalam model-V, tim uji dikaitkan dengan proyek sejak awal. Oleh karena itu mereka membangun pemahaman yang baik tentang artefak pengembangan, dan ini pada gilirannya, membantu mereka untuk melakukan pengujian perangkat lunak yang efektif. Sebaliknya, dalam model air terjun sering kali tim penguji datang terlambat dalam siklus pengembangan, karena tidak ada aktivitas pengujian yang dilakukan sebelum dimulainya fase implementasi dan pengujian.

Kekurangan model-V

Menjadi turunan dari model air terjun klasik, model ini mewarisi sebagian besar kelemahan model air terjun.

Model Prototipe

Model prototipe juga merupakan model siklus hidup yang populer. Model prototyping dapat dianggap sebagai perpanjangan dari model air terjun. Model ini menyarankan membangun prototipe kerja sistem, sebelum pengembangan perangkat lunak yang sebenarnya. Prototipe adalah mainan dan implementasi kasar dari suatu sistem. Ini memiliki kemampuan fungsional yang terbatas, keandalan yang rendah, atau kinerja yang tidak efisien dibandingkan dengan perangkat lunak yang sebenarnya. Sebuah prototipe dapat dibangun dengan sangat cepat dengan menggunakan beberapa jalan pintas. Cara pintas biasanya melibatkan pengembangan fungsi yang tidak efisien, tidak akurat, atau dummy. Implementasi jalan pintas dari suatu fungsi, misalnya, dapat menghasilkan hasil yang diinginkan dengan menggunakan pencarian tabel daripada dengan melakukan perhitungan yang sebenarnya. Biasanya istilah rapid prototyping digunakan ketika perangkat lunak digunakan untuk konstruksi prototipe. Misalnya, alat berdasarkan bahasa generasi keempat (4GL) dapat digunakan untuk membangun prototipe untuk bagian GUI.

Kebutuhan model prototyping

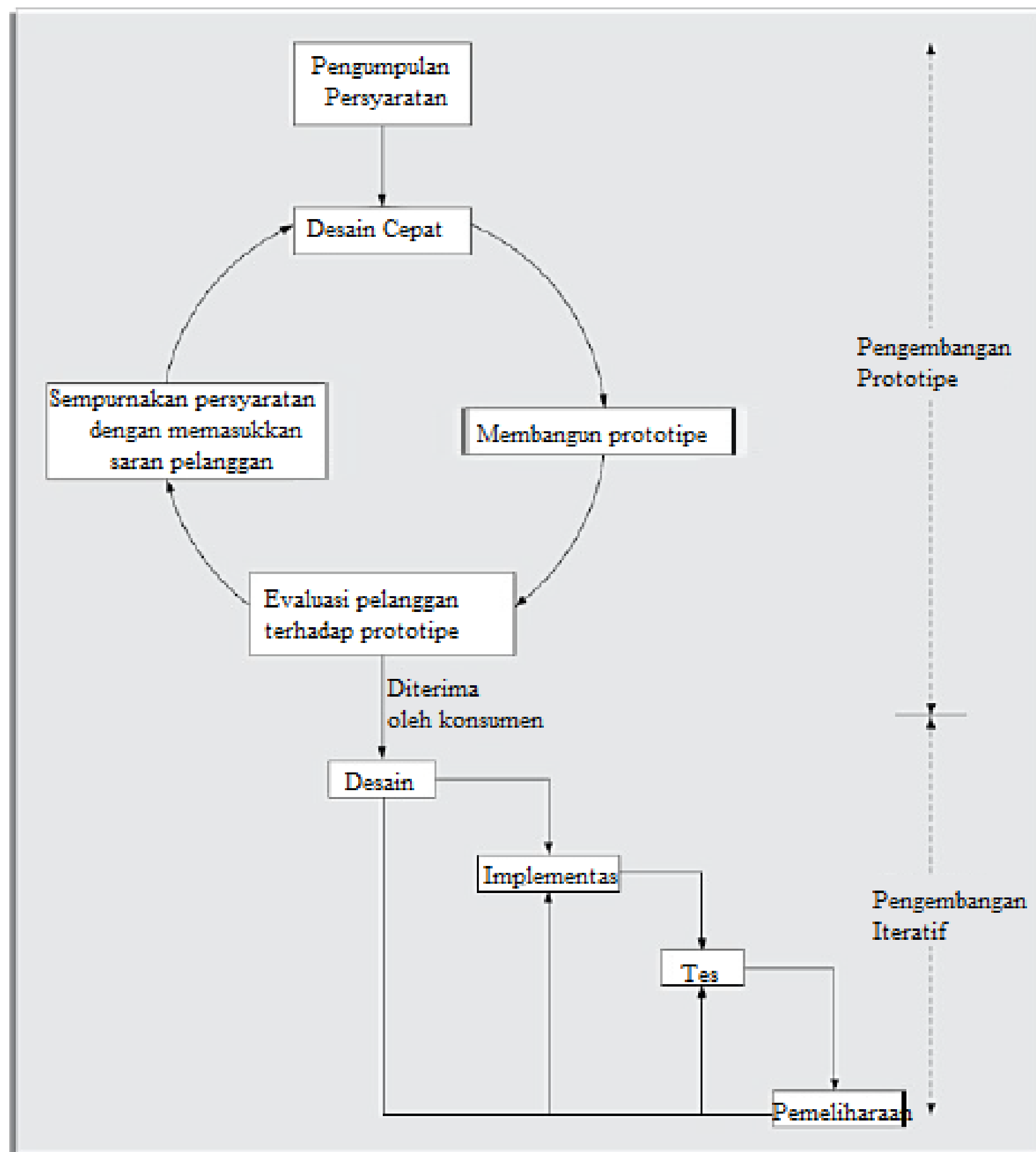
Model prototyping menguntungkan untuk digunakan untuk jenis proyek tertentu. Berikut ini adalah identifikasi tiga jenis proyek yang model prototyping dapat diikuti untuk keuntungan:

- Adalah menguntungkan untuk menggunakan model prototyping untuk pengembangan bagian antarmuka pengguna grafis (GUI) dari suatu aplikasi. Melalui penggunaan prototipe, menjadi lebih mudah untuk menggambarkan format input data, pesan, laporan, dan dialog interaktif kepada pelanggan. Ini adalah mekanisme yang berharga untuk mendapatkan pemahaman yang lebih baik tentang kebutuhan pelanggan. Dalam hal ini, model prototipe ternyata sangat berguna dalam mengembangkan bagian antarmuka pengguna grafis (GUI) dari suatu sistem. Bagi pengguna, menjadi lebih mudah untuk membentuk opini tentang apa yang akan lebih cocok dengan bereksperimen dengan antarmuka pengguna yang berfungsi, daripada mencoba membayangkan cara kerja antarmuka pengguna hipotetis. Bagian GUI dari sistem perangkat lunak hampir selalu dikembangkan menggunakan model prototipe.
- Model prototyping sangat berguna ketika solusi teknis yang tepat tidak jelas bagi tim pengembangan. Sebuah prototipe dapat membantu mereka untuk secara kritis memeriksa masalah teknis yang terkait dengan pengembangan produk. Misalnya, pertimbangkan situasi di mana tim pengembangan harus menulis penerjemah bahasa perintah sebagai bagian dari pengembangan antarmuka pengguna grafis. Misalkan tidak ada anggota tim yang pernah menulis kompiler sebelumnya. Kemudian, kurangnya keakraban dengan teknologi pengembangan yang dibutuhkan adalah risiko teknis. Risiko ini dapat diatasi dengan mengembangkan kompiler prototipe untuk bahasa yang sangat kecil untuk memahami masalah yang terkait dengan penulisan kompiler untuk bahasa perintah. Begitu mereka merasa percaya diri dalam menulis kompiler untuk bahasa kecil, mereka dapat menggunakan pengetahuan ini untuk mengembangkan kompiler untuk bahasa perintah. Seringkali, keputusan desain utama bergantung pada masalah seperti waktu respons pengontrol perangkat keras, atau efisiensi algoritma penyortiran, dll. Dalam keadaan seperti itu, prototipe sering kali merupakan cara terbaik untuk menyelesaikan masalah teknis.
- Alasan penting untuk mengembangkan prototipe adalah bahwa tidak mungkin untuk "melakukannya dengan benar" pertama kali. Seperti yang dianjurkan oleh Brooks [1975], seseorang harus merencanakan untuk membuang perangkat lunak untuk mengembangkan perangkat lunak yang baik nantinya. Dengan demikian, model prototyping dapat digunakan ketika pengembangan perangkat lunak yang sangat optimal dan efisien diperlukan.

Dari pembahasan di atas, dapat kita simpulkan sebagai berikut: *Model prototyping dianggap berguna untuk pengembangan tidak hanya bagian GUI dari perangkat lunak, tetapi juga untuk proyek perangkat lunak yang masalah teknis tertentu tidak jelas bagi tim pengembangan.*

Aktivitas siklus hidup model prototyping

Model prototyping pengembangan perangkat lunak secara grafis ditunjukkan pada Gambar 2.6. Seperti yang ditunjukkan pada Gambar 2.6, perangkat lunak dikembangkan melalui dua aktivitas utama—konstruksi prototipe dan pengembangan perangkat lunak berbasis air terjun berulang.



Gambar 2.6 Model pembuatan prototipe pengembangan perangkat lunak.

Pengembangan prototipe: Pengembangan prototipe dimulai dengan fase pengumpulan persyaratan awal. Sebuah desain cepat dilakukan dan prototipe dibangun. Prototipe yang dikembangkan diserahkan kepada pelanggan untuk dievaluasi. Berdasarkan umpan balik pelanggan, persyaratan disempurnakan dan prototipe dimodifikasi sesuai. Siklus mendapatkan umpan balik pelanggan dan memodifikasi prototipe berlanjut sampai pelanggan menyetujui prototipe.

Pengembangan berulang: Setelah pelanggan menyetujui prototipe, perangkat lunak yang sebenarnya dikembangkan menggunakan pendekatan air terjun berulang. Terlepas dari ketersediaan prototipe kerja, dokumen SRS biasanya perlu dikembangkan karena dokumen SRS sangat berharga untuk melakukan analisis ketertelusuran, verifikasi, dan desain kasus uji selama fase selanjutnya. Namun, untuk bagian GUI, analisis persyaratan dan fase spesifikasi menjadi berlebihan karena prototipe kerja yang telah disetujui oleh pelanggan berfungsi sebagai spesifikasi persyaratan animasi. Kode untuk prototipe biasanya dibuang. Namun, pengalaman yang dikumpulkan dari pengembangan prototipe sangat membantu dalam mengembangkan sistem yang sebenarnya.

Meskipun pembangunan prototipe sekali pakai mungkin memerlukan biaya tambahan, untuk sistem dengan persyaratan pelanggan yang tidak jelas dan untuk sistem dengan masalah teknis yang belum terselesaikan, biaya pengembangan keseluruhan biasanya ternyata lebih rendah dibandingkan dengan sistem setara yang dikembangkan menggunakan model air terjun berulang. Dengan membangun prototipe dan mengirimkannya untuk evaluasi pengguna, banyak persyaratan pelanggan dapat didefinisikan dengan benar dan masalah teknis diselesaikan dengan bereksperimen dengan prototipe. Ini meminimalkan permintaan perubahan di kemudian hari dari pelanggan dan biaya desain ulang terkait.

Kekuatan model prototyping

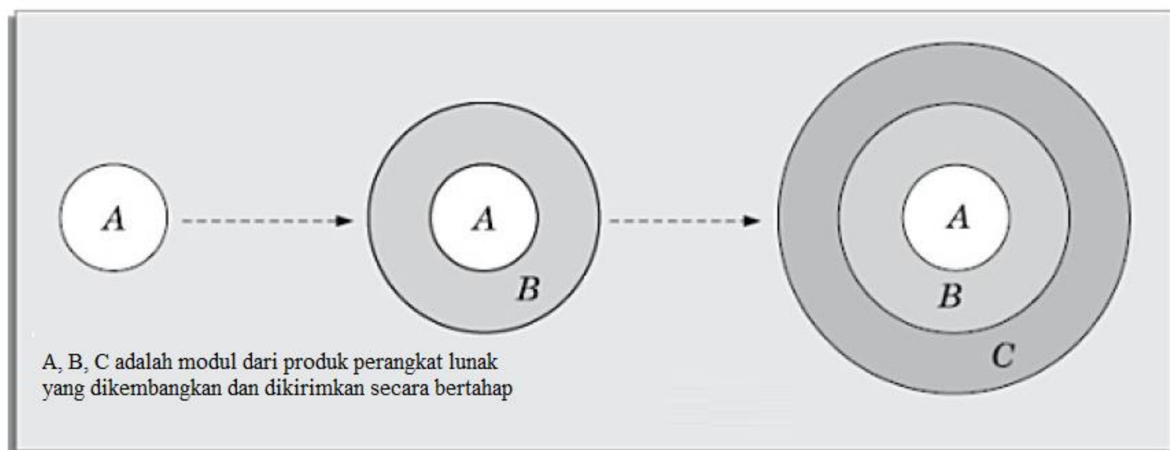
Model ini paling sesuai untuk proyek yang mengalami risiko teknis dan persyaratan. Prototipe yang dibangun membantu mengatasi risiko ini.

Kelemahan model prototyping

Model prototipe dapat meningkatkan biaya pengembangan untuk proyek-proyek yang merupakan pekerjaan pengembangan rutin dan tidak mengalami risiko yang signifikan. Bahkan ketika sebuah proyek rentan terhadap risiko, model prototyping hanya efektif untuk proyek-proyek yang risikonya dapat diidentifikasi di muka sebelum pengembangan dimulai. Karena prototipe dibangun hanya pada awal proyek, model prototipe tidak efektif untuk risiko yang diidentifikasi kemudian selama siklus pengembangan. Model prototyping tidak akan sesuai untuk proyek yang risikonya hanya dapat diidentifikasi setelah pengembangan berlangsung.

Model Pengembangan Inkremental

Model siklus hidup ini kadang-kadang disebut sebagai model versi yang berurutan dan kadang-kadang sebagai model inkremental. Dalam model siklus hidup ini, pertama-tama sebuah sistem kerja sederhana yang hanya mengimplementasikan beberapa fitur dasar dibangun dan dikirimkan ke pelanggan. Selama banyak iterasi yang berurutan, versi yang berurutan diimplementasikan dan dikirimkan ke pelanggan sampai sistem yang diinginkan terwujud. Model pengembangan inkremental telah ditunjukkan pada Gambar 2.7.



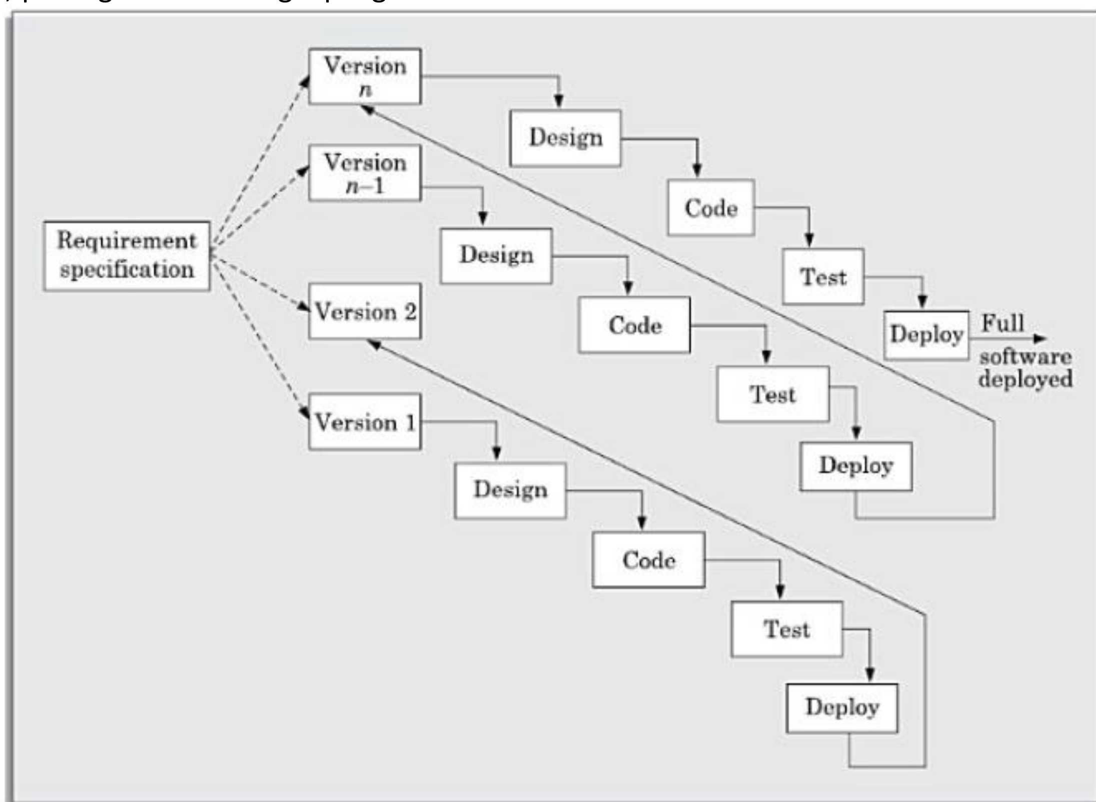
Gambar 2.7 Pengembangan perangkat lunak tambahan.

Aktivitas siklus hidup model pengembangan inkremental

Dalam model siklus hidup tambahan, persyaratan perangkat lunak pertama-tama dipecah menjadi beberapa modul atau fitur yang dapat dibangun dan dikirimkan secara bertahap. Ini telah digambarkan secara bergambar pada Gambar 2.7. Setiap saat, rencana dibuat hanya untuk kenaikan berikutnya dan tidak ada rencana jangka panjang yang dibuat. Oleh karena itu, menjadi lebih mudah untuk mengakomodasi permintaan perubahan dari pelanggan.

Tim developer pertama-tama berjanji untuk mengembangkan fitur inti dari sistem. Fitur inti atau dasar adalah fitur yang tidak perlu memanggil layanan apa pun dari fitur lainnya. Di sisi lain, fitur non-inti membutuhkan layanan dari fitur inti. Setelah fitur inti awal dikembangkan, fitur ini disempurnakan menjadi peningkatan tingkat kemampuan dengan menambahkan fungsionalitas baru dalam versi yang berurutan. Setiap versi inkremental biasanya dikembangkan menggunakan model pengembangan air terjun iteratif. Model inkremental secara skematis ditunjukkan pada Gambar 2.8. Karena setiap versi berturut-turut dari perangkat lunak dibangun dan dikirimkan ke pelanggan, umpan balik pelanggan diperoleh pada versi yang dikirimkan dan umpan balik ini digabungkan dalam versi berikutnya. Setiap versi perangkat lunak yang dikirimkan menggabungkan fitur tambahan dari versi sebelumnya dan juga menyempurnakan fitur yang telah dikirimkan ke pelanggan.

Model inkremental secara skematis ditunjukkan pada Gambar 2.8. Setelah pengumpulan persyaratan dan spesifikasi, persyaratan dibagi menjadi beberapa versi. Dimulai dengan inti (versi 1), di setiap kenaikan berturut-turut, versi berikutnya dibangun menggunakan model pengembangan air terjun berulang dan disebarkan di lokasi pelanggan. Setelah yang terakhir (ditampilkan sebagai versi n) telah dikembangkan dan digunakan di situs klien, perangkat lunak lengkap digunakan.



Gambar 2.8 Model inkremental pengembangan perangkat lunak.

Keuntungan

Model pengembangan inkremental menawarkan beberapa keuntungan. Dua yang penting adalah sebagai berikut:

- Pengurangan kesalahan: Modul inti digunakan oleh pelanggan sejak awal dan karena itu diuji secara menyeluruh. Ini mengurangi kemungkinan kesalahan dalam modul inti dari produk akhir, yang mengarah ke keandalan perangkat lunak yang lebih besar.

- Penyebaran sumber daya tambahan: Model ini meniadakan kebutuhan pelanggan untuk melakukan sumber daya yang besar sekaligus untuk pengembangan sistem. Ini juga menyelamatkan organisasi yang sedang berkembang dari mengerahkan sumber daya dan tenaga kerja yang besar untuk sebuah proyek sekaligus.

Model Evolusi

Model ini memiliki banyak fitur dari model inkremental. Seperti halnya model inkremental, perangkat lunak dikembangkan melalui sejumlah inkremen. Pada setiap kenaikan, konsep (fitur) diimplementasikan dan disebarkan di situs klien. Perangkat lunak secara berturut-turut disempurnakan dan diperkaya fitur hingga perangkat lunak lengkap direalisasikan. Ide utama di balik model siklus hidup evolusioner disampaikan oleh namanya. Dalam model pengembangan inkremental, persyaratan lengkap pertama kali dikembangkan dan dokumen SRS disiapkan. Sebaliknya, dalam model evolusi, persyaratan, rencana, perkiraan, dan solusi berkembang selama iterasi, daripada sepenuhnya didefinisikan dan dibekukan dalam upaya spesifikasi awal sebelum iterasi pengembangan dimulai. Evolusi tersebut konsisten dengan pola penemuan fitur yang tidak terduga dan perubahan fitur yang terjadi dalam pengembangan produk baru.

Meskipun model evolusi juga dapat dilihat sebagai perpanjangan dari model air terjun, tetapi menggabungkan pergeseran paradigma utama yang telah diadopsi secara luas di banyak model siklus hidup baru-baru ini. Karena alasan yang jelas, proses pengembangan perangkat lunak evolusi kadang-kadang disebut sebagai merancang sedikit, membangun sedikit, menguji sedikit, menyebarkan model kecil. Ini berarti bahwa setelah persyaratan ditentukan, aktivitas desain, pembuatan, pengujian, dan penerapan diulang. Sebuah representasi skema dari model evolusi pembangunan telah ditunjukkan pada Gambar 2.9.

Keuntungan

Model evolusioner pembangunan memiliki beberapa keunggulan. Dua keuntungan penting menggunakan model ini adalah sebagai berikut:

- Perolehan efektif dari kebutuhan pelanggan yang sebenarnya: Dalam model ini, pengguna mendapat kesempatan untuk bereksperimen dengan perangkat lunak yang dikembangkan sebagian jauh sebelum persyaratan lengkap dikembangkan. Oleh karena itu, model evolusioner membantu memperoleh kebutuhan pengguna secara akurat dengan bantuan umpan balik yang diperoleh pada pengiriman versi perangkat lunak yang berbeda. Akibatnya, permintaan perubahan setelah pengiriman perangkat lunak lengkap berkurang secara substansial.
- Penanganan permintaan perubahan yang mudah: Dalam model ini, penanganan permintaan perubahan lebih mudah karena tidak ada rencana jangka panjang yang dibuat. Akibatnya, pengerjaan ulang yang diperlukan karena permintaan perubahan biasanya jauh lebih kecil dibandingkan dengan model sekuensial.

Kekurangan

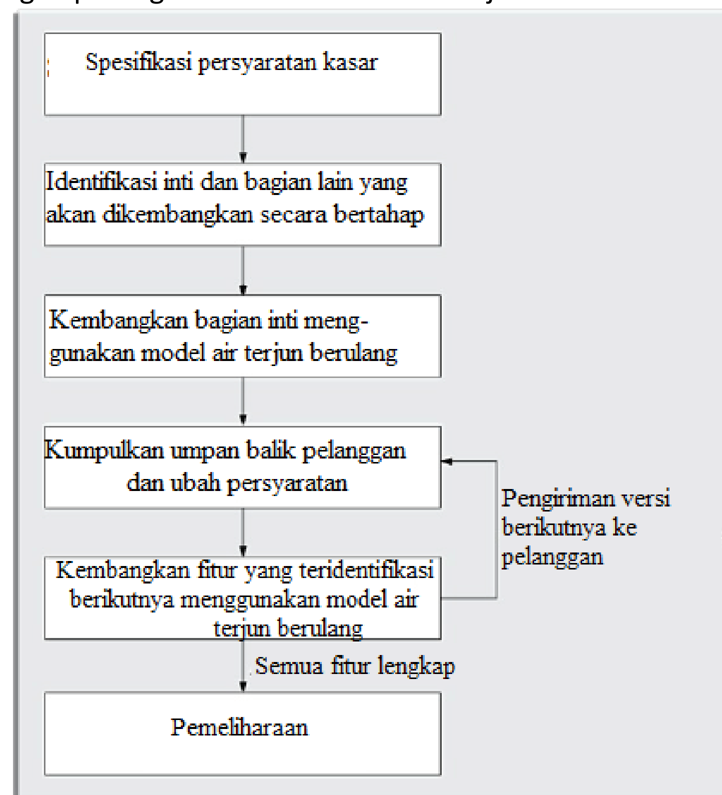
Kerugian utama dari model versi berturut-turut adalah sebagai berikut:

- ***Pembagian fitur menjadi bagian-bagian tambahan dapat menjadi non-sepele***: Untuk banyak proyek pengembangan, terutama untuk proyek-proyek berukuran kecil, sulit untuk membagi fitur yang diperlukan menjadi beberapa bagian yang dapat diimplementasikan dan disampaikan secara bertahap. Lebih jauh, bahkan untuk masalah yang lebih besar, seringkali fitur-fitur tersebut begitu saling terkait dan bergantung satu sama lain sehingga bahkan seorang ahli pun memerlukan upaya yang cukup besar untuk merencanakan pengiriman tambahan.
- ***Desain ad hoc:*** Karena pada suatu waktu desain hanya untuk peningkatan saat ini yang dilakukan, desain dapat menjadi ad hoc tanpa perhatian khusus diberikan

pada pemeliharaan dan optimalitas. Jelas, untuk masalah berukuran sedang dan untuk masalah yang kebutuhan pelanggannya jelas, model air terjun berulang dapat menghasilkan solusi yang lebih baik.

Penerapan model evolusi

Model evolusi biasanya berguna untuk produk yang sangat besar, di mana lebih mudah untuk menemukan modul untuk implementasi tambahan. Seringkali model evolusioner digunakan ketika pelanggan lebih memilih untuk menerima produk secara bertahap sehingga ia dapat mulai menggunakan fitur yang berbeda saat dan saat dikirimkan daripada menunggu sepanjang waktu hingga produk lengkap dikembangkan dan dikirimkan. Kategori penting lain dari proyek yang model evolusionernya cocok, adalah proyek yang menggunakan pengembangan berorientasi objek. Model evolusi sangat cocok untuk digunakan dalam proyek pengembangan perangkat lunak berorientasi objek.



Gambar 2.9 Model evolusi pengembangan perangkat lunak

Model evolusioner cocok untuk proyek pengembangan berorientasi objek, karena mudah untuk mempartisi perangkat lunak menjadi unit yang berdiri sendiri dalam hal kelas. Juga, kelas lebih atau kurang unit mandiri yang dapat dikembangkan secara mandiri.

2.3 *RAPID APPLICATION DEVELOPMENT* (RAD)

Model pengembangan aplikasi cepat/*Rapid Applicationm Development* (RAD) diusulkan pada awal tahun sembilan puluhan dalam upaya untuk mengatasi kekakuan model air terjun (dan turunannya) yang membuat sulit untuk mengakomodasi permintaan perubahan dari pelanggan. Ini mengusulkan beberapa ekstensi radikal untuk model air terjun. Model ini memiliki fitur model prototipe dan model evolusioner. Ini menyebarkan mo de l pengiriman evolusioner untuk mendapatkan dan menggabungkan umpan balik pelanggan pada versi yang dikirimkan secara bertahap.

Dalam model ini prototipe dibangun, dan secara bertahap fitur dikembangkan dan dikirimkan ke pelanggan. Tetapi tidak seperti model prototipe, prototipe tidak dibuang tetapi ditingkatkan dan digunakan dalam konstruksi perangkat lunak

Tujuan utama dari model RAD adalah sebagai berikut:

- Untuk mengurangi waktu yang dibutuhkan dan biaya yang dikeluarkan untuk mengembangkan sistem perangkat lunak.
- Untuk membatasi biaya mengakomodasi permintaan perubahan.
- Untuk mengurangi kesenjangan komunikasi antara pelanggan dan pengembang.

Motivasi utama

Dalam model air terjun berulang, persyaratan pelanggan perlu dikumpulkan, dianalisis, didokumentasikan, dan ditandatangani di muka, sebelum pengembangan apa pun dapat dimulai. Namun, seringkali klien tidak tahu apa yang sebenarnya mereka inginkan sampai mereka melihat sistem yang bekerja. Sekarang telah diterima dengan baik di antara para praktisi bahwa hanya melalui proses mengomentari aplikasi yang diinstal, persyaratan yang tepat dapat dibawa keluar. Tetapi dalam model air terjun berulang, pelanggan tidak dapat melihat perangkat lunak, sampai pengembangan selesai dalam segala hal dan perangkat lunak telah dikirimkan dan diinstal. Secara alami, perangkat lunak yang dikirimkan seringkali tidak memenuhi harapan pelanggan dan banyak permintaan perubahan dibuat oleh pelanggan. Perubahan dimasukkan melalui upaya pemeliharaan berikutnya. Hal ini membuat biaya untuk mengakomodasi perubahan menjadi sangat tinggi dan biasanya membutuhkan waktu lama untuk memiliki solusi yang baik yang dapat memenuhi kebutuhan pelanggan. Model RAD mencoba mengatasi masalah ini dengan mengundang dan memasukkan umpan balik pelanggan pada pengembangan dan prototipe.

Bekerja di RAD

Dalam model RAD, pengembangan berlangsung dalam serangkaian siklus pendek atau iterasi. Setiap saat, tim pengembangan hanya berfokus pada iterasi saat ini, dan oleh karena itu rencana dibuat untuk satu peningkatan pada satu waktu. Waktu yang direncanakan untuk setiap iterasi disebut kotak waktu. Setiap iterasi direncanakan untuk meningkatkan fungsionalitas aplikasi yang diimplementasikan hanya dalam jumlah kecil. Selama setiap kotak waktu, perangkat lunak bergaya prototipe cepat dan kotor untuk beberapa fungsi dikembangkan. Pelanggan mengevaluasi prototipe dan memberikan umpan balik tentang perbaikan spesifik yang mungkin diperlukan. Prototipe disempurnakan berdasarkan umpan balik pelanggan. Harap dicatat bahwa prototipe tidak dimaksudkan untuk dirilis ke pelanggan untuk penggunaan biasa.

Tim pengembangan hampir selalu menyertakan perwakilan pelanggan untuk mengklarifikasi persyaratan. Hal ini dimaksudkan untuk membuat sistem disesuaikan dengan kebutuhan pelanggan yang tepat dan juga untuk menjembatani kesenjangan komunikasi antara pelanggan dan tim pengembangan. Tim pengembangan biasanya terdiri dari sekitar lima hingga enam anggota, termasuk perwakilan pelanggan.

Bagaimana RAD memfasilitasi akomodasi permintaan perubahan?

Pelanggan biasanya menyarankan perubahan pada fitur tertentu hanya setelah mereka menggunakannya. Karena fitur dikirimkan sedikit demi sedikit, pelanggan dapat memberikan permintaan perubahan terkait fitur yang sudah dikirimkan. Penggabungan permintaan perubahan tersebut tepat setelah pengiriman fitur tambahan menghemat biaya karena hal ini dilakukan sebelum investasi besar dilakukan dalam pengembangan dan pengujian sejumlah besar fitur.

Bagaimana RAD memfasilitasi pengembangan yang lebih cepat?

Penurunan waktu dan biaya pengembangan, dan pada saat yang sama peningkatan fleksibilitas untuk memasukkan perubahan dicapai dalam model RAD dalam dua cara utama—penggunaan perencanaan yang minimal dan penggunaan ulang yang berat dari setiap kode yang ada melalui pembuatan prototipe cepat. Kurangnya perencanaan jangka panjang dan rinci memberikan fleksibilitas untuk mengakomodasi perubahan kebutuhan di kemudian hari. Penggunaan kembali kode yang ada telah diadopsi sebagai mekanisme penting untuk mengurangi biaya pengembangan.

Model RAD menekankan penggunaan kembali kode sebagai sarana penting untuk menyelesaikan proyek lebih cepat. Faktanya, pengadopsi model RAD adalah yang paling awal merangkul bahasa dan praktik berorientasi objek. Selanjutnya, RAD menganjurkan penggunaan alat khusus untuk memfasilitasi pembuatan prototipe kerja dengan cepat. Alat khusus ini biasanya mendukung fitur berikut:

- Gaya visual perkembangan.
- Penggunaan komponen yang dapat digunakan kembali.

Penerapan Model RAD

Berikut ini adalah beberapa karakteristik aplikasi yang menunjukkan kesesuaiannya dengan gaya pengembangan RAD:

- **Perangkat lunak yang disesuaikan:** Seperti yang telah ditunjukkan, perangkat lunak yang disesuaikan dikembangkan untuk satu atau dua pelanggan hanya dengan mengadaptasi perangkat lunak yang ada. Dalam proyek pengembangan perangkat lunak yang disesuaikan, penggunaan kembali yang substansial biasanya dibuat dari kode dari perangkat lunak yang sudah ada sebelumnya. Misalnya, sebuah perusahaan mungkin telah mengembangkan perangkat lunak untuk mengotomatisasi kegiatan pemrosesan data di satu atau lebih lembaga pendidikan. Ketika lembaga lain meminta paket otomasi untuk dikembangkan, biasanya hanya beberapa aspek yang perlu disesuaikan—karena di antara lembaga pendidikan yang berbeda, sebagian besar kegiatan pemrosesan data seperti pendaftaran siswa, penilaian, pengumpulan biaya, manajemen perkebunan, akuntansi, pemeliharaan catatan layanan staf dll mirip untuk sebagian besar. Proyek yang melibatkan penjahitan semacam itu dapat dilakukan dengan cepat dan hemat biaya menggunakan model RAD.
- **Perangkat lunak non-kritis**: Model RAD menyarankan bahwa perangkat lunak cepat dan kotor pertama-tama harus dikembangkan dan kemudian ini harus disempurnakan menjadi perangkat lunak akhir untuk pengiriman. Oleh karena itu, produk yang dikembangkan biasanya jauh dari kinerja dan keandalan yang optimal. Dalam hal ini, untuk proyek pengembangan yang dipahami dengan baik dan di mana ruang lingkup penggunaan kembali agak terbatas, model air terjun berulang dapat memberikan solusi yang lebih baik.
- **Jadwal proyek yang sangat terbatas**: RAD bertujuan untuk mengurangi waktu pengembangan dengan mengorbankan dokumentasi, kinerja, dan keandalan yang baik. Secara alami, untuk proyek dengan jadwal waktu yang sangat agresif, model RAD harus lebih disukai.
- **Perangkat lunak besar:** Hanya untuk perangkat lunak yang mendukung banyak fitur (perangkat lunak besar), pengembangan dan pengiriman tambahan dapat dilakukan secara bermakna.

Karakteristik aplikasi yang membuat RAD tidak sesuai

Gaya pengembangan RAD tidak disarankan jika proyek pengembangan memiliki satu atau lebih karakteristik berikut:

- Produk generik (distribusi luas): Produk perangkat lunak bersifat generik dan biasanya memiliki distribusi yang luas. Untuk sistem seperti itu, kinerja dan keandalan yang optimal sangat penting dalam pasar yang kompetitif. Seperti yang telah dibahas, model pengembangan RAD mungkin tidak menghasilkan sistem yang memiliki kinerja dan keandalan yang optimal.
- Persyaratan kinerja dan/atau keandalan yang optimal: Untuk kategori produk tertentu, diperlukan kinerja atau keandalan yang optimal. Contoh sistem tersebut termasuk sistem operasi (diperlukan keandalan tinggi) dan perangkat lunak simulator penerbangan (diperlukan kinerja tinggi). Jika sistem tersebut akan dikembangkan dengan menggunakan model RAD, kinerja dan keandalan produk yang diinginkan mungkin tidak dapat diwujudkan.
- Kurangnya produk serupa: Jika sebuah perusahaan belum mengembangkan perangkat lunak serupa, maka hampir tidak akan dapat menggunakan kembali banyak artefak yang ada. Dengan tidak adanya komponen plug-in yang memadai, menjadi sulit untuk mengembangkan prototipe cepat melalui penggunaan kembali, dan penggunaan model RAD menjadi tidak berarti.
- Entitas monolitik: Untuk perangkat lunak tertentu, terutama perangkat lunak berukuran kecil, mungkin sulit untuk membagi fitur yang diperlukan menjadi bagian-bagian yang dapat dikembangkan dan dikirimkan secara bertahap. Dalam hal ini, menjadi sulit untuk mengembangkan perangkat lunak secara bertahap.

Perbandingan RAD dengan Model Lain

Pada bagian ini, kita akan membandingkan keuntungan dan kerugian relatif dari RAD dengan model siklus hidup lainnya.

RAD versus model prototipe

Dalam model prototyping, prototipe yang dikembangkan terutama digunakan oleh tim pengembangan untuk mendapatkan wawasan tentang masalah, memilih antara alternatif, dan memperoleh umpan balik pelanggan. Kode yang dikembangkan selama konstruksi prototipe biasanya dibuang. Sebaliknya, di RAD itu adalah prototipe yang dikembangkan yang berkembang menjadi perangkat lunak yang dapat dikirimkan. Meskipun RAD diharapkan mengarah pada pengembangan perangkat lunak yang lebih cepat dibandingkan dengan model tradisional (seperti model prototyping), meskipun kualitas dan keandalannya akan lebih rendah.

RAD versus model air terjun berulang

Dalam model air terjun iteratif, semua fungsi perangkat lunak dikembangkan bersama. Di sisi lain, dalam model RAD, fungsionalitas produk dikembangkan secara bertahap melalui penggunaan ulang kode dan desain yang berat. Selanjutnya, dalam model RAD, umpan balik pelanggan diperoleh pada prototipe yang dikembangkan setelah setiap iterasi dan berdasarkan ini prototipe disempurnakan. Dengan demikian, menjadi mudah untuk mengakomodasi setiap permintaan untuk perubahan persyaratan. Namun, model air terjun berulang tidak mendukung mekanisme apa pun untuk mengakomodasi permintaan perubahan persyaratan apa pun. Model air terjun iteratif memang memiliki beberapa keunggulan penting yang antara lain sebagai berikut. Penggunaan model air terjun berulang mengarah pada produksi dokumentasi berkualitas baik yang dapat membantu selama pemeliharaan perangkat lunak. Selain itu, perangkat lunak yang dikembangkan biasanya memiliki kualitas dan keandalan yang lebih baik daripada yang dikembangkan menggunakan RAD.

RAD versus model evolusioner

Perkembangan bertahap adalah ciri khas model evolusioner dan RAD. Namun, dalam RAD setiap kenaikan pada dasarnya menghasilkan prototipe yang cepat dan kotor, sedangkan dalam model evolusioner setiap kenaikan dikembangkan secara sistematis menggunakan model air terjun iteratif. Juga dalam model RAD, perangkat lunak dikembangkan dalam peningkatan yang jauh lebih pendek dibandingkan model evolusioner. Dengan kata lain, fungsionalitas tambahan yang dikembangkan memiliki granularitas yang cukup besar dalam model evolusi.

2.4 MODEL PENGEMBANGAN CEPAT

Seperti yang telah ditunjukkan, meskipun model air terjun berulang telah sangat populer selama tahun 1970-an dan 1980-an, developer menghadapi beberapa masalah saat menggunakannya pada proyek perangkat lunak saat ini. Kesulitan utama termasuk menangani permintaan perubahan dari pelanggan selama pengembangan produk, dan biaya dan waktu yang terlalu tinggi yang dikeluarkan saat mengembangkan aplikasi yang disesuaikan. Capers Jones melakukan penelitian yang melibatkan 800 proyek pengembangan perangkat lunak kehidupan nyata, dan menyimpulkan bahwa rata-rata 40 persen dari persyaratan telah tiba setelah pengembangan dimulai. Dalam konteks ini, selama dua dekade terakhir ini, beberapa model siklus hidup telah diusulkan untuk mengatasi kekurangan penting dari model berbasis air terjun yang menjadi mencolok ketika digunakan dalam proyek pengembangan perangkat lunak modern.

Selama sekitar dua dekade terakhir, proyek yang menggunakan model siklus hidup berbasis air terjun berulang menjadi langka karena perubahan cepat dalam karakteristik proyek pengembangan perangkat lunak dari waktu ke waktu. Dua perubahan yang menjadi nyata adalah pergeseran cepat dari pengembangan produk perangkat lunak ke pengembangan perangkat lunak yang disesuaikan dan peningkatan penekanan dan ruang lingkup untuk digunakan kembali. Berikut ini, beberapa alasan mengapa pengembangan berbasis air terjun menjadi sulit untuk digunakan dalam proyek belakangan ini:

- Dalam model pengembangan perangkat lunak berbasis air terjun iteratif tradisional, persyaratan untuk sistem ditentukan pada awal proyek pengembangan dan diasumsikan telah diperbaiki sejak saat itu. Perubahan persyaratan selanjutnya setelah dokumen SRS diselesaikan tidak disarankan. Jika perubahan persyaratan di kemudian hari menjadi tidak dapat dihindari, maka biaya untuk mengakomodasinya menjadi sangat tinggi. Di sisi lain, akumulasi pengalaman menunjukkan bahwa pelanggan sering mengubah persyaratan mereka selama periode pengembangan karena berbagai alasan.
- Seperti yang ditunjukkan dalam Bab 1, selama dua dekade terakhir ini, aplikasi (layanan) yang disesuaikan telah menjadi tempat umum dan pendapatan penjualan yang dihasilkan di seluruh dunia dari layanan sudah melebihi produk perangkat lunak. Jelas, model air terjun berulang tidak cocok untuk pengembangan perangkat lunak semacam itu. Karena kustomisasi pada dasarnya melibatkan penggunaan kembali sebagian besar bagian dari aplikasi yang ada dan hanya terdiri dari melakukan modifikasi kecil dengan menulis kode dalam jumlah minimal. Untuk proyek pengembangan seperti itu, kebutuhan akan model pengembangan yang lebih tepat sangat dirasakan, dan banyak peneliti mulai menyelidiki ke arah ini.
- Model air terjun disebut model berat, karena terlalu banyak penekanan pada pembuatan dokumentasi dan penggunaan alat. Hal ini seringkali menjadi sumber

inefisiensi dan menyebabkan waktu penyelesaian proyek menjadi lebih lama dibandingkan dengan harapan pelanggan.

- Model air terjun mengatur hampir tidak ada interaksi pelanggan setelah persyaratan telah ditentukan. Faktanya, dalam model pengembangan perangkat lunak air terjun, interaksi pelanggan sebagian besar terbatas pada tahap inisiasi proyek dan penyelesaian proyek.

Model pengembangan perangkat lunak tangkas diusulkan pada pertengahan 1990-an untuk mengatasi kekurangan serius dari model pengembangan air terjun yang diidentifikasi di atas. Model tangkas terutama dirancang untuk membantu proyek beradaptasi dengan permintaan perubahan dengan cepat.1 Jadi, tujuan utama model tangkas adalah untuk memfasilitasi penyelesaian proyek dengan cepat. Tapi, bagaimana kelincahan dicapai dalam model ini? Kelincahan dicapai dengan menyesuaikan proses dengan proyek, yaitu menghilangkan aktivitas yang mungkin tidak diperlukan untuk proyek tertentu. Juga, apa pun yang membuang-buang waktu dan usaha dihindari.

Harap dicatat bahwa model tangkas digunakan sebagai istilah umum untuk merujuk pada sekelompok proses pengembangan. Proses-proses ini memiliki karakteristik umum tertentu, tetapi memiliki perbedaan halus tertentu di antara mereka sendiri. Beberapa model SDLC tangkas yang populer adalah sebagai berikut:

- Kristal
- Atern (sebelumnya DSDM)
- Pengembangan berbasis fitur
- Scrum
- Pemrograman ekstrim (XP)
- Pengembangan ramping
- Proses terpadu

Dalam model tangkas, persyaratan didekomposisi menjadi banyak bagian kecil yang dapat dikembangkan secara bertahap. Model tangkas mengadopsi pendekatan iteratif. Setiap bagian inkremental dikembangkan melalui iterasi. Setiap iterasi dimaksudkan untuk menjadi kecil dan mudah dikelola dan berlangsung selama beberapa minggu saja. Pada suatu waktu, hanya satu peningkatan yang direncanakan, dikembangkan, dan kemudian disebarkan di lokasi pelanggan. Tidak ada rencana jangka panjang yang dibuat. Waktu untuk menyelesaikan iterasi disebut kotak waktu. Implikasi dari istilah kotak waktu adalah bahwa tanggal akhir untuk iterasi tidak berubah. Artinya, tanggal pengiriman dianggap sakral. Namun, tim pengembangan dapat memutuskan untuk mengurangi fungsionalitas yang diberikan selama kotak waktu jika perlu. Prinsip utama dari model tangkas adalah pengiriman kenaikan ke pelanggan setelah setiap kotak waktu. Beberapa prinsip lain yang penting untuk model tangkas dibahas di bawah ini.

Ide Penting di Balik Model Agile

Untuk menjalin kontak dekat dengan pelanggan selama pengembangan dan untuk mendapatkan pemahaman yang jelas tentang masalah khusus domain, setiap proyek tangkas biasanya menyertakan perwakilan pelanggan dalam tim. Pada akhir setiap iterasi, pemangku kepentingan dan perwakilan pelanggan meninjau kemajuan yang dibuat dan mengevaluasi kembali persyaratan. Karakteristik yang membedakan model tangkas adalah seringnya pengiriman peningkatan perangkat lunak kepada pelanggan.

Model Agile menekankan komunikasi tatap muka atas dokumen tertulis. Disarankan agar ukuran tim pengembangan sengaja dibuat kecil (5-9 orang) untuk membantu anggota tim terlibat secara bermakna dalam komunikasi tatap muka dan memiliki lingkungan kerja yang kolaboratif. Maka tersirat bahwa model tangkas cocok untuk pengembangan proyek-proyek

kecil. Namun, jika proyek besar diperlukan untuk dikembangkan menggunakan model tangkas, kemungkinan tim yang berkolaborasi mungkin bekerja di lokasi yang berbeda. Dalam hal ini, tim yang berbeda diperlukan untuk mempertahankan kontak harian sebanyak mungkin melalui konferensi video, telepon, email, dll.

- Model tangkas menekankan rilis tambahan dari perangkat lunak yang berfungsi sebagai ukuran utama kemajuan. Prinsip-prinsip penting berikut di balik model tangkas dipublikasikan dalam manifesto tangkas pada tahun 2001:
- Perangkat lunak yang berfungsi melalui dokumentasi yang komprehensif.
- Pengiriman versi tambahan perangkat lunak secara berkala kepada pelanggan dalam interval beberapa minggu.
- Permintaan perubahan kebutuhan dari pelanggan didorong dan dimasukkan secara efisien.
- Memiliki anggota tim yang kompeten dan meningkatkan interaksi di antara mereka dianggap jauh lebih penting daripada masalah seperti penggunaan alat canggih atau kepatuhan yang ketat terhadap proses yang terdokumentasi. Disarankan bahwa peningkatan komunikasi di antara anggota tim pengembangan dapat diwujudkan melalui komunikasi tatap muka daripada melalui pertukaran dokumen formal.

Interaksi berkelanjutan dengan pelanggan dianggap jauh lebih penting daripada negosiasi kontrak yang efektif. Perwakilan pelanggan diperlukan untuk menjadi bagian dari tim pengembangan, sehingga memfasilitasi kerjasama harian yang erat antara pelanggan dan pengembang.

Proyek pengembangan tangkas biasanya menggunakan pemrograman berpasangan. Dalam pemrograman berpasangan, dua programmer bekerja sama di satu stasiun kerja. Satu mengetik kode sementara yang lain meninjau kode saat diketik. Dua programmer berganti peran setiap jam atau lebih. Beberapa penelitian menunjukkan bahwa programmer yang bekerja berpasangan menghasilkan program yang ditulis dengan baik dan melakukan lebih sedikit kesalahan dibandingkan dengan programmer yang bekerja sendiri.

Keuntungan dan kerugian dari metode tangkas

Metode tangkas memperoleh banyak kelincahan mereka dengan mengandalkan pengetahuan tacit dari anggota tim tentang proyek pengembangan dan komunikasi informal untuk mengklarifikasi masalah, daripada menghabiskan banyak waktu dalam menyiapkan dokumen formal dan meninjaunya. Meskipun ini menghilangkan beberapa overhead, tetapi kurangnya dokumentasi yang memadai dapat menyebabkan beberapa jenis masalah, yaitu sebagai berikut:

- Kurangnya dokumen formal menyisakan ruang untuk kebingungan dan keputusan penting yang diambil selama fase yang berbeda dapat disalahartikan di kemudian hari oleh anggota tim yang berbeda.
- Dengan tidak adanya dokumen formal, menjadi sulit untuk mendapatkan keputusan proyek penting seperti keputusan desain untuk ditinjau oleh ahli eksternal.
- Ketika proyek selesai dan developer bubar, pemeliharaan bisa menjadi masalah.

Agile versus Model Lain

Berikut ini adalah perbandingan karakteristik model tangkas dengan model pengembangan lainnya.

Model tangkas versus model air terjun berulang

Model air terjun sangat terstruktur dan langkah sistematis melalui persyaratan-capture, analisis, spesifikasi, desain, coding, dan tahap pengujian dalam urutan yang

direncanakan. Kemajuan umumnya diukur dalam hal jumlah artefak yang telah diselesaikan dan ditinjau seperti spesifikasi persyaratan, dokumen desain, rencana pengujian, tinjauan kode, dll. yang tinjauannya telah selesai. Sebaliknya, saat menggunakan model tangkas, kemajuan diukur dalam hal fungsionalitas yang dikembangkan dan disampaikan. Dalam model tangkas, pengiriman versi kerja perangkat lunak dibuat dalam beberapa peningkatan. Namun, dari segi kesamaan dapat dikatakan bahwa tim tangkas menggunakan model air terjun dalam skala kecil, mengulangi seluruh siklus air terjun di setiap iterasi.

Jika proyek yang sedang dikembangkan menggunakan model air terjun dibatalkan di tengah jalan selama pengembangan, maka tidak ada yang bisa ditunjukkan dari proyek yang ditinggalkan di luar beberapa dokumen. Dengan model tangkas, bahkan jika sebuah proyek dibatalkan di tengah jalan, itu masih meninggalkan pelanggan dengan beberapa kode yang berharga, yang mungkin telah dimasukkan ke dalam operasi langsung.

Pemrograman tangkas versus eksplorasi

Meskipun beberapa kesamaan memang ada antara gaya pemrograman tangkas dan eksploratif, ada perbedaan besar antara keduanya juga. Evaluasi ulang rencana model pengembangan tangkas, penekanan pada komunikasi tatap muka, dan penggunaan dokumentasi yang relatif jarang mirip dengan gaya eksplorasi. Tim yang gesit, bagaimanapun, mengikuti proses yang ditentukan dan disiplin dan melakukan penangkapan persyaratan sistematis, desain yang ketat, dibandingkan dengan pengkodean yang kacau dalam pemrograman eksplorasi.

Model tangkas versus model RAD

Perbedaan penting antara model gesit dan RAD adalah sebagai berikut:

- Model Agile tidak merekomendasikan pengembangan prototipe, tetapi menekankan pengembangan sistematis dari setiap fitur tambahan. Sebaliknya, tema sentral RAD didasarkan pada perancangan prototipe cepat dan kotor, yang kemudian disempurnakan menjadi kode kualitas produksi.
- Proyek tangkas secara logis memecah solusi menjadi fitur yang dikembangkan dan disampaikan secara bertahap. Pendekatan RAD tidak merekomendasikan hal ini. Sebaliknya, developer yang menggunakan model RAD berfokus pada pengembangan semua fitur aplikasi dengan terlebih dahulu melakukannya dengan buruk dan kemudian secara berturut-turut meningkatkan kode dari waktu ke waktu.
- Tim tangkas hanya mendemonstrasikan pekerjaan yang telah selesai kepada pelanggan. Sebaliknya, tim RAD mendemonstrasikan kepada pelanggan layar mock up, dan prototipe, yang mungkin didasarkan pada penyederhanaan seperti pencarian tabel daripada perhitungan.

Model Pemrograman Ekstrim (*Extreme Programming)*

Extreme programming (XP) adalah model proses penting di bawah payung tangkas dan diusulkan oleh Kent Beck pada tahun 1999. Nama model ini mencerminkan fakta bahwa model ini merekomendasikan untuk mengambil praktik terbaik yang telah bekerja dengan baik di masa lalu dalam pengembangan program. proyek ke tingkat ekstrim. Model ini didasarkan pada filosofi yang agak sederhana: "Jika sesuatu diketahui bermanfaat, mengapa tidak menggunakannya terus-menerus?" Berdasarkan prinsip ini, ia mengajukan beberapa praktik utama yang perlu dipraktikkan secara ekstrem. Harap dicatat bahwa sebagian besar praktik utama yang ditekankan telah diakui sebagai praktik yang baik untuk beberapa waktu.

Praktik baik yang perlu dipraktikkan secara ekstrem

Beberapa praktik yang diakui dalam model pemrograman ekstrem dan cara yang disarankan untuk memaksimalkan penggunaannya:

Tinjauan kode: Ini bagus karena membantu mendeteksi dan memperbaiki masalah dengan paling efisien. Ini menyarankan pemrograman berpasangan sebagai cara untuk mencapai tinjauan berkelanjutan. Dalam pair programming, pengkodean dilakukan oleh pasangan programmer. Programmer bergiliran menulis program dan sementara yang satu menulis kode ulasan lainnya yang sedang ditulis.

Pengujian: Kode pengujian membantu menghilangkan bug dan meningkatkan keandalannya. XP menyarankan pengembangan berbasis uji (TDD) untuk terus menulis dan menjalankan kasus uji. Dalam pendekatan TDD, kasus uji ditulis bahkan sebelum kode apa pun ditulis.

Pengembangan inkremental: Pengembangan inkremental bagus, karena membantu mendapatkan umpan balik pelanggan, dan tingkat fitur yang disampaikan merupakan indikator kemajuan yang dapat diandalkan. Ini menunjukkan bahwa tim harus membuat peningkatan baru setiap beberapa hari.

Kesederhanaan: Kesederhanaan membuatnya lebih mudah untuk mengembangkan kode berkualitas baik, serta untuk menguji dan men-debugnya. Oleh karena itu, seseorang harus mencoba membuat kode paling sederhana yang membuat fungsionalitas dasar yang ditulis berfungsi. Untuk membuat kode yang paling sederhana, seseorang dapat mengabaikan aspek-aspek seperti efisiensi, keandalan, pemeliharaan, dll. Setelah hal yang paling sederhana berhasil, aspek lain dapat diperkenalkan melalui refactoring.

Desain: Karena memiliki desain berkualitas baik penting untuk mengembangkan solusi berkualitas baik, setiap orang harus mendesain setiap hari. Ini dapat dicapai melalui refactoring, di mana kode kerja ditingkatkan untuk efisiensi dan pemeliharaan.

Pengujian integrasi: Ini penting karena membantu mengidentifikasi bug pada antarmuka fungsi yang berbeda. Untuk tujuan ini, pemrograman ekstrim menyarankan bahwa developer harus mencapai integrasi berkelanjutan, dengan membangun dan melakukan pengujian integrasi beberapa kali sehari.

Ide dasar model pemrograman ekstrim

XP didasarkan pada rilis yang sering (disebut iterasi), di mana developer menerapkan "cerita pengguna". Cerita pengguna mirip dengan kasus penggunaan, tetapi lebih informal dan lebih sederhana. Kisah pengguna adalah deskripsi percakapan oleh pengguna tentang fitur sistem yang diperlukan. Misalnya, cerita pengguna tentang perangkat lunak perpustakaan dapat berupa:

- Seorang anggota perpustakaan dapat menerbitkan buku.
- Seorang anggota perpustakaan dapat menanyakan tentang ketersediaan buku.
- Seorang anggota perpustakaan harus dapat mengembalikan buku yang dipinjam.

Kisah pengguna adalah pernyataan sederhana dari pelanggan tentang fungsionalitas yang dia butuhkan, tidak menyebutkan tentang detail yang lebih baik seperti skenario berbeda yang dapat terjadi, prasyarat (keadaan di mana sistem) harus dipenuhi sebelum fitur dapat dipanggil , dll.

Berdasarkan cerita pengguna, tim proyek mengusulkan "metafora"—sebuah visi bersama tentang bagaimana sistem akan bekerja. Tim pengembangan dapat memutuskan untuk membuat lonjakan untuk beberapa fitur. Sebuah spike, adalah program yang sangat sederhana yang dibangun untuk mengeksplorasi kesesuaian solusi yang diusulkan. Lonjakan dapat dianggap mirip dengan prototipe. XP mengatur beberapa aktivitas dasar untuk menjadi bagian dari proses pengembangan perangkat lunak. Kegiatan ini adalah sebagai berikut:

Coding: XP berpendapat bahwa kode adalah bagian penting dari setiap proses pengembangan sistem, karena tanpa kode tidak mungkin memiliki sistem yang berfungsi. Oleh

karena itu, perhatian dan perhatian penuh perlu ditempatkan pada aktivitas pengkodean. Namun, konsep kode seperti yang digunakan di XP memiliki arti yang sedikit berbeda dari apa yang dipahami secara tradisional. Misalnya, kegiatan coding meliputi menggambar diagram (modelling) yang akan diubah menjadi kode, membuat script sistem berbasis web, dan memilih di antara beberapa alternatif solusi.

Pengujian: XP sangat mementingkan pengujian dan menganggapnya sebagai sarana utama untuk mengembangkan perangkat lunak bebas kesalahan.

Mendengarkan: developer perlu mendengarkan pelanggan dengan cermat jika mereka harus mengembangkan perangkat lunak berkualitas baik. Programmer mungkin belum tentu memiliki pengetahuan mendalam tentang domain spesifik dari sistem yang sedang dikembangkan. Di sisi lain, pelanggan biasanya memiliki pengetahuan domain ini. Oleh karena itu, agar programmer dapat memahami dengan baik apa fungsi sistem yang seharusnya, mereka harus mendengarkan pelanggan.

Merancang: Tanpa desain yang tepat, implementasi sistem menjadi terlalu kompleks dan ketergantungan dalam sistem menjadi terlalu banyak dan menjadi sangat sulit untuk memahami solusinya, dan dengan demikian membuat biaya pemeliharaan menjadi sangat mahal. Sebuah desain yang baik harus menghasilkan penghapusan ketergantungan yang kompleks dalam suatu sistem. Dengan demikian, penggunaan yang efektif dari teknik desain yang sesuai ditekankan.

Umpan balik: Ini mendukung kebijaksanaan: "Sebuah sistem yang menjauh dari pengguna adalah masalah yang menunggu untuk terjadi". Ini mengakui pentingnya umpan balik pengguna dalam memahami kebutuhan pelanggan yang tepat. Waktu yang berlalu antara pengembangan versi dan pengumpulan umpan balik di dalamnya sangat penting untuk belajar dan membuat perubahan.

Kesederhanaan: Sebuah landasan XP didasarkan pada prinsip: "bangun sesuatu yang sederhana yang akan berhasil hari ini, daripada mencoba membangun sesuatu yang akan memakan waktu namun mungkin tidak akan pernah digunakan". Ini pada dasarnya berarti bahwa perhatian harus difokuskan pada fitur spesifik yang segera dibutuhkan dan membuatnya berfungsi, daripada mencurahkan waktu dan energi untuk spekulasi tentang persyaratan di masa depan.

XP mendukung pembuatan solusi untuk masalah sesederhana mungkin. Sebaliknya, metode pengembangan sistem tradisional merekomendasikan perencanaan untuk penggunaan kembali dan ekstensibilitas kode dan desain di masa depan dengan mengorbankan kode yang lebih tinggi dan kompleksitas desain.

Penerapan model pemrograman ekstrim

Berikut ini adalah beberapa karakteristik proyek yang menunjukkan kelayakan suatu proyek untuk dikembangkan dengan menggunakan model extreme programming:

- **Proyek yang melibatkan teknologi baru atau proyek penelitian:** Dalam hal ini, persyaratan berubah dengan cepat dan masalah teknis yang tidak terduga perlu diselesaikan.
- **Proyek kecil:** Pemrograman ekstrem diusulkan dalam konteks tim kecil karena pertemuan tatap muka lebih mudah dicapai.

Karakteristik proyek tidak cocok untuk pengembangan menggunakan model tangkas

Berikut ini adalah beberapa karakteristik proyek yang menunjukkan ketidaksesuaian model pengembangan agile untuk digunakan dalam proyek pengembangan:

Persyaratan stabil: Model pengembangan konvensional lebih cocok untuk digunakan dalam proyek yang ditandai dengan persyaratan stabil. Untuk proyek-proyek seperti itu, diketahui bahwa hanya sedikit perubahan, jika sama sekali, akan terjadi. Oleh karena itu,

model proses seperti model air terjun berulang yang melibatkan pembuatan rencana jangka panjang selama inisiasi proyek dapat digunakan secara bermakna.

Sistem kritis misi atau kritis keselamatan: Dalam pengembangan sistem seperti itu, model SDLC tradisional biasanya lebih disukai untuk memastikan keandalan.

Model Scrum

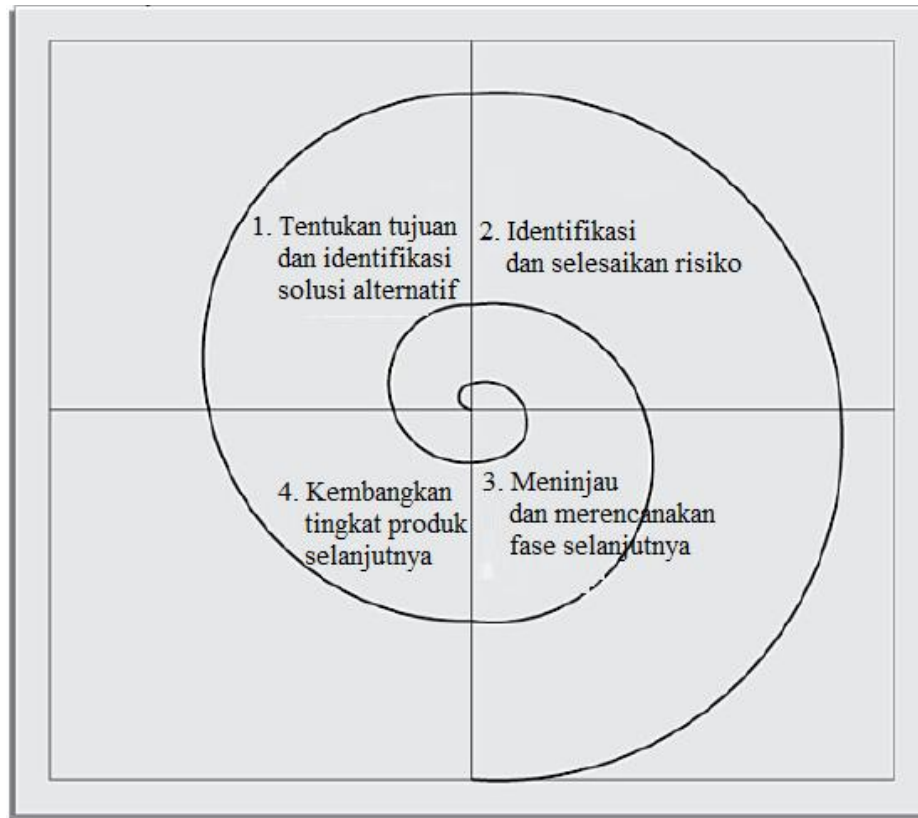
Dalam model scrum, sebuah proyek dibagi menjadi bagian-bagian kecil dari pekerjaan yang dapat dikembangkan secara bertahap dan dikirimkan dari waktu ke waktu disebut sprint. Oleh karena itu, perangkat lunak dikembangkan melalui serangkaian potongan yang dapat dikelola. Setiap sprint biasanya hanya membutuhkan waktu beberapa minggu untuk diselesaikan. Di akhir setiap sprint, pemangku kepentingan dan anggota tim bertemu untuk menilai kemajuan yang dibuat dan pemangku kepentingan menyarankan kepada tim pengembangan setiap perubahan yang diperlukan untuk fitur yang telah dikembangkan dan peningkatan keseluruhan yang mungkin mereka rasa perlu.

Dalam model scrum, anggota tim mengambil tiga peran mendasar — pemilik perangkat lunak, master scrum, dan anggota tim. Pemilik perangkat lunak bertanggung jawab untuk mengkomunikasikan visi pelanggan dari perangkat lunak kepada tim pengembangan. Scrum master bertindak sebagai penghubung antara pemilik perangkat lunak dan tim, sehingga memfasilitasi pekerjaan pengembangan.

2.5 MODEL SPIRAL

Model ini mendapatkan namanya dari tampilan representasi diagram yang terlihat seperti spiral dengan banyak loop (lihat Gambar 2.10). Jumlah pasti loop spiral tidak tetap dan dapat bervariasi dari satu proyek ke proyek lainnya. Jumlah loop yang ditunjukkan pada Gambar 2.10 hanyalah sebuah contoh. Setiap loop spiral disebut fase proses perangkat lunak. Jumlah pasti fase di mana produk dikembangkan dapat bervariasi oleh manajer proyek tergantung pada risiko proyek. Fitur menonjol dari model spiral adalah menangani risiko tak terduga yang dapat muncul jauh setelah proyek dimulai. Dalam konteks ini, harap diingat bahwa model prototyping dapat digunakan secara efektif hanya ketika risiko dalam suatu proyek dapat diidentifikasi di muka sebelum pekerjaan pengembangan dimulai. Model ini mencapai ini dengan memasukkan lebih banyak fleksibilitas dibandingkan dengan model SDLC lainnya.

Sementara model prototyping memberikan dukungan eksplisit untuk penanganan risiko, risiko diasumsikan telah diidentifikasi sepenuhnya sebelum proyek dimulai. Ini diperlukan karena prototipe dibangun hanya pada awal proyek. Sebaliknya, dalam model spiral, prototipe dibangun pada awal setiap fase. Setiap fase model direpresentasikan sebagai loop dalam representasi diagram. Pada setiap loop, satu atau lebih fitur produk diuraikan dan dianalisis dan risiko pada saat itu diidentifikasi dan diselesaikan melalui pembuatan prototipe. Berdasarkan ini, fitur yang diidentifikasi diimplementasikan.



Gambar 2.10 Model spiral pengembangan perangkat lunak.

Penanganan risiko dalam model spiral

Risiko pada dasarnya adalah setiap keadaan buruk yang mungkin menghambat keberhasilan penyelesaian proyek perangkat lunak. Sebagai contoh, pertimbangkan sebuah proyek yang risikonya adalah akses data dari basis data jarak jauh mungkin terlalu lambat untuk dapat diterima oleh pelanggan. Risiko ini dapat diatasi dengan membangun prototipe subsistem akses data dan bereksperimen dengan kecepatan akses yang tepat. Jika kecepatan akses data terlalu lambat, mungkin skema caching dapat diterapkan atau skema komunikasi yang lebih cepat dapat digunakan untuk mengatasi kecepatan akses data yang lambat. Resolusi risiko tersebut lebih mudah dilakukan dengan menggunakan prototipe karena pro dan kontra dari skema solusi alternatif dapat dievaluasi lebih cepat dan murah, dibandingkan dengan bereksperimen menggunakan aplikasi perangkat lunak yang sebenarnya sedang dikembangkan. Model spiral mendukung mengatasi risiko dengan menyediakan ruang lingkup untuk membangun prototipe di setiap fase pengembangan perangkat lunak.

Fase Model Spiral

Setiap fase dalam model ini dibagi menjadi empat sektor (atau kuadran) seperti yang ditunjukkan pada Gambar 2.10. Di kuadran pertama, beberapa fitur perangkat lunak diidentifikasi untuk diambil untuk pengembangan segera berdasarkan seberapa penting itu untuk pengembangan perangkat lunak secara keseluruhan. Dengan setiap iterasi di sekitar spiral (dimulai di tengah dan bergerak ke luar), versi perangkat lunak yang semakin lengkap dapat dibangun. Dengan kata lain, implementasi fitur-fitur yang teridentifikasi membentuk sebuah fase.

Kuadran 1: Tujuan diselidiki, diuraikan, dan dianalisis. Berdasarkan ini, risiko yang terlibat dalam memenuhi tujuan fase diidentifikasi. Di kuadran ini, solusi alternatif yang mungkin untuk fase yang dipertimbangkan diusulkan.

Kuadran 2: Selama kuadran kedua, solusi alternatif dievaluasi untuk memilih solusi terbaik. Untuk dapat melakukan ini, solusi dievaluasi dengan mengembangkan prototipe yang sesuai.

Kuadran 3: Kegiatan selama kuadran ketiga terdiri dari pengembangan dan verifikasi perangkat lunak tingkat berikutnya. Pada akhir kuadran ketiga, fitur yang diidentifikasi telah diimplementasikan dan versi perangkat lunak berikutnya tersedia.

Kuadran 4: Kegiatan selama kuadran keempat berkaitan dengan meninjau hasil tahapan yang dilalui sejauh ini (yaitu versi perangkat lunak yang dikembangkan) dengan pelanggan dan merencanakan iterasi spiral berikutnya.

Jari-jari spiral pada setiap titik mewakili biaya yang dikeluarkan dalam proyek sejauh ini, dan dimensi sudut mewakili kemajuan yang dibuat sejauh ini dalam fase saat ini. Dalam model spiral pengembangan, manajer proyek secara dinamis menentukan jumlah fase saat proyek berlangsung. Oleh karena itu, dalam model ini, manajer proyek memainkan peran penting untuk menyesuaikan model dengan proyek tertentu. Untuk membuat model lebih efisien, fitur yang berbeda dari perangkat lunak yang dapat dikembangkan secara bersamaan melalui siklus paralel diidentifikasi. Untuk menjaga diskusi kita tetap sederhana, kita tidak akan menyelidiki siklus paralel dalam model spiral.

Keuntungan/pro dan kerugian/kontra model spiral

Ada beberapa kelemahan model spiral yang membatasi penggunaannya hanya untuk beberapa jenis proyek. Untuk developer proyek, model spiral biasanya muncul sebagai model yang kompleks untuk diikuti, karena ini berisiko dan merupakan struktur fase yang lebih rumit daripada model lain yang kita diskusikan. Oleh karena itu akan menjadi kontraproduktif untuk digunakan, kecuali ada staf yang berpengetahuan dan berpengalaman dalam proyek tersebut. Juga, sangat tidak cocok untuk digunakan dalam pengembangan proyek outsourcing, karena risiko perangkat lunak perlu terus dinilai saat dikembangkan.

Terlepas dari kelemahan model spiral, untuk kategori proyek tertentu, kelebihan model spiral dapat melebihi kerugiannya. Untuk proyek yang memiliki banyak risiko yang tidak diketahui yang mungkin muncul saat pengembangan berlangsung, model spiral akan menjadi model pengembangan yang paling tepat untuk diikuti.

Dalam hal ini, ini jauh lebih kuat daripada model prototipe. Model prototipe dapat digunakan secara bermakna ketika semua risiko yang terkait dengan suatu proyek diketahui sebelumnya. Semua risiko ini diselesaikan dengan membangun prototipe sebelum pengembangan perangkat lunak yang sebenarnya dimulai.

Model spiral sebagai model meta

Dibandingkan dengan model yang dibahas sebelumnya, model spiral dapat dilihat sebagai model meta, karena mencakup semua model yang dibahas. Misalnya, spiral loop tunggal sebenarnya mewakili model air terjun. Model spiral menggunakan pendekatan model prototyping dengan terlebih dahulu membangun prototipe di setiap fase sebelum pengembangan yang sebenarnya dimulai. Prototipe ini digunakan sebagai mekanisme pengurangan risiko. Model spiral menggabungkan pendekatan bertahap yang sistematis dari model air terjun. Selain itu, model spiral dapat dianggap sebagai pendukung model evolusi—iterasi sepanjang spiral dapat dianggap sebagai tingkat evolusi yang melaluinya sistem lengkap dibangun. Hal ini memungkinkan developer untuk memahami dan menyelesaikan risiko pada setiap tingkat evolusi (yaitu iterasi sepanjang spiral).

Studi kasus 2.2

Galaxy Inc. melakukan pengembangan komunikasi berbasis satelit antar handset mobile yang dapat berada di mana saja di muka bumi. Berbeda dengan telepon seluler tradisional, dengan menggunakan telepon seluler berbasis satelit, panggilan dapat dilakukan

selama telepon sumber dan tujuan berada di area jangkauan beberapa stasiun pangkalan. Sistem ini akan berfungsi melalui sekitar enam lusin satelit yang mengorbit bumi. Satelit akan langsung mengambil sinyal dari handset dan mengirimkan sinyal ke handset tujuan. Karena jejak kaki satelit yang berputar akan menutupi seluruh bumi, komunikasi antara dua titik mana pun di bumi, bahkan antara tempat-tempat terpencil seperti yang ada di Samudra Arktik dan Antartika, juga akan dimungkinkan. Namun, risiko dalam proyek ini banyak, termasuk menentukan bagaimana panggilan di antara satelit dapat dialihkan ketika mereka sendiri berputar dengan kecepatan yang sangat tinggi. Dengan tidak adanya materi yang dipublikasikan dan ketersediaan staf yang berpengalaman dalam pengembangan produk serupa, banyak risiko yang tidak dapat diidentifikasi pada awal proyek dan kemungkinan akan muncul seiring berjalannya proyek. Perangkat lunak akan membutuhkan beberapa juta baris kode untuk ditulis. Galaxy Inc. memutuskan untuk menggunakan model spiral untuk pengembangan perangkat lunak setelah mempekerjakan staf yang berkualifikasi tinggi. Untuk mempercepat pengembangan perangkat lunak, bagian independen dari perangkat lunak dikembangkan melalui siklus paralel pada spiral. Biaya dan jadwal pengiriman disempurnakan berkali-kali, seiring berjalannya proyek. Proyek ini berhasil diselesaikan setelah lima tahun dari tanggal mulai.

2.6 PERBANDINGAN MODEL SIKLUS HIDUP YANG BERBEDA

Model air terjun klasik dapat dianggap sebagai model dasar dan semua model siklus hidup lainnya sebagai hiasan dari model ini. Namun, model air terjun klasik tidak dapat digunakan dalam proyek pengembangan praktis, karena model ini tidak mendukung mekanisme untuk memperbaiki kesalahan yang dilakukan selama fase mana pun tetapi terdeteksi pada fase selanjutnya. Masalah ini diatasi dengan model air terjun iteratif melalui penyediaan jalur umpan balik. Model air terjun berulang mungkin merupakan model pengembangan perangkat lunak yang paling banyak digunakan sejauh ini. Model ini mudah dipahami dan digunakan. Namun, model ini hanya cocok untuk masalah yang dipahami dengan baik, dan tidak cocok untuk pengembangan proyek yang sangat besar dan proyek yang menderita sejumlah besar risiko.

Model prototyping cocok untuk proyek yang persyaratan pengguna atau aspek teknis yang mendasarinya tidak dipahami dengan baik, namun semua risiko dapat diidentifikasi sebelum proyek dimulai. Model ini sangat populer untuk pengembangan bagian antarmuka pengguna proyek. Pendekatan evolusioner cocok untuk masalah besar yang dapat didekomposisi menjadi satu set modul untuk pengembangan dan pengiriman tambahan. Model ini juga digunakan secara luas untuk proyek pengembangan berorientasi objek. Tentu saja, model ini hanya dapat digunakan jika pengiriman tambahan dari sistem dapat diterima oleh pelanggan.

Model spiral dianggap sebagai model meta dan mencakup semua model siklus hidup lainnya. Fleksibilitas dan penanganan risiko secara inheren dibangun ke dalam model ini. Model spiral cocok untuk pengembangan perangkat lunak besar yang menantang secara teknis dan rentan terhadap beberapa jenis risiko yang sulit diantisipasi pada awal proyek. Namun, model ini jauh lebih kompleks daripada model lainnya—ini mungkin merupakan faktor yang menghalangi penggunaannya dalam proyek biasa.

Sekarang mari kita bandingkan model prototyping dengan model spiral. Model prototyping dapat digunakan jika risikonya sedikit dan dapat ditentukan pada awal proyek. Model spiral, di sisi lain, berguna ketika risiko sulit diantisipasi di awal proyek, tetapi kemungkinan akan muncul saat pembangunan berlangsung. Mari kita bandingkan model siklus hidup yang berbeda dari sudut pandang pelanggan. Awalnya, kepercayaan pelanggan

biasanya tinggi pada tim pengembangan terlepas dari model pengembangan yang diikuti. Selama proses pengembangan yang panjang, kepercayaan pelanggan biasanya menurun, karena belum ada perangkat lunak yang berfungsi yang terlihat. developer menjawab pertanyaan pelanggan menggunakan bahasa gaul teknis, dan penundaan diumumkan. Hal ini menimbulkan kekesalan pelanggan. Di sisi lain, pendekatan evolusioner memungkinkan pelanggan bereksperimen dengan perangkat lunak yang berfungsi jauh lebih awal daripada pendekatan monolitik. Keuntungan penting lainnya dari model inkremental adalah mengurangi trauma pelanggan karena terbiasa dengan sistem yang sama sekali baru. Pengenalan perangkat lunak secara bertahap melalui fase tambahan memberikan waktu kepada pelanggan untuk menyesuaikan diri dengan perangkat lunak baru. Selain itu, dari sudut pandang keuangan pelanggan, pengembangan inkremental tidak memerlukan pengeluaran modal awal yang besar. Pelanggan dapat memesan versi inkremental saat dan ketika dia mampu membelinya.

Memilih Model Siklus Hidup yang Tepat untuk Proyek

Kita sudahmempelajari tentang keuntungan dan kerugian dari berbagai model siklus hidup. Namun, bagaimana memilih model siklus hidup yang sesuai untuk proyek tertentu? Jawaban atas pertanyaan ini akan tergantung pada beberapa faktor. Model siklus hidup yang sesuai mungkin dapat dipilih berdasarkan analisis masalah seperti berikut:

Karakteristik perangkat lunak yang akan dikembangkan: Pilihan model siklus hidup sebagian besar tergantung pada sifat perangkat lunak yang sedang dikembangkan. Untuk proyek layanan kecil, model tangkas lebih disukai. Di sisi lain, untuk pengembangan produk dan perangkat lunak tertanam, model air terjun berulang dapat lebih disukai. Model evolusioner adalah model yang cocok untuk proyek pengembangan berorientasi objek.

Karakteristik tim pengembangan: Tingkat keterampilan anggota tim merupakan faktor penting dalam memutuskan model siklus hidup yang akan digunakan. Jika tim developer berpengalaman dalam mengembangkan perangkat lunak serupa, bahkan perangkat lunak tertanam dapat dikembangkan menggunakan model air terjun berulang. Jika tim developer sepenuhnya pemula, maka aplikasi pemrosesan data sederhana pun mungkin memerlukan model prototipe untuk diadopsi.

Karakteristik pelanggan: Jika pelanggan tidak cukup akrab dengan komputer, maka persyaratan cenderung sering berubah karena akan sulit untuk membentuk persyaratan yang lengkap, konsisten, dan tidak ambigu. Dengan demikian, model prototyping mungkin diperlukan untuk mengurangi permintaan perubahan di kemudian hari dari pelanggan.

2.7 RINGKASAN

- Selama pengembangan semua jenis perangkat lunak, kepatuhan terhadap model proses yang sesuai telah diterima secara universal oleh organisasi pengembangan perangkat lunak. Adopsi model siklus hidup yang sesuai sekarang diterima sebagai kebutuhan utama untuk berhasil menyelesaikan proyek.
- Organisasi pengembangan perangkat lunak yang baik akan sangat hati-hati mendokumentasikan model proses yang tepat yang mereka ikuti dan biasanya menyertakan yang berikut ini dalam dokumen:
    - Identifikasi fase yang berbeda.
    - Identifikasi aktivitas yang berbeda di setiap fase dan urutan pelaksanaannya.
    - Kriteria fase masuk dan keluar untuk fase yang berbeda.
    - Metodologi yang diikuti untuk melakukan aktivitas yang berbeda.

- Kepatuhan pada model siklus hidup software mendorong anggota tim untuk melakukan berbagai kegiatan pengembangan secara sistematis dan disiplin. Itu juga membuat manajemen proyek pengembangan perangkat lunak lebih mudah.
- Prinsip mendeteksi kesalahan sedekat mungkin dengan titik awal dikenal sebagai fase penahanan kesalahan. Penahanan fase meminimalkan biaya untuk memperbaiki kesalahan.
- Model air terjun klasik dapat dianggap sebagai model dasar dan semua model siklus hidup lainnya adalah hiasan dari model ini. Model air terjun iteratif telah menjadi model siklus hidup yang paling banyak digunakan sejauh ini, meskipun penggunaan model RAD dan agile telah meningkat.
- Model siklus hidup yang berbeda memiliki kelebihan dan kekurangannya sendiri. Oleh karena itu, model siklus hidup yang sesuai harus dipilih untuk masalah yang dihadapi. Setelah memilih model siklus hidup dasar, organisasi pengembangan perangkat lunak biasanya menyesuaikan model siklus hidup standar sesuai dengan kebutuhan mereka.
- Meskipun suatu organisasi dapat mengikuti model siklus hidup mana pun yang sesuai untuk suatu proyek, dokumen akhir harus mencerminkan seolah-olah perangkat lunak dikembangkan menggunakan model air terjun klasik. Ini memudahkan pengelola untuk memahami dokumen perangkat lunak.

2.8 LATIHAN

1. Pilihlah opsi yang benar untuk setiap pertanyaan berikut:
   a. Manakah dari kerugian berikut yang mungkin dialami ketika model proses pengembangan sistematis diadopsi dalam preferensi daripada gaya pengembangan *build-and-fix*?
      i. Peningkatan biaya dokumentasi
      ii. Peningkatan biaya pengembangan
      iii. Menurunnya kemampuan pemeliharaan
      iv. Peningkatan waktu pengembangan
   b. Model proses perangkat lunak mewakili salah satu dari berikut ini:
      i. Cara perangkat lunak dikembangkan
      ii. Cara perangkat lunak memproses data
      iii. Cara perangkat lunak digunakan
      iv. Cara perangkat lunak mungkin gagal
   c. Dalam model air terjun SDLC, pengujian unit dilakukan selama salah satu fase berikut?
      i. Pengkodean
      ii. Pengujian
      iii. Desain
      iv. Pemeliharaan
   d. Manakah dari aktivitas berikut yang mencakup semua tahap siklus hidup pengembangan perangkat lunak (SDLC)?
      i. Pengkodean
      ii. Pengujian
      iii. Manajemen proyek
      iv. Desain
   e. Fase operasi dalam model air terjun adalah sinonim untuk salah satu fase berikut:
      i. Fase pengkodean dan pengujian unit
      ii. Integrasi dan fase pengujian sistem

iii. Fase pemeliharaan
iv. Fase desain

f. Fase implementasi dalam model air terjun adalah sinonim untuk salah satu fase berikut:
   i. Fase pengkodean dan pengujian unit
   ii. Integrasi dan fase pengujian sistem
   iii. Fase pemeliharaan
   iv. Fase desain
g. Unit testing dilakukan pada fase mana dari model waterfall:
   i. Tahap implementasi
   ii. Fase pengujian
   iii. Fase pemeliharaan
   iv. Fase desain
h. Manakah dari fase berikut yang menunjukkan upaya maksimal selama pengembangan perangkat lunak biasa?
   i. Pengkodean
   ii. Pengujian
   iii. Merancang
   iv. Spesifikasi
i. Manakah dari berikut ini yang bukan model proses pengembangan perangkat lunak standar?
   i. Model Air Terjun
   ii. Model Daerah Aliran Sungai
   iii. Model RAD
   iv. Model-V
j. Manakah dari jalur umpan balik berikut yang tidak ada dalam model air terjun berulang?
   i. Fase desain ke fase studi kelayakan
   ii. Fase implementasi hingga fase desain
   iii. Fase implementasi hingga fase spesifikasi kebutuhan
   iv. Fase desain ke fase spesifikasi kebutuhan
k. Manakah dari berikut ini yang merupakan model SDLC yang cocok untuk mengembangkan perangkat lunak berukuran sedang yang pelanggannya tidak jelas tentang persyaratannya yang sebenarnya?
   i. model RAD
   ii. model-V
   iii. Model air terjun berulang
   iv. Model air terjun klasik
   v. Model-V
l. Manakah dari jalur umpan balik berikut yang tidak ada dalam model air terjun berulang?
   i. Fase desain ke fase studi kelayakan
   ii. Fase implementasi hingga fase desain
   iii. Fase implementasi hingga fase spesifikasi kebutuhan
   iv. Fase desain ke fase spesifikasi kebutuhan
m. Manakah dari berikut ini yang merupakan model SDLC yang cocok untuk mengembangkan perangkat lunak berukuran sedang yang pelanggannya tidak jelas tentang persyaratannya yang sebenarnya?

i. Model RAD
ii. Model-V
iii. Model air terjun berulang
iv. Model air terjun klasik

n. Manakah dari model SDLC berikut yang cocok untuk digunakan dalam proyek yang melibatkan kustomisasi paket komunikasi komputer? Asumsikan bahwa proyek akan diawaki oleh personel yang berpengalaman. Jadwal proyek telah diatur dengan sangat agresif?
   i. Model spiral
   ii. Model air terjun berulang
   iii. Model RAD
   iv. Model tangkas
o. Manakah dari model siklus hidup berikut yang tidak memiliki karakteristik pengembangan perangkat lunak berulang?
   i. Model spiral
   ii. Model pembuatan prototipe
   iii. Model air terjun klasik
   iv. Model evolusioner
p. Manakah dari model siklus hidup berikut yang tidak melibatkan pembuatan prototipe setiap saat selama pengembangan perangkat lunak?
   i. Model spiral
   ii. Model pembuatan prototipe
   iii. model RAD
   iv. Model evolusioner
q. Bagian GUI dari perangkat lunak aplikasi biasanya dikembangkan menggunakan model siklus hidup yang mana?
   i. Model air terjun berulang
   ii. Model spiral
   iii. Model pembuatan prototipe
   iv. Model evolusioner
r. Manakah dari berikut ini yang bukan merupakan karakteristik dari model pengembangan perangkat lunak tangkas?
   i. Konstruksi prototipe
   ii. Perkembangan evolusioner
   iii. Pengembangan berulang
   iv. Pengiriman berkala perangkat lunak yang berfungsi
s. Manakah dari model SDLC berikut yang dapat dianggap lebih efektif untuk menentukan kebutuhan pelanggan yang tepat?
   i. Model air terjun berulang
   ii. model-V
   iii. Model pembuatan prototipe
   iv. Model air terjun klasik
t. Perubahan permintaan dari pelanggan kemudian dalam siklus pengembangan adalah paling mudah untuk menangani di mana dari model siklus hidup berikut?
   i. Model air terjun berulang
   ii. Model pembuatan prototipe
   iii. model-V
   iv. Model evolusioner

u. Asumsikan bahwa Anda adalah manajer proyek dari proyek pengembangan untuk aplikasi pemrosesan data di mana persyaratan pengguna untuk bagian GUI tidak terlalu jelas. Model siklus hidup mana yang akan Anda gunakan untuk mengembangkan bagian GUI?
   i. Model air terjun klasik
   ii. Model air terjun berulang
   iii. Model pembuatan prototipe
   iv. Model spiral
v. Dimensi sudut model spiral tidak mewakili yang mana dari berikut ini?
   i. Biaya yang dikeluarkan sejauh ini
   ii. Jumlah fitur yang diimplementasikan sejauh ini
   iii. Kemajuan dalam penerapan fitur saat ini
   iv. Jumlah risiko yang telah diselesaikan sejauh ini
w. Dimensi radial model spiral menunjukkan yang mana dari berikut ini?
   i. Biaya yang dikeluarkan sejauh ini
   ii. Jumlah fitur yang diimplementasikan sejauh ini
   iii. Kemajuan dalam penerapan fitur saat ini
   iv. Jumlah risiko yang telah diselesaikan sejauh ini

2. Apa yang Anda pahami dengan istilah siklus hidup perangkat lunak? Mengapa perlu memodelkan siklus hidup perangkat lunak dan mendokumentasikannya?
3. Apa yang Anda pahami dengan istilah model siklus hidup pengembangan perangkat lunak (SDLC)? Masalah apa yang mungkin dihadapi organisasi pengembangan perangkat lunak jika tidak mengikuti SDLC apa pun untuk pengembangan perangkat lunak berukuran besar?
4. Masalah apa yang akan dihadapi organisasi pengembangan perangkat lunak jika tidak memiliki model proses yang terdokumentasi, dan karena itu hanya mengikuti model informal?
5. Apakah istilah SDLC dan proses pengembangan perangkat lunak sama? Jelaskan jawaban Anda.
6. Mengapa penting bagi organisasi untuk mendokumentasikan proses pengembangannya dengan benar?
7. Sebutkan kegiatan utama yang dilakukan selama pengembangan suatu perangkat lunak perangkat lunak dan Sebutkan aktivitas yang mencakup semua fase pengembangan.
8. Apakah yang Anda pahami tentang: proses pengembangan perangkat lunak Apa perbedaan antara metodologi dan proses? Jelaskan jawaban Anda dengan menggunakan contoh yang sesuai.
9. Manakah fase utama dalam model pengembangan perangkat lunak air terjun? Fase mana yang menghabiskan upaya maksimal untuk mengembangkan perangkat lunak biasa?
10. Mengapa model air terjun klasik disebut model pengembangan idealis? Apakah model pengembangan ini memiliki kegunaan praktis sama sekali?
11. Perhatikan pernyataan berikut: *"Model air terjun klasik adalah model idealis".* Berdasarkan pernyataan tersebut, jawablah pertanyaan berikut:
    a. Berikan alasan mengapa pernyataan di atas benar.
    b. Bahkan jika model air terjun klasik adalah model idealis, apakah ada kegunaan praktis dari model ini sama sekali? Jelaskan jawaban Anda.

12. Apa perbedaan antara *programming-in-the-small* dan *programming-in-the-large*? Apakah menggunakan model air terjun SDLC ide yang bagus untuk pemrograman kecil?
13. Gambarlah diagram skematik untuk mewakili model air terjun iteratif dari pengembangan perangkat lunak. Pada diagram Anda, tunjukkan yang berikut:
    a. Kriteria masuk dan keluar fase untuk setiap fase.
    b. Hasil yang perlu diproduksi pada akhir setiap fase.
14. Apa tujuan dari fase studi kelayakan pengembangan perangkat lunak? Jelaskan kegiatan penting yang dilakukan selama fase studi kelayakan proyek pengembangan perangkat lunak. Siapa yang melaksanakan kegiatan-kegiatan tersebut? Sebutkan kriteria masuk dan keluar fase yang sesuai untuk fase ini.
15. Berikan contoh proyek pengembangan perangkat lunak yang model air terjun iteratifnya tidak cocok. Jelaskan secara singkat jawaban Anda.
16. Dalam proyek pengembangan perangkat lunak praktis menggunakan SDLC air terjun berulang, mengapa fase yang berbeda tumpang tindih? Jelaskan distribusi upaya selama fase yang berbeda.
17. Identifikasi lima alasan mengapa persyaratan pelanggan dapat berubah setelah fase persyaratan selesai dan dokumen SRS telah ditandatangani.
18. Identifikasi kriteria berdasarkan model siklus hidup yang cocok yang dapat dipilih untuk pengembangan proyek tertentu. Ilustrasikan jawaban Anda dengan menggunakan contoh-contoh yang sesuai.
19. Jelaskan secara singkat perbedaan dan persamaan penting antara model inkremental dan evolusioner SDLC.
20. Apa yang Anda pahami dengan gaya pengembangan perangkat lunak "membangun-dan-memperbaiki"? Secara diagramatis menggambarkan aktivitas khas dalam gaya perkembangan ini dan urutannya. Identifikasi setidaknya empat masalah utama yang akan muncul, jika proyek pengembangan perangkat lunak profesional yang besar dilakukan dengan menggunakan gaya pengembangan perangkat lunak “build-and-fix”.
21. Nyatakan apakah pernyataan berikut BENAR atau SALAH. Berikan alasan di balik jawaban Anda.
    a. Jika prinsip fase penahanan kesalahan tidak diikuti selama pengembangan perangkat lunak, maka biaya pengembangan akan meningkat.
    b. Model siklus hidup evolusioner akan sesuai untuk mengembangkan perangkat lunak yang tampaknya dipenuhi dengan sejumlah besar risiko.
    c. Jumlah fase dalam model siklus hidup spiral tidak tetap dan biasanya ditentukan oleh manajer proyek saat proyek berlangsung.
    d. Tujuan utama dari fase penahanan kesalahan adalah untuk mengembangkan perangkat lunak bebas kesalahan.
    e. Pengembangan perangkat lunak menggunakan model siklus hidup prototyping selalu lebih mahal daripada pengembangan perangkat lunak yang sama menggunakan model air terjun iteratif karena biaya tambahan yang dikeluarkan untuk membangun prototipe sekali pakai.
    f. Ketika perangkat lunak besar dikembangkan oleh rumah pengembangan perangkat lunak komersial menggunakan model air terjun berulang, tidak ada titik waktu yang tepat di mana transisi dari satu fase ke fase lain terjadi.
    g. Di antara semua fase pengembangan perangkat lunak, kesalahan yang tidak terdeteksi dari fase desain yang akhirnya terdeteksi selama pengujian penerimaan sistem menghabiskan biaya maksimum.

h. Jika tim yang mengembangkan produk perangkat lunak berukuran sedang tidak peduli dengan fase penahanan kesalahan, itu masih dapat menghasilkan perangkat lunak yang andal, meskipun dengan biaya lebih tinggi dibandingkan dengan kasus di mana ia mencoba fase penahanan kesalahan.
i. Dimensi sudut dalam model spiral pengembangan perangkat lunak menunjukkan total biaya yang dikeluarkan dalam proyek sampai saat itu.
j. Ketika model spiral digunakan dalam proyek pengembangan perangkat lunak, jumlah loop dalam spiral ditetapkan oleh manajer proyek selama tahap perencanaan proyek.
k. RAD akan menjadi model siklus hidup yang cocok untuk mengembangkan sistem operasi komersial.
l. RAD adalah model proses yang cocok digunakan untuk mengembangkan aplikasi kritis keselamatan seperti pengontrol untuk reaktor nuklir.

22. Apa yang Anda pahami dengan sindrom "99 persen selesai" yang terkadang dihadapi oleh manajer proyek perangkat lunak? Apa penyebab yang mendasarinya? Masalah apa yang dibuatnya untuk manajemen proyek? Apa saja obatnya?
23. Saat menggunakan model air terjun berulang untuk mengembangkan perangkat lunak komersial untuk aplikasi industri, diskusikan bagaimana upaya yang dihabiskan pada fase yang berbeda tersebar dari waktu ke waktu.
24. Model siklus hidup mana yang akan Anda ikuti untuk mengembangkan perangkat lunak untuk setiap aplikasi berikut? Sebutkan alasan di balik pilihan Anda atas model siklus hidup tertentu.
    a. Aplikasi pemrosesan data yang dipahami dengan baik.
    b. Perangkat lunak baru yang akan menghubungkan komputer melalui komunikasi satelit Asumsikan bahwa tim Anda tidak memiliki pengalaman sebelumnya dalam mengembangkan perangkat lunak komunikasi satelit.
    c. Perangkat lunak yang akan berfungsi sebagai pengontrol sistem switching telepon.
    d. Perangkat lunak otomatisasi perpustakaan baru yang akan menghubungkan berbagai perpustakaan di kota.
    e. Perangkat lunak yang sangat besar yang akan menyediakan, memantau, dan mengontrol komunikasi seluler di antara para pelanggannya menggunakan seperangkat satelit berputar.
    f. Sebuah editor teks baru.
    g. Kompilator untuk bahasa baru.
    h. Upaya pengembangan perangkat lunak berorientasi objek.
    i. Bagian antarmuka pengguna grafis dari perangkat lunak besar.
25. Jelaskan secara singkat model V SDLC dan jawab pertanyaan spesifik berikut yang berkaitan dengan V SDLC.
    a. Apa kekuatan dan kelemahan model-V?
    b. Uraikan persamaan dan perbedaan model-V dengan model air terjun iteratif.
    c. Berikan contoh proyek pengembangan yang model-V dapat dianggap sesuai dan berikan juga contoh proyek yang jelas tidak sesuai.
26. Jelaskan secara singkat model V-SDLC. Identifikasi mengapa untuk mengembangkan perangkat lunak kritis keselamatan, model V SDLC biasanya dianggap cocok.
27. Sehubungan dengan model prototyping untuk pengembangan perangkat lunak, jawablah sebagai berikut:

a. Apa itu prototipe?
b. Apakah perlu mengembangkan prototipe untuk semua jenis proyek?
c. Jika Anda menjawab bagian (b) dari pertanyaan tidak, sebutkan dalam keadaan apa menguntungkan untuk membangun prototipe.
d. Jika jawaban Anda untuk bagian (b) dari pertanyaan adalah ya, maka jelaskan apakah konstruksi prototipe selalu meningkatkan biaya pengembangan perangkat lunak secara keseluruhan.

28. Jika model prototyping digunakan dalam proyek pengembangan ukuran sedang, apakah perlu untuk mengembangkan dokumen SRS? Justifikasi jawaban Anda.
29. Pertimbangkan bahwa proyek pengembangan perangkat lunak yang dilanda banyak risiko. Namun, asumsikan bahwa tidak mungkin untuk mengantisipasi semua risiko dalam proyek pada awal proyek dan beberapa risiko hanya dapat diidentifikasi jauh setelah pembangunan berlangsung. Sebagai manajer proyek merekomendasikan penggunaan prototipe atau model spiral?
30. Apa keuntungan utama dari pertama membangun prototipe kerja sebelum mulai mengembangkan perangkat lunak yang sebenarnya? Apa kerugian dari pendekatan ini?
31. Jelaskan bagaimana upaya pengembangan perangkat lunak dimulai dan akhirnya diakhiri dalam model spiral.
32. Misalkan sebuah biro perjalanan membutuhkan perangkat lunak untuk mengotomatisasi kegiatan pembukuannya. Serangkaian aktivitas yang akan diotomatisasi agak sederhana dan saat ini dilakukan secara manual. Agen perjalanan telah mengindikasikan bahwa mereka tidak yakin tentang jenis antarmuka pengguna yang cocok untuk karyawan dan pelanggannya. Apakah pantas bagi tim pengembangan untuk menggunakan model spiral untuk mengembangkan perangkat lunak ini?
33. Jelaskan mengapa model siklus hidup spiral dianggap sebagai model meta.
34. Baik model prototipe maupun model spiral telah dirancang untuk menangani risiko. Identifikasi bagaimana tepatnya risiko ditangani di masing-masing. Bagaimana kedua model ini dapat dibandingkan sehubungan dengan kemampuan penanganan risikonya?
35. Jelaskan dengan contoh yang sesuai, jenis pengembangan perangkat lunak yang sesuai dengan model spiral. Apakah jumlah putaran spiral tetap untuk proyek pembangunan yang berbeda? Jika tidak, jelaskan bagaimana jumlah loop dalam spiral ditentukan.
36. Diskusikan manfaat relatif dari pengembangan prototipe sekali pakai untuk mengatasi risiko versus menyempurnakan prototipe yang dikembangkan menjadi perangkat lunak akhir.
37. Jawablah pertanyaan-pertanyaan berikut, dengan menggunakan satu kalimat untuk masing-masing pertanyaan:
    a. Bagaimana risiko yang terkait dengan proyek ditangani dalam model spiral pengembangan perangkat lunak?
    b. Jenis risiko mana yang lebih baik ditangani dengan menggunakan model spiral dibandingkan dengan model prototipe?
    c. Berikan contoh proyek di mana model spiral dapat digunakan secara bermakna.
38. Bandingkan keuntungan relatif menggunakan model air terjun iteratif dan model spiral pengembangan perangkat lunak untuk mengembangkan aplikasi MIS.

Jelaskan dengan bantuan masing-masing satu contoh yang sesuai, jenis proyek yang akan Anda gunakan model pengembangan perangkat lunak air terjun, dan jenis proyek yang akan Anda gunakan model spiralnya.

39. Secara singkat membahas model proses evolusi. Jelaskan dengan menggunakan contoh yang sesuai jenis proyek pengembangan perangkat lunak yang sesuai dengan model siklus hidup evolusioner. Bandingkan kelebihan dan kekurangan model ini dengan model air terjun iteratif.
40. Asumsikan bahwa perusahaan developer perangkat lunak sudah berpengalaman dalam mengembangkan perangkat lunak penggajian dan telah mengembangkan perangkat lunak serupa untuk beberapa pelanggan (organisasi). Asumsikan bahwa perusahaan developer perangkat lunak telah menerima permintaan dari pelanggan (organisasi) tertentu, yang masih menggunakan pemrosesan daftar gajinya secara manual. Untuk mengembangkan perangkat lunak penggajian untuk organisasi ini, model siklus hidup mana yang harus digunakan? Justifikasi jawaban Anda.
41. Jelaskan mengapa mungkin tidak bijaksana untuk menggunakan model spiral dalam pengembangan perangkat lunak besar.
42. Alih-alih memiliki satu kali pengujian perangkat lunak di akhir pengembangannya, mengapa tiga tingkat pengujian yang berbeda—pengujian unit, pengujian integrasi, dan pengujian sistem—diperlukan? Apa tujuan utama dari masing-masing tingkat pengujian yang berbeda ini?
43. Apa yang Anda pahami dengan istilah fase penahanan kesalahan? Mengapa fase penahanan kesalahan dianggap penting? Bagaimana fase penahanan kesalahan dicapai dalam proyek pengembangan perangkat lunak?
44. Terlepas dari model siklus hidup mana pun yang diikuti untuk mengembangkan perangkat lunak, mengapa dokumen akhir diperlukan untuk menggambarkan perangkat lunak seolah-olah dikembangkan menggunakan model air terjun klasik?
45. Apa kekurangan utama dari model air terjun iteratif Sebutkan model siklus hidup yang mengatasi kekurangan tertentu. Bagaimana cara mengatasi kekurangan pada model tersebut?
46. Untuk jenis proyek pengembangan apa model-V sesuai? Jelaskan secara singkat model-V dan tunjukkan kekuatan dan kelemahannya.
47. Identifikasi motivasi dan tujuan utama di balik pengembangan model RAD. Bagaimana model membantu mencapai tujuan yang diidentifikasi?
48. Jelaskan aspek-aspek berikut dari model SDLC pengembangan aplikasi cepat (RAD):
    a. Apa yang dimaksud dengan kotak waktu dalam model RAD?
    b. Bagaimana RAD memfasilitasi pengembangan yang lebih cepat?
    c. Identifikasi perbedaan utama antara model RAD dan model prototipe.
    d. Mengidentifikasi jenis proyek yang sesuai dengan model RAD dan jenis yang tidak sesuai.
49. Sarankan model siklus hidup yang sesuai untuk proyek perangkat lunak yang telah dilakukan organisasi Anda atas nama pelanggan tertentu yang tidak yakin dengan persyaratannya dan cenderung sering mengubah persyaratannya, karena proses bisnis pelanggan (organisasi) terlambat berubah dengan cepat. Berikan alasan di balik jawaban Anda.
50. Gambarlah diagram skematik berlabel untuk mewakili model spiral pengembangan perangkat lunak. Apakah jumlah lilitan spiral tetap? Jika jawaban Anda setuju, tuliskan jumlah loop yang dimiliki spiral. Jika jawaban Anda negatif, jelaskan bagaimana dan atas dasar apa jumlah putaran spiral dapat ditentukan.

51. Misalnya Anda adalah manajer proyek tim pengembangan yang menggunakan model air terjun berulang untuk mengembangkan perangkat lunak tertentu. Apakah Anda merekomendasikan bahwa tim pengembangan harus memulai fase pengembangan hanya setelah fase sebelumnya selesai sepenuhnya? Jelaskan jawaban Anda.
52. Identifikasi perbedaan utama antara SDLC iteratif dan evolusioner.
53. Jelaskan bagaimana karakteristik produk, tim pengembangan, dan pelanggan memengaruhi pemilihan SDLC yang sesuai untuk suatu proyek.
54. Sehubungan dengan model pengembangan aplikasi cepat (RAD), jawablah pertanyaan berikut:
    a. Jelaskan berbagai aktivitas siklus hidup yang dilakukan dalam model RAD.
    b. Bagaimana model RAD membantu mengakomodasi permintaan perubahan di akhir pengembangan.
    c. Bagaimana RAD membantu dalam pengembangan perangkat lunak yang lebih cepat.
    d. Berikan contoh dua proyek di mana RAD akan menjadi model yang cocok untuk pengembangan.
    e. Tunjukkan kelebihan dan kekurangan model RAD dibandingkan dengan (i) model pembuatan prototipe dan (ii) model evolusioner.
    f. Tunjukkan kelemahan model RAD dibandingkan dengan model air terjun iteratif.
    g. Mengidentifikasi karakteristik yang membuat proyek cocok dengan gaya pengembangan RAD.
    h. Mengidentifikasi karakteristik yang membuat proyek tidak sesuai dengan gaya pengembangan RAD.
55. Mengidentifikasi faktor-faktor penting yang mempengaruhi pilihan model SDLC yang sesuai untuk proyek pengembangan perangkat lunak.
56. Jelaskan fitur penting dari model pengembangan perangkat lunak tangkas.
    a. Bandingkan kelebihan dan kekurangan model tangkas dengan air terjun iteratif dan model pemrograman eksplorasi.
    b. Apakah model siklus hidup tangkas cocok untuk pengembangan perangkat lunak tertanam? Jelaskan secara singkat jawaban Anda.
57. Jelaskan secara singkat model pengembangan perangkat lunak tangkas. Berikan contoh proyek yang model tangkasnya cocok dan satu proyek proyek yang model tangkasnya tidak sesuai.
58. Jelaskan kesamaan dalam tujuan dan praktik model pengembangan perangkat lunak RAD, Agile, dan Extreme Programming (XP). Jelaskan juga perbedaan di antara ketiga model ini.
59. Diskusikan secara singkat model RAD. Identifikasi keunggulan utama model RAD dibandingkan dengan model air terjun berulang. Bagaimana model RAD mencapai pengembangan yang lebih cepat dibandingkan dengan model air terjun berulang?
60. Bandingkan keuntungan relatif dari RAD, air terjun berulang, dan model evolusioner dari pengembangan perangkat lunak.
61. Identifikasi dan jelaskan praktik terbaik penting yang telah dimasukkan dalam model pemrograman ekstrem.
62. Menggunakan satu atau dua kalimat menjelaskan kekurangan penting dari model klasik air terjun yang RAD, tangkas, dan model pemrograman ekstrim (XP) alamat pengembangan perangkat lunak.

63. Jelaskan secara singkat model SDLC pemrograman ekstrim (XP). Identifikasi prinsip-prinsip utama yang perlu dipraktikkan secara ekstrem di XP. Apa itu lonjakan di XP? Mengapa diperlukan?
64. Identifikasi bagaimana SDLC tangkas mencapai pengurangan waktu dan biaya pengembangan. Apakah ada jebakan untuk mencapai pengurangan biaya dan waktu dengan cara ini?
65. Misalkan proyek pengembangan telah dilakukan oleh perusahaan untuk menyesuaikan salah satu perangkat lunak yang ada atas nama pelanggan tertentu. Identifikasi dua keuntungan utama menggunakan model tangkas dibandingkan model air terjun berulang.
66. Model siklus hidup mana yang akan Anda rekomendasikan untuk mengembangkan perangkat lunak berorientasi objek? Jelaskan jawaban Anda.
67. Apa yang Anda pahami tentang pemrograman berpasangan? Apa kelebihannya dibandingkan pemrograman tradisional?
68. Secara grafis mewakili aktivitas yang dilakukan dalam gaya membangun dan memperbaiki khas pengembangan perangkat lunak dan menunjukkan urutan di antara aktivitas.
69. Menganalisis dan secara grafis mewakili model siklus hidup perangkat lunak sumber terbuka seperti Linux atau Apache.

BAB 3
MANAJEMEN PROYEK PERANGKAT LUNAK

Manajemen proyek yang efektif sangat penting untuk keberhasilan setiap proyek pengembangan perangkat lunak. Di masa lalu, beberapa proyek gagal bukan karena kekurangan profesional teknis yang kompeten atau karena kurangnya sumber daya, tetapi karena penggunaan praktik manajemen proyek yang salah. Oleh karena itu, penting untuk mempelajari teknik manajemen proyek perangkat lunak terbaru dengan cermat.

Manajemen proyek perangkat lunak adalah topik yang sangat luas. Bahkan, pengajaran semester penuh dapat dilakukan pada teknik yang efektif untuk manajemen proyek perangkat lunak. Namun, dalam bab ini, kita akan membatasi diri kita hanya pada beberapa masalah dasar. Mari kita pahami dulu apa sebenarnya tujuan utama dari manajemen proyek perangkat lunak. Tujuan utama dari manajemen proyek perangkat lunak adalah untuk memungkinkan sekelompok developer bekerja secara efektif menuju penyelesaian proyek yang berhasil.

Seperti yang dapat disimpulkan dari definisi di atas, manajemen proyek melibatkan penggunaan seperangkat teknik dan keterampilan untuk mengarahkan proyek menuju kesuksesan. Sebelum berfokus pada teknik manajemen proyek ini, mari kita cari tahu terlebih dahulu siapa yang harus bertanggung jawab untuk mengelola sebuah proyek. Manajer proyek biasanya adalah anggota tim yang berpengalaman yang pada dasarnya bekerja sebagai pemimpin administratif tim. Untuk proyek pengembangan perangkat lunak kecil, satu anggota tim memikul tanggung jawab untuk manajemen proyek dan manajemen teknis. Untuk proyek besar, anggota tim yang berbeda (selain manajer proyek) memikul tanggung jawab kepemimpinan teknis. Tanggung jawab pemimpin teknis termasuk menangani masalah seperti alat dan teknik mana yang digunakan dalam proyek, solusi tingkat tinggi untuk masalah tersebut, algoritme khusus yang akan digunakan, dll.

3.1 KOMPLEKSITAS MANAJEMEN PROYEK PERANGKAT LUNAK

Manajemen proyek perangkat lunak jauh lebih kompleks daripada manajemen banyak jenis proyek lainnya. Faktor utama yang berkontribusi terhadap kompleksitas pengelolaan proyek perangkat lunak, seperti yang diidentifikasi oleh [Brooks75], adalah sebagai berikut:

Invisibility: Perangkat lunak tetap tidak terlihat, sampai pengembangannya selesai dan beroperasi. Apa pun yang tidak terlihat, sulit diatur dan dikendalikan. Pertimbangkan proyek pembangunan rumah. Untuk proyek ini, manajer proyek dapat dengan mudah menilai kemajuan proyek melalui pemeriksaan visual dari bangunan yang sedang dibangun. Oleh karena itu, manajer dapat memantau dengan cermat kemajuan proyek, dan mengambil tindakan perbaikan setiap kali ia menemukan bahwa kemajuannya tidak sesuai rencana. Sebaliknya, menjadi sangat sulit bagi manajer proyek perangkat lunak untuk menilai kemajuan proyek karena perangkat lunak tidak terlihat. Yang terbaik yang bisa dia lakukan mungkin adalah memantau milestone yang telah diselesaikan oleh tim pengembangan dan dokumen yang telah dihasilkan—yang merupakan indikator kasar dari kemajuan yang dicapai. Ketidaktampakan perangkat lunak membuat sulit untuk menilai kemajuan suatu proyek dan merupakan penyebab utama kompleksitas pengelolaan proyek perangkat lunak.

Perubahan: Karena bagian perangkat lunak dari sistem apa pun lebih mudah diubah dibandingkan dengan bagian perangkat keras, bagian perangkat lunak adalah yang paling sering diubah. Hal ini terutama berlaku pada tahap akhir proyek. Sejauh menyangkut pengembangan perangkat keras, setiap perubahan yang terlambat pada spesifikasi sistem

perangkat keras yang sedang dikembangkan biasanya berarti mengulang keseluruhan proyek. Ini membuat perubahan yang terlambat pada proyek perangkat keras menjadi sangat mahal untuk dilakukan. Ini mungkin adalah alasan mengapa perubahan persyaratan sering terjadi dalam proyek perangkat lunak. Perubahan ini biasanya timbul dari perubahan praktik bisnis, perubahan perangkat keras atau perangkat lunak yang mendasarinya (misalnya sistem operasi, aplikasi lain), atau hanya karena pikiran kliennya.

Kompleksitas: Bahkan perangkat lunak berukuran sedang memiliki jutaan bagian (fungsi) yang berinteraksi satu sama lain dalam banyak cara—penggabungan data, proses serial dan bersamaan, transisi status, ketergantungan kontrol, berbagi file, dll. Karena kompleksitas fungsi yang melekat produk perangkat lunak dalam hal bagian dasar yang membentuk perangkat lunak, banyak jenis risiko yang terkait dengan pengembangannya. Hal ini membuat pengelolaan proyek-proyek ini jauh lebih sulit dibandingkan dengan banyak jenis proyek lainnya.

Keunikan: Setiap proyek perangkat lunak biasanya dikaitkan dengan banyak fitur atau situasi unik. Hal ini membuat setiap proyek jauh berbeda dari yang lain. Ini tidak seperti proyek di domain lain, seperti manufaktur mobil atau manufaktur baja di mana proyek lebih dapat diprediksi. Karena keunikan proyek perangkat lunak, seorang manajer proyek dalam menjalankan proyek menghadapi banyak masalah yang sangat berbeda dengan masalah lain yang pernah ia temui di masa lalu. Akibatnya, seorang manajer proyek perangkat lunak harus menghadapi banyak masalah tak terduga di hampir setiap proyek yang dia kelola.

Solusi yang tepat: Komponen mekanis seperti mur dan baut biasanya bekerja dengan baik selama masih dalam toleransi 1 persen atau lebih dari ukuran yang ditentukan. Namun, parameter pemanggilan fungsi dalam suatu program harus benar-benar sesuai dengan definisi fungsi. Persyaratan ini tidak hanya mempersulit untuk mendapatkan produk perangkat lunak dan bekerja, tetapi juga membuat penggunaan kembali bagian dari satu produk perangkat lunak menjadi sulit. Persyaratan kesesuaian yang tepat dari parameter fungsi ini menimbulkan risiko tambahan dan berkontribusi pada kompleksitas pengelolaan proyek perangkat lunak.

Pekerjaan yang berorientasi pada tim dan intelektual: Proyek pengembangan perangkat lunak mirip dengan proyek penelitian dalam arti bahwa keduanya melibatkan pekerjaan yang berorientasi pada tim dan intensif kecerdasan. Sebaliknya, proyek di banyak domain bersifat padat karya dan setiap anggota bekerja dalam tingkat otonomi yang tinggi. Contoh proyek tersebut adalah menanam padi, meletakkan jalan, manufaktur jalur perakitan, membangun gedung bertingkat, dll. Dalam proyek pengembangan perangkat lunak, aktivitas siklus hidup tidak hanya sangat intensif, tetapi setiap anggota harus berinteraksi, meninjau, dan antarmuka dengan beberapa anggota lain, yang merupakan dimensi lain dari kompleksitas proyek perangkat lunak.

3.2 TANGGUNG JAWAB MANAJER PROYEK PERANGKAT LUNAK

Pada bagian ini, kita akan memeriksa tanggung jawab pekerjaan utama seorang manajer proyek dan keterampilan yang diperlukan untuk mencapainya.

Tanggung Jawab Pekerjaan untuk Mengelola Proyek Perangkat Lunak

Seorang manajer proyek perangkat lunak mengambil tanggung jawab keseluruhan mengarahkan proyek untuk sukses. Faktanya, sangat sulit untuk secara objektif menggambarkan tanggung jawab pekerjaan yang tepat dari seorang manajer proyek. Tanggung jawab pekerjaan seorang manajer proyek berkisar dari aktivitas yang tidak terlihat seperti membangun moral tim hingga presentasi pelanggan yang sangat terlihat. Sebagian besar manajer mengambil tanggung jawab untuk penulisan proposal proyek, estimasi biaya proyek, penjadwalan, staf proyek, penyesuaian proses perangkat lunak, pemantauan dan

pengendalian proyek, manajemen konfigurasi perangkat lunak, manajemen risiko, penulisan dan presentasi laporan manajerial, dan berinteraksi dengan klien. Kegiatan tersebut tentunya banyak dan beragam. Kita masih dapat secara luas mengklasifikasikan kegiatan ini menjadi dua jenis utama—perencanaan proyek dan pemantauan dan pengendalian proyek.

Beragam tanggung jawab manajer proyek secara luas dapat mengklasifikasikan ke dalam dua kategori utama berikut:

- Perencanaan proyek, dan
- Pemantauan dan pengendalian proyek.

Perencanaan proyek: Perencanaan proyek dilakukan segera setelah fase studi kelayakan dan sebelum dimulainya fase analisis kebutuhan dan spesifikasi. Perencanaan proyek melibatkan memperkirakan beberapa karakteristik proyek dan kemudian merencanakan kegiatan proyek berdasarkan perkiraan yang dibuat. Rencana proyek awal direvisi dari waktu ke waktu seiring kemajuan proyek dan lebih banyak data proyek tersedia.

Pemantauan dan pengendalian proyek: Kegiatan pemantauan dan pengendalian proyek dilakukan setelah kegiatan pengembangan dimulai. Fokus kegiatan pemantauan dan pengendalian proyek adalah untuk memastikan bahwa pengembangan perangkat lunak berjalan sesuai rencana. Saat melaksanakan kegiatan pemantauan dan pengendalian proyek, manajer proyek terkadang merasa perlu untuk mengubah rencana untuk mengatasi situasi tertentu yang dihadapi.

Keterampilan yang Diperlukan untuk Mengelola Proyek Perangkat Lunak

Pengetahuan teoritis tentang berbagai teknik manajemen proyek tentu penting untuk menjadi manajer proyek yang sukses. Namun, pengetahuan teoritis murni dari berbagai teknik manajemen proyek tidak akan membuat seseorang menjadi manajer proyek yang sukses. Manajemen proyek perangkat lunak yang efektif membutuhkan penilaian kualitatif yang baik dan kemampuan pengambilan keputusan. Selain memiliki pemahaman yang baik tentang teknik manajemen proyek perangkat lunak terbaru seperti estimasi biaya, manajemen risiko, dan manajemen konfigurasi, dll., manajer proyek memerlukan keterampilan komunikasi yang baik dan kemampuan untuk menyelesaikan pekerjaan. Beberapa keterampilan seperti melacak dan mengendalikan kemajuan proyek, interaksi pelanggan, presentasi manajerial, dan pembangunan tim sebagian besar diperoleh melalui pengalaman. Namun demikian, pentingnya pengetahuan yang baik tentang teknik manajemen proyek yang lazim tidak dapat terlalu ditekankan. Tujuan dari sisa bab ini adalah untuk memperkenalkan pembaca pada hal yang sama.

Tiga keterampilan yang paling penting untuk manajemen proyek yang sukses adalah sebagai berikut:

- Pengetahuan tentang teknik manajemen proyek.
- Kemampuan pengambilan keputusan.
- Pengalaman sebelumnya dalam mengelola proyek serupa.

Dengan diskusi singkat tentang tanggung jawab keseluruhan dari manajer proyek perangkat lunak dan keterampilan yang diperlukan untuk mencapainya, di bagian selanjutnya kita akan memeriksa beberapa masalah penting dalam perencanaan proyek.

3.3 PERENCANAAN PROYEK

Setelah proyek telah ditemukan layak, manajer proyek perangkat lunak melakukan perencanaan proyek. Perencanaan proyek dilakukan dan diselesaikan sebelum kegiatan pembangunan dimulai. Perencanaan proyek membutuhkan kehati-hatian dan perhatian penuh karena komitmen terhadap waktu dan perkiraan sumber daya yang tidak realistis mengakibatkan selip jadwal. Penundaan jadwal dapat menyebabkan ketidakpuasan

pelanggan dan berdampak buruk pada moral tim. Bahkan dapat menyebabkan kegagalan proyek. Untuk alasan ini, perencanaan proyek dilakukan oleh manajer proyek dengan sangat hati-hati dan penuh perhatian.

Namun, untuk perencanaan proyek yang efektif, selain pengetahuan menyeluruh tentang berbagai teknik estimasi, pengalaman masa lalu sangat penting. Selama perencanaan proyek, manajer proyek melakukan kegiatan berikut.

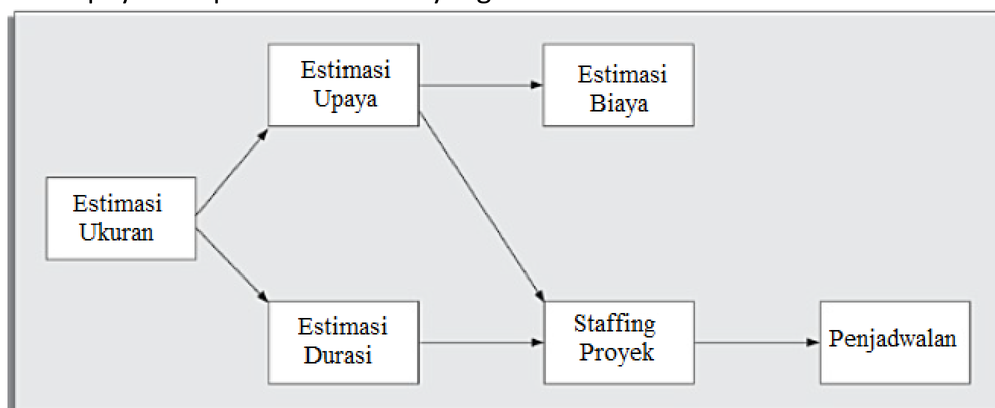
Estimasi: Atribut proyek berikut diperkirakan.

- Biaya: Berapa biaya untuk mengembangkan produk perangkat lunak?
- Durasi: Berapa lama waktu yang dibutuhkan untuk mengembangkan produk?
- Upaya: Berapa banyak usaha yang diperlukan untuk mengembangkan produk?

Keefektifan semua aktivitas perencanaan selanjutnya seperti penjadwalan dan penempatan staf bergantung pada akurasi yang digunakan untuk membuat ketiga estimasi ini.

- **Penjadwalan:** Setelah semua parameter proyek yang diperlukan telah diperkirakan, jadwal untuk tenaga kerja dan sumber daya lainnya dikembangkan.
- **Kepegawaian:** Organisasi staf dan rencana kepegawaian dibuat.
- **Manajemen risiko :** Ini termasuk identifikasi risiko, analisis, dan perencanaan pengurangan.
- **Paket lain-lain:** Ini termasuk membuat beberapa rencana lain seperti rencana jaminan kualitas, dan rencana manajemen konfigurasi, dll.

Gambar 3.1 menunjukkan urutan kegiatan perencanaan yang dilakukan. Amati bahwa estimasi ukuran adalah aktivitas pertama yang dilakukan manajer proyek selama perencanaan proyek. Ukuran adalah parameter paling mendasar yang menjadi dasar pembuatan semua estimasi dan rencana proyek lainnya. Seperti dapat dilihat dari Gambar 3.1, berdasarkan estimasi ukuran, upaya yang diperlukan untuk menyelesaikan proyek dan durasi pembangunan yang akan dilakukan diperkirakan. Berdasarkan estimasi usaha, biaya proyek dihitung. Estimasi biaya menjadi dasar untuk melakukan negosiasi harga dengan pelanggan. Kegiatan perencanaan lainnya seperti penempatan staf, penjadwalan, dll. dilakukan berdasarkan upaya dan perkiraan durasi yang dibuat.



Gambar 3.1 Urutan prioritas di antara kegiatan perencanaan.

Perencanaan Jendela Geser

Biasanya sangat sulit untuk membuat rencana yang akurat untuk proyek-proyek besar pada permulaan proyek. Sebagian dari kesulitan muncul dari kenyataan bahwa proyek-proyek besar membutuhkan waktu beberapa tahun untuk diselesaikan. Akibatnya, selama rentang waktu proyek, parameter proyek, ruang lingkup proyek, staf proyek, dll., sering berubah

secara drastis yang mengakibatkan rencana awal menjadi kacau. Untuk mengatasi masalah ini, terkadang manajer proyek melakukan perencanaan proyek melalui beberapa tahap. Artinya, setelah rencana proyek awal dibuat, ini direvisi secara berkala. Merencanakan proyek melalui beberapa tahap melindungi manajer dari membuat komitmen besar di awal proyek. Teknik perencanaan terhuyung-huyung ini dikenal sebagai perencanaan jendela geser. Dalam teknik perencanaan jendela geser, dimulai dengan rencana awal, proyek direncanakan lebih akurat melalui beberapa tahap.

Pada awal proyek, manajer proyek memiliki pengetahuan yang tidak lengkap tentang seluk beluk proyek. Basis informasinya secara bertahap meningkat seiring kemajuan proyek melalui fase pengembangan yang berbeda. Kompleksitas kegiatan proyek yang berbeda menjadi jelas, beberapa risiko yang diantisipasi dapat diselesaikan, dan risiko baru muncul. Parameter proyek diestimasi ulang secara berkala seiring dengan berkembangnya pemahaman dan juga secara aperiodik saat parameter proyek berubah. Dengan mempertimbangkan perkembangan ini, manajer proyek dapat merencanakan kegiatan selanjutnya dengan lebih akurat dan dengan tingkat kepercayaan yang meningkat.

Dokumen SPMP Perencanaan Proyek

Setelah perencanaan proyek selesai, manajer proyek mendokumentasikan rencana mereka dalam dokumen rencana manajemen proyek perangkat lunak (SPMP). Di bawah ini adalah item-item berbeda yang harus dibahas dalam dokumen SPMP. Daftar ini dapat digunakan sebagai kemungkinan pengorganisasian dokumen SPMP.

Organisasi dokumen rencana manajemen proyek perangkat lunak (SPMP)

1. Perkenalan
   (a) Tujuan
   (b) Fungsi Utama
   (c) Masalah Kinerja
   (d) Kendala Manajemen dan Teknis
2. Perkiraan proyek
   (a) Data Historis yang Digunakan
   (b) Teknik Estimasi yang Digunakan
   (c) Estimasi Upaya, Sumber Daya, Biaya, dan Durasi Proyek
3. Jadwal
   (a) Struktur Perincian Kerja
   (b) Representasi Jaringan Tugas
   (c) Representasi Gantt Chart
   (d) Representasi Grafik PERT
4. Sumber daya proyek
   (a) Orang
   (b) Perangkat Keras dan Perangkat Lunak
   (c) Sumber Daya Khusus
5. Organisasi staf
   (a) Struktur Tim
   (b) Pelaporan Manajemen
6. Rencana manajemen risiko
   (a) Analisis Risiko
   (b) Identifikasi Risiko
   (c) Estimasi Risiko
   (d) Prosedur Pengurangan Risiko
7. Rencana pelacakan dan kontrol proyek

(a) Metrik yang akan dilacak
(b) Rencana pelacakan
(c) Rencana pengendalian

8. Berbagai rencana
   (a) Penjahitan Proses
   (b) Rencana Penjaminan Mutu
   (c) Rencana Manajemen Konfigurasi
   (d) Validasi dan Verifikasi
   (e) Rencana Pengujian Sistem
   (f) Rencana Pengiriman, Pemasangan, dan Pemeliharaan

3.4 METRIK UNTUK ESTIMASI UKURAN PROYEK

Seperti yang telah disebutkan, estimasi ukuran proyek yang akurat sangat penting untuk estimasi yang memuaskan dari semua parameter proyek lainnya seperti upaya, waktu penyelesaian, dan total biaya proyek. Sebelum membahas metrik yang tersedia untuk memperkirakan ukuran proyek, mari kita periksa apa arti sebenarnya dari istilah "ukuran proyek". Ukuran proyek jelas bukan jumlah byte yang ditempati oleh kode sumber, juga bukan ukuran kode yang dapat dieksekusi.

Ukuran proyek adalah ukuran kompleksitas masalah dalam hal upaya dan waktu yang dibutuhkan untuk mengembangkan produk. Saat ini, dua metrik yang populer digunakan untuk mengukur ukuran—baris kode (LOC) dan titik fungsi (FP). Masing-masing metrik ini memiliki kelebihan dan kekurangannya sendiri. Berdasarkan keuntungan relatifnya, satu metrik mungkin lebih tepat daripada yang lain dalam situasi tertentu.

Baris Kode (LOC)

LOC mungkin yang paling sederhana di antara semua metrik yang tersedia untuk mengukur ukuran proyek. Akibatnya, metrik ini sangat populer. Metrik ini mengukur ukuran proyek dengan menghitung jumlah instruksi sumber dalam program yang dikembangkan. Jelas, saat menghitung jumlah instruksi sumber, baris komentar, dan baris header diabaikan.

Menentukan jumlah LOC di akhir proyek sangat sederhana. Namun, estimasi akurat jumlah LOC pada awal proyek adalah tugas yang sangat sulit. Seseorang mungkin dapat memperkirakan jumlah LOC pada awal proyek, hanya dengan menggunakan beberapa bentuk kerja tebakan sistematis. Tebakan sistematis biasanya melibatkan hal berikut. Manajer proyek membagi masalah ke dalam modul, dan setiap modul menjadi sub-modul dan seterusnya, sampai LOC modul tingkat daun cukup kecil untuk diprediksi. Untuk dapat memprediksi jumlah LOC untuk berbagai modul tingkat daun dengan cukup akurat, pengalaman masa lalu dalam mengembangkan modul serupa sangat membantu. Dengan menambahkan perkiraan untuk semua modul level daun bersama-sama, manajer proyek sampai pada estimasi ukuran total. Terlepas dari kesederhanaan konseptualnya, metrik LOC memiliki beberapa kekurangan ketika digunakan untuk mengukur ukuran masalah. Beberapa kekurangan penting dari metrik LOC adalah:

LOC adalah ukuran aktivitas pengkodean saja. Ukuran masalah yang baik harus mempertimbangkan upaya total yang diperlukan untuk melakukan berbagai aktivitas siklus hidup (yaitu spesifikasi, desain, kode, pengujian, dll.) dan bukan hanya upaya pengkodean. LOC, bagaimanapun, berfokus pada aktivitas pengkodean saja—itu hanya menghitung jumlah baris sumber dalam program akhir. Kita telah membahas di Bab 2 bahwa pengkodean hanyalah sebagian kecil dari keseluruhan upaya pengembangan perangkat lunak. Asumsi implisit yang dibuat oleh metrik LOC adalah bahwa upaya pengembangan produk secara keseluruhan semata-mata ditentukan dari upaya pengkodean saja adalah cacat.

Anggapan bahwa upaya total yang diperlukan untuk mengembangkan proyek sebanding dengan upaya pengkodean mudah dilawan dengan mencatat fakta bahwa bahkan ketika masalah desain atau pengujian sangat kompleks, ukuran kode mungkin kecil dan sebaliknya. Dengan demikian, upaya desain dan pengujian bisa sangat tidak proporsional dengan upaya pengkodean. Ukuran kode, oleh karena itu, jelas merupakan indikator yang tidak tepat dari ukuran masalah.

Hitungan LOC tergantung pada pilihan instruksi spesifik: LOC memberikan nilai numerik dari ukuran masalah yang dapat sangat bervariasi dengan gaya pengkodean masing-masing programmer. Dengan gaya pengkodean (pilihan tata letak kode) pilihan instruksi dalam penulisan program, dan algoritma khusus yang digunakan. Programmer yang berbeda dapat menyusun kode mereka dengan cara yang sangat berbeda. Misalnya, satu programmer mungkin menulis beberapa instruksi sumber pada satu baris, sedangkan yang lain mungkin membagi satu instruksi menjadi beberapa baris. Kecuali masalah ini ditangani dengan memuaskan, ada kemungkinan untuk mencapai ukuran ukuran yang sangat berbeda untuk program yang pada dasarnya identik. Masalah ini sebagian besar dapat diatasi dengan menghitung token bahasa dalam suatu program daripada baris kode. Namun, masalah yang lebih rumit muncul karena pilihan spesifik dari instruksi yang dibuat dalam penulisan program. Misalnya, seorang programmer dapat menggunakan pernyataan switch dalam menulis program C dan yang lain dapat menggunakan urutan pernyataan if ... then ... else .... Oleh karena itu, berikut ini dapat dengan mudah disimpulkan. Bahkan untuk masalah pemrograman yang sama, programmer yang berbeda mungkin menghasilkan program yang memiliki jumlah LOC yang sangat berbeda. Situasi ini tidak membaik, bahkan jika token bahasa dihitung alih-alih baris kode.

Ukuran LOC berkorelasi buruk dengan kualitas dan efisiensi kode: Ukuran kode yang lebih besar tidak selalu berarti kualitas kode yang lebih baik atau efisiensi yang lebih tinggi. Beberapa programmer menghasilkan kode yang panjang dan rumit karena mereka tidak menggunakan set instruksi yang tersedia secara efektif atau menggunakan algoritma yang tidak tepat. Faktanya, benar bahwa sepotong kode yang buruk dan ditulis dengan ceroboh dapat memiliki jumlah instruksi sumber yang lebih banyak daripada sepotong kode yang efisien dan telah ditulis dengan cermat. Menghitung produktivitas sebagai LOC yang dihasilkan per orang-bulan dapat mendorong programmer untuk menulis banyak kode berkualitas buruk daripada lebih sedikit baris kode berkualitas tinggi untuk mencapai fungsionalitas yang sama.

Metrik LOC menghukum penggunaan bahasa pemrograman tingkat tinggi dan penggunaan kembali kode: Paradoksnya adalah jika seorang programmer secara sadar menggunakan beberapa rutinitas perpustakaan, maka jumlah LOC akan lebih rendah. Ini akan muncul sebagai ukuran program yang lebih kecil, dan pada gilirannya, akan menunjukkan upaya yang lebih rendah, jadi, jika manajer menggunakan jumlah LOC untuk mengukur upaya yang dilakukan oleh developer yang berbeda (yaitu, produktivitas mereka), mereka akan mencegah penggunaan kembali kode oleh pengembang. Metode pemrograman modern seperti pemrograman berorientasi objek dan penggunaan kembali komponen membuat hubungan antara LOC dan atribut proyek lainnya menjadi kurang tepat.

Metrik LOC mengukur kompleksitas leksikal suatu program dan tidak membahas masalah yang lebih penting dari kompleksitas logis dan struktural: Antara dua program dengan jumlah LOC yang sama, program yang menggabungkan logika kompleks akan membutuhkan lebih banyak upaya untuk dikembangkan daripada program dengan logika yang sangat sederhana. Untuk menyadari mengapa demikian, bayangkan upaya yang diperlukan untuk mengembangkan program yang memiliki banyak loop bersarang dan

konstruksi keputusan dan bandingkan dengan program lain yang hanya memiliki aliran kontrol sekuensial.

Sangat sulit untuk secara akurat memperkirakan LOC dari program akhir dari spesifikasi masalah: Seperti yang telah dibahas, pada waktu inisiasi proyek, adalah tugas yang sangat sulit untuk secara akurat memperkirakan jumlah baris kode (LOC) yang akan dimiliki program setelahnya. perkembangan. Hitungan LOC dapat dihitung secara akurat hanya setelah kode dikembangkan sepenuhnya. Karena perencanaan proyek dilakukan bahkan sebelum aktivitas pengembangan dimulai, metrik LOC tidak banyak berguna bagi manajer proyek selama perencanaan proyek.

Dari perspektif manajer proyek, kelemahan terbesar dari metrik LOC adalah bahwa jumlah LOC sangat sulit untuk diperkirakan selama tahap perencanaan proyek, dan hanya dapat dihitung secara akurat setelah pengembangan perangkat lunak selesai.

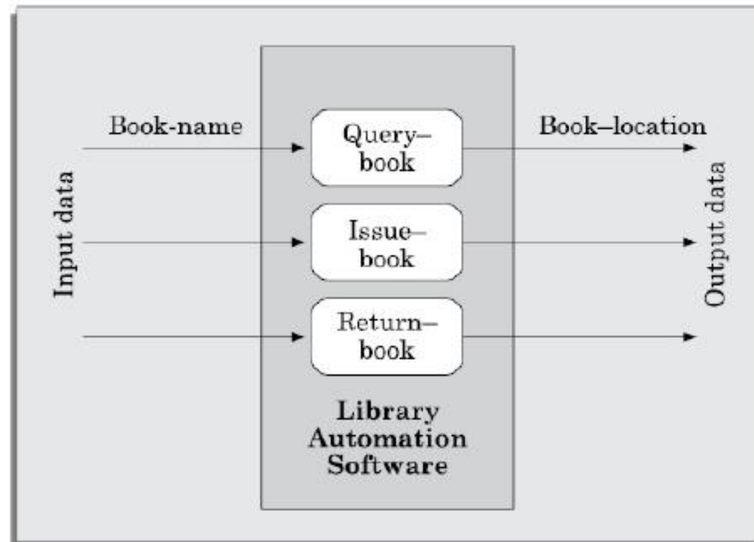
Metrik Titik Fungsi/*Fuction point* (FP)

Metrik titik fungsi diusulkan oleh Albrecht pada tahun 1983. Metrik ini mengatasi banyak kekurangan metrik LOC. Sejak dimulai pada akhir 1970-an, metrik titik fungsi terus mendapatkan popularitas. Metrik titik fungsi memiliki beberapa keunggulan dibandingkan metrik LOC. Salah satu keuntungan penting metrik titik fungsi dibandingkan metrik LOC adalah dapat dengan mudah dihitung dari spesifikasi masalah itu sendiri. Menggunakan metrik LOC, di sisi lain, ukuran dapat ditentukan secara akurat hanya setelah produk sepenuhnya dikembangkan.

Ide konseptual di balik metrik titik fungsi adalah sebagai berikut. Ukuran produk perangkat lunak secara langsung bergantung pada jumlah fungsi atau fitur tingkat tinggi yang berbeda yang didukungnya. Asumsi ini masuk akal, karena setiap fitur akan membutuhkan upaya tambahan untuk diterapkan. Secara konseptual, metrik titik fungsi didasarkan pada gagasan bahwa produk perangkat lunak yang mendukung banyak fitur pasti akan berukuran lebih besar daripada produk dengan jumlah fitur yang lebih sedikit.

Meskipun setiap fitur membutuhkan upaya untuk dikembangkan, fitur yang berbeda mungkin membutuhkan upaya yang sangat berbeda untuk dikembangkan. Misalnya, dalam perangkat lunak perbankan, fungsi untuk menampilkan pesan bantuan mungkin jauh lebih mudah untuk dikembangkan dibandingkan dengan mengatakan fungsi yang melakukan transaksi perbankan yang sebenarnya. Ini telah dipertimbangkan oleh metrik titik fungsi dengan menghitung jumlah item data input dan output dan jumlah file yang diakses oleh fungsi tersebut. Asumsi implisit yang dibuat adalah bahwa semakin banyak jumlah item data yang dibaca oleh suatu fungsi dari pengguna dan keluaran dan semakin banyak jumlah file yang diakses, semakin tinggi kompleksitas fungsi tersebut. Sekarang mari kita menganalisis mengapa asumsi ini harus benar secara intuitif. Setiap fitur ketika dipanggil biasanya membaca beberapa data input dan kemudian mengubahnya menjadi data output yang diperlukan.

Misalnya, fitur kueri buku (lihat Gambar 3.2) dari Perangkat Lunak Otomasi Perpustakaan mengambil nama buku sebagai masukan dan menampilkan lokasinya di perpustakaan dan jumlah total salinan yang tersedia. Demikian pula, fitur buku terbitan dan buku pengembalian menghasilkan outputnya berdasarkan data input yang sesuai. Oleh karena itu dapat dikatakan bahwa perhitungan jumlah item data input dan output akan memberikan indikasi ukuran kode yang lebih akurat dibandingkan dengan hanya menghitung jumlah fungsi tingkat tinggi yang didukung oleh sistem.



Gambar 3.2 Sistem berfungsi sebagai pemetaan data input ke data output.

Albrecht mendalilkan bahwa selain jumlah fungsi dasar yang dilakukan perangkat lunak, ukuran juga tergantung pada jumlah file dan jumlah antarmuka yang terkait dengan perangkat lunak. Di sini, antarmuka mengacu pada mekanisme yang berbeda untuk transfer data dengan sistem eksternal termasuk antarmuka dengan pengguna, antarmuka dengan komputer eksternal, dll.

Komputasi metrik titik fungsi (FP)

Ukuran produk perangkat lunak (dalam satuan titik fungsi atau FP) dihitung menggunakan karakteristik berbeda dari produk yang diidentifikasi dalam spesifikasi persyaratannya. Ini dihitung menggunakan tiga langkah berikut:

- Langkah 1: Hitung titik fungsi yang tidak disesuaikan (UFP) menggunakan ekspresi heuristik.
- Langkah 2: Perbaiki UFP untuk mencerminkan kompleksitas aktual dari berbagai parameter yang digunakan dalam perhitungan UFP.
- Langkah 3: Hitung FP dengan menyempurnakan UFP lebih lanjut untuk memperhitungkan karakteristik spesifik proyek yang dapat memengaruhi seluruh upaya pengembangan.

Langkah 1: Perhitungan UFP

Titik fungsi yang tidak disesuaikan (UFP) dihitung sebagai jumlah bobot dari lima karakteristik produk seperti yang ditunjukkan dalam ekspresi berikut. Bobot yang terkait dengan lima karakteristik ditentukan secara empiris oleh Albrecht melalui data yang dikumpulkan dari banyak proyek.

Persamaan (3.1)

UFP = (Jumlah input)*4 + (Jumlah output)*5 + (Jumlah pertanyaan)*4 + (Jumlah file)*10 + (Jumlah antarmuka)*10

Arti dari parameter yang berbeda dari Persamaan. 3.1 adalah sebagai berikut:

1. **Jumlah input:** Setiap input item data oleh pengguna dihitung. Namun, perlu dicatat bahwa input data dianggap berbeda dari pertanyaan pengguna. Permintaan adalah perintah pengguna seperti printaccount- balance yang tidak memerlukan nilai data untuk dimasukkan oleh pengguna. Pertanyaan dihitung secara terpisah (lihat poin ketiga di bawah). Perlu dicatat lebih lanjut bahwa

item data individual yang dimasukkan oleh pengguna tidak hanya ditambahkan untuk menghitung jumlah input, tetapi input terkait dikelompokkan dan dianggap sebagai input tunggal. Misalnya, saat memasukkan data tentang seorang karyawan ke perangkat lunak daftar gaji karyawan; nama item data, usia, jenis kelamin, alamat, nomor telepon, dll. bersama-sama dianggap sebagai satu input. Semua item data ini dapat dianggap terkait, karena mereka menggambarkan satu karyawan.

2. **Jumlah keluaran**: Keluaran yang dipertimbangkan termasuk laporan yang dicetak, keluaran layar, pesan kesalahan yang dihasilkan, dll. Saat menghitung jumlah keluaran, item data individual dalam laporan tidak dipertimbangkan; tetapi satu set item data terkait dihitung hanya sebagai satu keluaran.
3. **Jumlah pertanyaan**: Permintaan adalah perintah pengguna (tanpa input data apa pun) dan hanya memerlukan beberapa tindakan yang harus dilakukan oleh sistem. Jadi, jumlah total pertanyaan pada dasarnya adalah jumlah kueri interaktif yang berbeda (tanpa input data) yang dapat dibuat oleh pengguna. Contoh pertanyaan tersebut adalah mencetak saldo rekening, mencetak semua nilai siswa, menampilkan nama pemegang peringkat, dll.
4. **Jumlah file:** File yang dimaksud di sini adalah file logis. File logis mewakili sekelompok data yang terkait secara logis. File logis termasuk struktur data serta file fisik.
5. **Jumlah antarmuka**: Di sini antarmuka menunjukkan mekanisme berbeda yang digunakan untuk bertukar informasi dengan sistem lain. Contoh antarmuka tersebut adalah file data pada kaset, disk, tautan komunikasi dengan sistem lain, dll.

Langkah 2: Perbaiki parameter

UFP yang dihitung pada akhir langkah 1 adalah indikator kasar dari ukuran masalah. UFP ini perlu disempurnakan. Hal ini dimungkinkan, karena setiap parameter (input, output, dll.) secara implisit diasumsikan memiliki kompleksitas rata-rata. Namun, ini jarang benar. Misalnya, beberapa nilai input mungkin sangat kompleks, beberapa sangat sederhana, dll. Untuk mempertimbangkan masalah ini, UFP disempurnakan dengan mempertimbangkan kompleksitas parameter perhitungan UFP (Persamaan 3.1). Kompleksitas setiap parameter dinilai ke dalam tiga kategori besar—sederhana, rata-rata, atau kompleks. Bobot untuk parameter yang berbeda ditentukan berdasarkan nilai numerik yang ditunjukkan pada Tabel 3.1. Berdasarkan bobot parameter ini, nilai parameter dalam UFP disempurnakan. Misalnya, daripada setiap input dihitung sebagai empat FP, input yang sangat sederhana dihitung sebagai tiga FP dan input yang sangat kompleks sebagai enam FP.

Tabel 3.1 Penyempurnaan Entitas Titik Fungsi

Jenis	Simpel	Menengah	Komplek
Input (I)	3	4	6
Output (O)	4	5	7
Permintaan (E)	3	4	6
Jumlah File (F)	7	10	15
Jumlah interface	5	7	10

Langkah 3: Perbaiki UFP berdasarkan kompleksitas proyek secara keseluruhan

Pada langkah terakhir, beberapa faktor yang dapat memengaruhi ukuran proyek secara keseluruhan dipertimbangkan untuk menyempurnakan UFP yang dihitung pada

langkah 2. Contoh parameter proyek tersebut yang dapat memengaruhi ukuran proyek termasuk tingkat transaksi yang tinggi, persyaratan waktu respons, ruang lingkup untuk digunakan kembali, dll. Albrecht mengidentifikasi 14 parameter yang dapat mempengaruhi upaya pengembangan. Daftar parameter ini telah ditunjukkan pada Tabel 3.2. Masing-masing dari 14 parameter ini diberi nilai dari 0 (tidak ada atau tidak ada pengaruh) hingga 6 (pengaruh kuat). Angka-angka yang dihasilkan dijumlahkan, menghasilkan total derajat pengaruh (DI). Faktor kompleksitas teknis (TCF) untuk proyek dihitung dan TCF dikalikan dengan UFP untuk menghasilkan FP. TCF mengungkapkan dampak keseluruhan dari parameter proyek terkait pada upaya pengembangan. TCF dihitung sebagai (0,65+0,01*DI). Karena DI dapat bervariasi dari 0 hingga 84, TCF dapat bervariasi dari 0,65 hingga 1,49. Terakhir, FP diberikan sebagai produk dari UFP dan TCF. Yaitu, FP=UFP*TCF.

Tabel 3.2 Faktor Penyesuaian Kompleksitas Relatif Titik Fungsi

Faktor
Persyaratan untuk pencadangan dan pemulihan yang andal
Persyaratan untuk komunikasi data
Tingkat pemrosesan terdistribusi
Persyaratan kinerja
Lingkungan operasional yang diharapkan
Luas entri data online
Tingkat input data online multi-layar atau multi-operasi
Tingkat pembaruan online file master
Tingkat input, output, kueri dan file online yang kompleks
Tingkat pemrosesan data yang kompleks
Sejauh kode yang dikembangkan saat ini dapat dirancang untuk digunakan kembali
Tingkat konversi dan pemasangan termasuk dalam desain
Luas beberapa instalasi dalam sebuah organisasi dan berbagai organisasi pelanggan
Tingkat perubahan dan fokus pada kemudahan penggunaan

Contoh 3.1 Tentukan ukuran titik fungsi dari ukuran software supermarket berikut. Sebuah supermarket perlu mengembangkan perangkat lunak berikut untuk mendorong pelanggan tetap. Untuk ini, pelanggan perlu memberikan alamat tempat tinggal, nomor telepon, dan nomor SIM. Setiap pelanggan yang mendaftar untuk skema ini diberi nomor pelanggan unik (CN) oleh komputer. Berdasarkan CN yang dihasilkan, petugas secara manual menyiapkan kartu identitas pelanggan setelah mendapatkan tanda tangan manajer pasar di atasnya. Pelanggan dapat menunjukkan kartu identitas pelanggannya kepada staf check out saat melakukan pembelian. Dalam hal ini, nilai pembeliannya dikreditkan terhadap CN-nya. Setiap akhir tahun, supermarket bermaksud untuk memberikan hadiah kejutan kepada 10 pelanggan yang melakukan pembelian total tertinggi sepanjang tahun. Juga, itu bermaksud untuk memberikan koin emas 22 caret kepada setiap pelanggan yang pembeliannya melebihi Rp. 10.000. Entri terhadap CN diatur ulang pada hari terakhir setiap tahun setelah daftar pemenang hadiah dibuat. Asumsikan bahwa berbagai karakteristik proyek yang menentukan kompleksitas pengembangan perangkat lunak menjadi rata-rata.

Jawaban:

Langkah 1: Dari pemeriksaan deskripsi masalah, kita menemukan bahwa ada dua input, tiga output, dua file, dan tidak ada antarmuka. Dua file akan diperlukan, satu untuk menyimpan rincian pelanggan dan satu lagi untuk menyimpan catatan pembelian harian. Sekarang, dengan menggunakan persamaan 3.1, kita mendapatkan:

$UFP = 2 \times 4 + 3 \times 5 + 1 \times 4 + 10 \times 2 + 0 \times 10 = 47$

Langkah 2: A l l parameternya memiliki kompleksitas sedang, kecuali parameter keluaran registrasi pelanggan, di mana satu-satunya keluaran adalah nilai CN. Akibatnya, kompleksitas parameter keluaran dari fungsi registrasi pelanggan dapat dikategorikan sederhana. Dengan melihat Tabel 3.1, kita menemukan bahwa nilai untuk output sederhana diberikan menjadi 4. UFP dapat disempurnakan sebagai berikut:

UFP = 3 × 4 + 2 × 5 + 1 × 4 + 10 × 2 + 0 × 10 = 46

Oleh karena itu, UFP akan menjadi 46.

Langkah 3: Karena faktor penyesuaian kompleksitas memiliki nilai rata-rata, maka total derajat pengaruhnya adalah: DI = 14 × 4 = 56

TCF = 0,65 + 0,01 + 56 = 1,21

Oleh karena itu, FP yang disesuaikan=46*1.21=55.66

Kekurangan metrik titik fitur: Kelemahan utama dari pengukuran titik fungsi adalah tidak memperhitungkan kompleksitas algoritmik suatu fungsi. Artinya, metrik titik fungsi secara implisit mengasumsikan bahwa upaya yang diperlukan untuk merancang dan mengembangkan dua fungsi berbeda dari sistem adalah sama. Tapi, kita tahu bahwa ini sangat tidak mungkin benar. Upaya yang diperlukan untuk mengembangkan dua fungsi dapat sangat bervariasi. Sebagai contoh, pada software otomatisasi perpustakaan, fitur create-member akan jauh lebih sederhana dibandingkan dengan fitur loan-from-remote-library. FP hanya mempertimbangkan jumlah fungsi yang didukung sistem, tanpa membedakan tingkat kesulitan pengembangan berbagai fungsi. Untuk mengatasi masalah ini, perluasan ke metrik titik fungsi yang disebut metrik titik fitur telah diusulkan.

Metrik titik fitur menggabungkan kompleksitas algoritme sebagai parameter tambahan. Parameter ini memastikan bahwa ukuran yang dihitung menggunakan metrik titik fitur mencerminkan fakta bahwa semakin tinggi kompleksitas suatu fungsi, semakin besar upaya yang diperlukan untuk mengembangkannya—oleh karena itu, fungsi tersebut harus memiliki ukuran yang lebih besar dibandingkan dengan fungsi yang lebih sederhana.

Komentar kritis pada metrik titik fungsi dan titik fitur

Pendukung metrik titik fungsi dan titik fitur mengklaim bahwa kedua metrik ini tidak bergantung pada bahasa dan dapat dengan mudah dihitung dari dokumen SRS selama tahap perencanaan proyek itu sendiri. Di sisi lain, lawan mengklaim bahwa metrik ini subjektif dan memerlukan sulap. Contoh sifat subjektif dari metrik titik fungsi adalah cara seseorang mengelompokkan item data input dan output ke dalam grup yang terkait secara logis bisa sangat subjektif. Misalnya, pertimbangkan bahwa fungsi tertentu memerlukan nama karyawan dan alamat karyawan untuk dimasukkan. Ada kemungkinan bahwa seseorang dapat mempertimbangkan kedua item ini sebagai satu unit data, karena bagaimanapun, ini menggambarkan satu karyawan. Dimungkinkan juga bagi orang lain untuk mempertimbangkan alamat karyawan sebagai satu unit data input dan nama sebagai unit lainnya. Ambiguitas seperti itu meninggalkan ruang lingkup yang cukup untuk diperdebatkan dan tetap membuka kemungkinan bagi manajer proyek yang berbeda untuk sampai pada ukuran titik fungsi yang berbeda untuk masalah yang pada dasarnya sama.

3.5 TEKNIK ESTIMASI PROYEK

Estimasi berbagai parameter proyek merupakan kegiatan perencanaan proyek yang penting. Berbagai parameter proyek yang perlu diestimasi meliputi—ukuran proyek, upaya yang diperlukan untuk menyelesaikan proyek, durasi proyek, dan biaya. Estimasi akurat dari parameter ini penting, karena ini tidak hanya membantu dalam menentukan biaya proyek yang sesuai untuk pelanggan, tetapi juga membentuk dasar untuk perencanaan dan

penjadwalan sumber daya. Sejumlah besar teknik estimasi telah diusulkan oleh para peneliti. Ini secara luas dapat diklasifikasikan ke dalam tiga kategori utama:

- Teknik estimasi empiris
- Teknik heuristik
- Teknik estimasi analitis

Teknik Estimasi Empiris

Teknik estimasi empiris pada dasarnya didasarkan pada membuat tebakan terdidik dari parameter proyek. Saat menggunakan teknik ini, pengalaman sebelumnya dengan pengembangan produk serupa sangat membantu. Meskipun teknik estimasi empiris didasarkan pada akal sehat dan keputusan subjektif, selama bertahun-tahun, berbagai kegiatan yang terlibat dalam estimasi telah diformalkan untuk sebagian besar.

Teknik Heuristik

Teknik heuristik mengasumsikan bahwa hubungan yang ada di antara parameter proyek yang berbeda dapat dimodelkan secara memuaskan menggunakan ekspresi matematika yang sesuai. Setelah parameter dasar (independen) diketahui, parameter lain (tergantung) dapat dengan mudah ditentukan dengan mengganti nilai parameter independen dalam ekspresi matematika yang sesuai. Model estimasi heuristik yang berbeda dapat dibagi menjadi dua kategori besar berikut — model variabel tunggal dan multivariabel.

Model estimasi variabel tunggal mengasumsikan bahwa berbagai karakteristik proyek dapat diprediksi berdasarkan satu estimasi karakteristik dasar (independen) perangkat lunak sebelumnya seperti ukurannya. Model estimasi variabel tunggal mengasumsikan bahwa hubungan antara parameter yang akan diestimasi dan parameter independen yang sesuai dapat dicirikan oleh ekspresi bentuk berikut:

$$\text{Perkiraan Parameter} = c_1 \times e^{d_1}$$

Dalam ekspresi di atas, e mewakili karakteristik perangkat lunak yang telah diestimasi (variabel bebas). Parameter Estimasi adalah parameter dependen (yang akan diestimasi). Parameter dependen yang akan diestimasi dapat berupa usaha, durasi proyek, ukuran staf, dll., c_1 dan d_1 adalah konstanta. Nilai konstanta c_1 dan d_1 biasanya ditentukan dengan menggunakan data yang dikumpulkan dari proyek sebelumnya (data historis). Model COCOMO adalah contoh model estimasi biaya variabel tunggal. Model estimasi biaya multivariabel mengasumsikan bahwa suatu parameter dapat diprediksi berdasarkan nilai lebih dari satu parameter independen. Ini mengambil bentuk berikut:

$$\text{Perkiraan Sumber Daya} = c_1 \times p_1^{d_1} + c_2 \times p_2^{d_2} + \ldots$$

di mana, p_1, p_2, ... adalah karakteristik dasar (independen) dari perangkat lunak yang sudah diperkirakan, dan c_1, c_2, d_1, d_2, ... adalah konstanta. Model estimasi multivariabel diharapkan dapat memberikan estimasi yang lebih akurat dibandingkan dengan model variabel tunggal, karena parameter proyek biasanya dipengaruhi oleh beberapa parameter independen. Parameter independen mempengaruhi parameter dependen ke luasan yang berbeda. Ini dimodelkan oleh set konstanta yang berbeda c_1, d_1, c_2, d_2, ... Nilai konstanta ini biasanya ditentukan dari analisis data historis. Model COCOMO menengah dapat dianggap sebagai contoh model estimasi multivariabel.

Teknik Estimasi Analitik

Teknik estimasi analitis memperoleh hasil yang diperlukan dimulai dengan asumsi dasar tertentu mengenai suatu proyek. Tidak seperti teknik empiris dan heuristik, teknik analisis memang memiliki dasar ilmiah tertentu. Kita akan melihat bahwa dimulai dengan beberapa asumsi sederhana, ilmu perangkat lunak Halstead memperoleh beberapa hasil yang menarik. Ilmu perangkat lunak Halstead sangat berguna untuk memperkirakan upaya

pemeliharaan perangkat lunak. Bahkan, itu mengungguli teknik empiris dan heuristik sejauh memperkirakan upaya pemeliharaan perangkat lunak yang bersangkutan.

3.6 TEKNIK ESTIMASI EMPIRIS

Teknik estimasi empiris, selama bertahun-tahun, telah diformalkan sampai batas tertentu. Namun, ini pada dasarnya masih eufemisme untuk pekerjaan tebakan murni. Teknik-teknik ini mudah digunakan dan memberikan perkiraan yang cukup akurat. Dua teknik estimasi empiris yang populer adalah—Expert judgement dan teknik estimasi Delphi.

Penilaian Ahli

Penilaian ahli adalah teknik estimasi ukuran yang banyak digunakan. Dalam teknik ini, seorang ahli membuat tebakan terdidik tentang ukuran masalah setelah menganalisis masalah secara menyeluruh. Biasanya, ahli memperkirakan biaya komponen yang berbeda (yaitu modul atau subsistem) yang akan membentuk sistem dan kemudian menggabungkan perkiraan untuk masing-masing modul untuk sampai pada perkiraan keseluruhan. Namun, teknik ini menderita beberapa kekurangan. Hasil dari teknik penilaian ahli tunduk pada kesalahan manusia dan bias individu. Juga, ada kemungkinan bahwa seorang ahli mungkin mengabaikan beberapa faktor secara tidak sengaja. Selanjutnya, seorang ahli yang membuat perkiraan mungkin tidak memiliki pengalaman dan pengetahuan yang relevan tentang semua aspek proyek. Misalnya, dia mungkin fasih dengan bagian database dan antarmuka pengguna, tetapi mungkin tidak terlalu paham tentang bagian komunikasi komputer. Karena faktor-faktor ini, estimasi ukuran yang dicapai oleh penilaian seorang ahli tunggal mungkin jauh dari akurat.

Bentuk penilaian ahli yang lebih halus adalah estimasi yang dibuat oleh sekelompok ahli. Kemungkinan kesalahan yang timbul dari masalah seperti pengawasan individu, kurangnya keakraban dengan aspek tertentu dari suatu proyek, bias pribadi, dan keinginan untuk memenangkan kontrak melalui perkiraan yang terlalu optimis diminimalkan ketika estimasi dilakukan oleh sekelompok ahli. Namun, perkiraan yang dibuat oleh sekelompok ahli mungkin masih menunjukkan bias. Misalnya, pada isu-isu tertentu seluruh kelompok ahli mungkin bias karena alasan seperti yang timbul dari pertimbangan politik atau sosial. Kelemahan penting lainnya dari teknik penilaian ahli adalah bahwa keputusan yang dibuat oleh suatu kelompok mungkin didominasi oleh anggota yang terlalu asertif.

Estimasi Biaya Delphi

Teknik estimasi biaya Delphi mencoba mengatasi beberapa kekurangan dari pendekatan expert judgement. Estimasi Delphi dilakukan oleh tim yang terdiri dari sekelompok ahli dan seorang koordinator. Dalam pendekatan ini, koordinator memberikan setiap penaksir salinan dokumen spesifikasi kebutuhan perangkat lunak (SRS) dan formulir untuk mencatat perkiraan biayanya. Penaksir menyelesaikan perkiraan individu mereka secara anonim dan menyerahkannya kepada koordinator. Dalam estimasi mereka, penduga menyebutkan karakteristik produk yang tidak biasa yang mempengaruhi estimasi mereka. Koordinator menyiapkan ringkasan tanggapan dari semua penaksir, dan juga memasukkan alasan yang tidak biasa yang dicatat oleh salah satu penaksir. Ringkasan informasi yang disiapkan didistribusikan ke penduga.

Berdasarkan ringkasan ini, estimator melakukan estimasi ulang. Proses ini diulang untuk beberapa putaran. Namun, tidak ada diskusi di antara para penaksir yang diperbolehkan selama seluruh proses estimasi. Maksud di balik pembatasan ini adalah bahwa jika ada diskusi yang diperbolehkan di antara para penaksir, maka banyak penaksir dapat dengan mudah dipengaruhi oleh alasan seorang penaksir yang mungkin lebih berpengalaman atau senior. Setelah menyelesaikan beberapa iterasi estimasi, koordinator bertanggung jawab untuk

mengumpulkan hasil dan menyiapkan estimasi akhir. Estimasi Delphi, meskipun menghabiskan lebih banyak waktu dan usaha, mengatasi kelemahan penting dari teknik penilaian ahli karena hasilnya tidak dapat dipengaruhi secara tidak adil oleh anggota yang terlalu asertif dan senior.

3.7 COMO—TEKNIK ESTIMASI HEURISTIS

Model estimasi biaya konstruktif (COCOMO) diusulkan oleh Boehm [1981]. COCOMO menetapkan proses tiga tahap untuk estimasi proyek. Pada tahap pertama, perkiraan awal tiba. Selama dua tahap berikutnya, perkiraan awal disempurnakan untuk sampai pada perkiraan yang lebih akurat. COCOMO menggunakan model estimasi tunggal dan multivariabel pada tahap estimasi yang berbeda. Tiga tahapan teknik estimasi COCOMO adalah—COCOMO dasar, COCOMO intermediate, dan COCOMO lengkap.

Model COCOMO Dasar

Boehm mendalilkan bahwa setiap proyek pengembangan perangkat lunak dapat diklasifikasikan ke dalam salah satu dari tiga kategori berikut berdasarkan kompleksitas pengembangan-organik, semi-terpisah, dan tertanam. Berdasarkan kategori proyek pengembangan perangkat lunak, ia memberikan serangkaian rumus yang berbeda untuk memperkirakan upaya dan durasi dari perkiraan ukuran.

Tiga kelas dasar proyek pengembangan perangkat lunak

Untuk mengklasifikasikan proyek ke dalam kategori yang diidentifikasi, Boehm mengharuskan kita untuk mempertimbangkan tidak hanya karakteristik produk tetapi juga karakteristik tim pengembangan dan lingkungan pengembangan. Secara kasar, tiga kelas pengembangan produk sesuai dengan pengembangan aplikasi, utilitas, dan perangkat lunak sistem. Biasanya, program pemrosesan data1 dianggap sebagai program aplikasi. Compiler, linker, dll, adalah program utilitas. Sistem operasi dan program sistem waktu nyata, dll. adalah program sistem. Program sistem berinteraksi langsung dengan perangkat keras dan kompleksitas pemrograman juga muncul dari persyaratan untuk memenuhi batasan waktu dan pemrosesan tugas secara bersamaan.

Brooks [1975] menyatakan bahwa program utilitas kira-kira tiga kali lebih sulit untuk ditulis sebagai program aplikasi dan program sistem kira-kira tiga kali lebih sulit daripada program utilitas. Jadi menurut Brooks, tingkat relatif kompleksitas pengembangan produk untuk tiga kategori (aplikasi, utilitas dan program sistem) produk adalah 1:3:9. Definisi Boehm [1981] tentang perangkat lunak organik, semi-terpisah, dan tertanam diuraikan sebagai berikut:

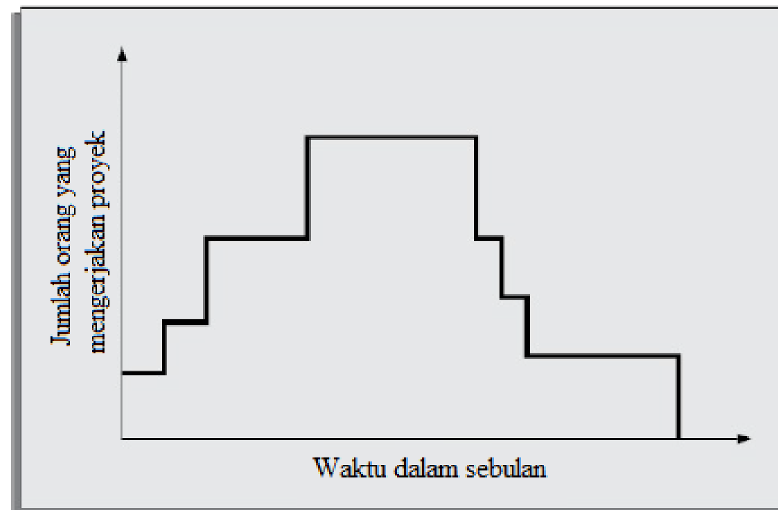
Organik: kita dapat mengklasifikasikan proyek pengembangan menjadi tipe organik, jika proyek tersebut berkaitan dengan pengembangan program aplikasi yang dipahami dengan baik, ukuran tim pengembangan cukup kecil, dan anggota tim berpengalaman dalam mengembangkan jenis proyek serupa.

Semidetached: Sebuah proyek pengembangan dapat diklasifikasikan menjadi tipe semidetached, jika tim pengembangan terdiri dari campuran staf yang berpengalaman dan tidak berpengalaman. Anggota tim mungkin memiliki pengalaman yang terbatas pada sistem terkait tetapi mungkin tidak terbiasa dengan beberapa aspek dari sistem yang sedang dikembangkan.

Embedded: Sebuah proyek pengembangan dianggap tipe tertanam, jika perangkat lunak yang dikembangkan sangat digabungkan dengan perangkat keras, atau jika ada peraturan ketat tentang prosedur operasional. Anggota tim mungkin memiliki pengalaman yang terbatas pada sistem terkait tetapi mungkin tidak terbiasa dengan beberapa aspek dari sistem yang sedang dikembangkan.

Perhatikan bahwa dalam menentukan kategori proyek pengembangan, selain mempertimbangkan karakteristik produk yang dikembangkan, kita perlu mempertimbangkan karakteristik anggota tim. Dengan demikian, program pemrosesan data sederhana dapat diklasifikasikan sebagai semi-terpisah, jika anggota tim tidak berpengalaman dalam pengembangan produk serupa.

Untuk tiga kategori produk, Boehm menyediakan rangkaian ekspresi yang berbeda untuk memprediksi upaya (dalam satuan orang-bulan) dan waktu pengembangan dari estimasi ukuran yang diberikan dalam kilo baris kode sumber (KLSC). Tapi, berapa banyak usaha satu orang-bulan? Satu orang per bulan adalah upaya yang biasanya dilakukan seseorang dalam sebulan. Perkiraan orang-bulan secara implisit memperhitungkan kerugian produktivitas yang biasanya terjadi karena hilangnya waktu di hari libur, libur mingguan, rehat kopi, dll.]



Gambar 3.3 Kurva orang-bulan.

Apa itu Person-Month?

Orang-bulan (PM) adalah unit populer untuk pengukuran usaha. Person-month (PM) dianggap sebagai unit yang tepat untuk mengukur upaya, karena developer biasanya ditugaskan ke sebuah proyek selama beberapa bulan tertentu. Perlu dicatat bahwa perkiraan upaya 100 PM tidak berarti bahwa 100 orang harus bekerja selama 1 bulan. Juga tidak berarti bahwa 1 orang harus dipekerjakan selama 100 bulan untuk menyelesaikan proyek. Estimasi usaha hanya menunjukkan area di bawah kurva orang-bulan (lihat Gambar 3.3) untuk proyek tersebut. Plot pada Gambar 3.3 menunjukkan bahwa jumlah personel yang berbeda dapat bekerja pada titik yang berbeda dalam pengembangan proyek. Jumlah personel yang bekerja pada proyek biasanya bertambah atau berkurang dengan jumlah yang tidak terpisahkan, menghasilkan tepian yang tajam di plot.

Bentuk umum dari ekspresi COCOMO

Model COCOMO dasar adalah model heuristik variabel tunggal yang memberikan perkiraan perkiraan parameter proyek. Model estimasi COCOMO dasar diberikan oleh ekspresi dari bentuk-bentuk berikut:

Usaha = a1 × (KLOC)a2 PM

Tdev = b1 × (Usaha)b2 bulan

di mana,

- KLOC adalah perkiraan ukuran produk perangkat lunak yang dinyatakan dalam Kilo Lines Of Code.
- a1, a2, b1, b2 adalah konstanta untuk setiap kategori produk perangkat lunak.

- Tdev adalah perkiraan waktu untuk mengembangkan perangkat lunak, dinyatakan dalam bulan.
- Upaya adalah upaya total yang diperlukan untuk mengembangkan produk perangkat lunak, dinyatakan dalam person-months (PM)

Menurut Boehm, setiap baris teks sumber harus dihitung sebagai satu LOC terlepas dari jumlah instruksi sebenarnya pada baris itu. Jadi, jika satu instruksi mencakup beberapa baris (katakanlah n baris), itu dianggap sebagai nLOC. Nilai a1, a2, b1, b2 untuk berbagai kategori produk seperti yang diberikan oleh Boehm [1981] diringkas di bawah ini. Dia memperoleh nilai-nilai ini dengan memeriksa data historis yang dikumpulkan dari sejumlah besar proyek aktual. Estimasi upaya pengembangan: Untuk tiga kelas produk perangkat lunak, rumus untuk memperkirakan upaya berdasarkan ukuran kode ditunjukkan di bawah ini:

Organik : Upaya = 2,4(KLOC)1,05 PM

Semi-terpisah : Upaya = 3.0(KLOC)1.12 PM

Tertanam : Upaya = 3.6(KLOC)1.20 PM

Estimasi waktu pengembangan: Untuk ketiga kelas produk perangkat lunak, rumus untuk memperkirakan waktu pengembangan berdasarkan upaya diberikan di bawah ini:

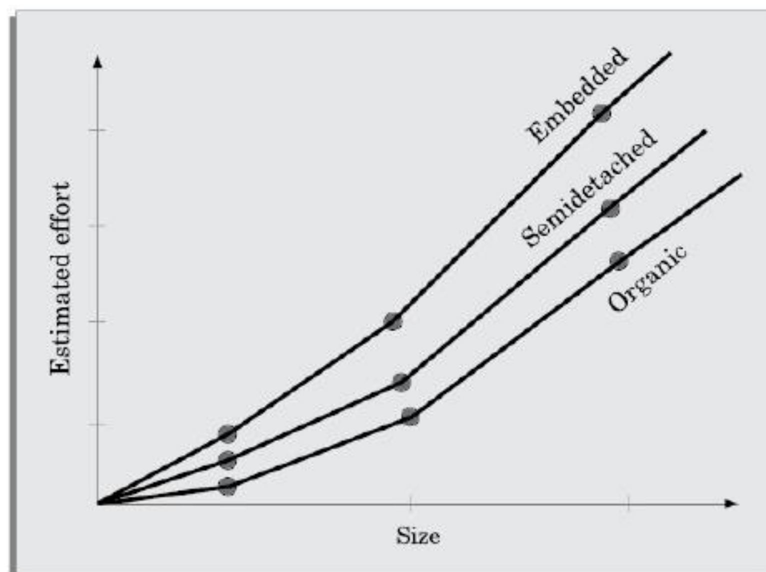
Organik : Tdev = 2.5(Effort)0.38 Bulan

Semi-terpisah : Tdev = 2.5(Effort)0.35 Bulan

Tertanam : Tdev = 2.5(Effort)0.32 Bulan

Kita dapat memperoleh beberapa wawasan tentang model COCOMO dasar, jika kita memplot perkiraan nilai upaya dan durasi untuk ukuran perangkat lunak yang berbeda. Gambar 3.4 menunjukkan plot perkiraan upaya versus ukuran produk untuk berbagai kategori produk perangkat lunak.

Pengamatan dari plot ukuran usaha Dari Gambar 3.4, kita dapat mengamati bahwa usaha adalah beberapa yang superlinear (yaitu, kemiringan kurva>1) dalam ukuran produk perangkat lunak.



Gambar 3.4 Upaya versus ukuran produk.

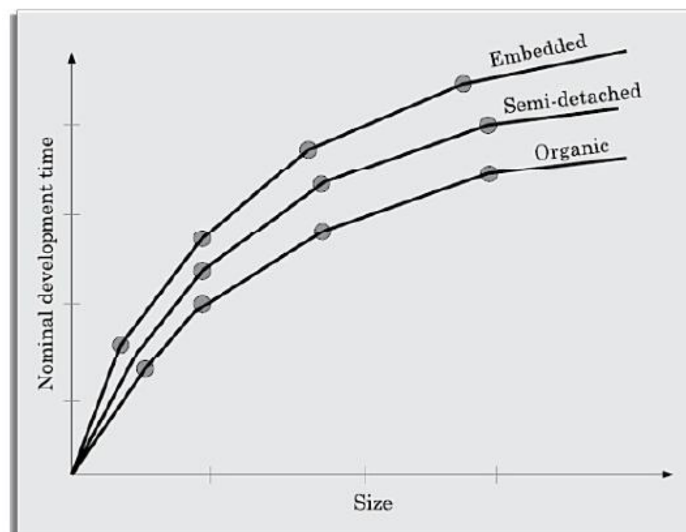
Hal ini karena eksponen dalam ekspresi usaha lebih dari 1. Dengan demikian, usaha yang dibutuhkan untuk mengembangkan produk meningkat pesat dengan ukuran proyek. Namun, perhatikan bahwa peningkatan upaya dengan ukuran tidak seburuk yang digambarkan dalam Bab 1. Alasannya adalah karena COCOMO mengasumsikan bahwa proyek

dirancang dan dikembangkan dengan hati-hati dengan menggunakan prinsip-prinsip rekayasa perangkat lunak.

Pengamatan dari waktu pengembangan—ukuran plot

Waktu pengembangan versus ukuran produk di KLOC diplot pada Gambar 3.5. Dari Gambar 3.5, kita dapat mengamati hal berikut:

- Waktu pengembangan adalah fungsi sublinier dari ukuran produk. Artinya, ketika ukuran produk meningkat dua kali lipat, waktu untuk mengembangkan produk tidak berlipat ganda tetapi meningkat secara moderat. Misalnya, untuk mengembangkan produk dua kali lebih besar dari produk ukuran 100KLOC, peningkatan durasi mungkin hanya 20 persen. Mungkin tampak mengejutkan bahwa kurva durasi tidak meningkat secara superlinear—seseorang biasanya mengharapkan kurva berperilaku serupa dengan yang ada di plot ukuran usaha. Anomali yang tampak ini dapat dijelaskan oleh fakta bahwa COCOMO mengasumsikan bahwa pengembangan proyek tidak dilakukan oleh satu orang tetapi oleh tim pengembang.
- Dari Gambar 3.5 kita dapat mengamati bahwa untuk proyek dengan ukuran tertentu, waktu pengembangan kira-kira sama untuk ketiga kategori produk. Misalnya, program 60 KLOC dapat dikembangkan dalam waktu sekitar 18 bulan, terlepas dari apakah itu jenis organik, semi-terpisah, atau tertanam. (Harap verifikasi ini menggunakan rumus dasar COCOMO yang dibahas di bagian ini). Namun, menurut rumus COCOMO, program yang disematkan membutuhkan upaya yang jauh lebih tinggi daripada program aplikasi atau utilitas. Kita dapat mengartikannya bahwa ada lebih banyak ruang untuk kegiatan paralel untuk program sistem daripada di program utilitas atau aplikasi.



Gambar 3.5 Waktu pengembangan versus ukuran.

Perkiraan biaya

Dari estimasi tenaga, biaya proyek dapat diperoleh dengan mengalikan estimasi tenaga (dalam man-month) dengan biaya tenaga kerja per bulan. Tersirat dalam perhitungan biaya proyek ini adalah asumsi bahwa

seluruh biaya proyek dikeluarkan karena biaya tenaga kerja saja. Namun, selain biaya tenaga kerja, suatu proyek akan dikenakan beberapa jenis biaya lain yang akan kita sebut sebagai biaya overhead. Biaya overhead akan mencakup biaya karena perangkat keras dan

perangkat lunak yang diperlukan untuk proyek dan biaya overhead perusahaan untuk administrasi, ruang kantor, listrik, dll. Tergantung pada nilai yang diharapkan dari biaya overhead, manajer proyek harus meningkatkan skala yang sesuai biaya tiba dengan menggunakan rumus COCOMO.

Implikasi dari upaya dan perkiraan durasi

Implikasi implisit penting dari perkiraan COCOMO adalah jika Anda mencoba menyelesaikan proyek dalam waktu yang lebih singkat dari perkiraan durasi, maka biayanya akan meningkat secara drastis. Namun, jika Anda menyelesaikan proyek dalam jangka waktu yang lebih lama dari perkiraan, maka hampir tidak ada penurunan nilai perkiraan biaya. Dengan demikian, kita dapat mempertimbangkan bahwa COCOMO upaya dan nilai durasi untuk menunjukkan berikut ini. Nilai upaya dan durasi yang dihitung oleh COCOMO adalah nilai untuk menyelesaikan pekerjaan dalam waktu singkat tanpa terlalu meningkatkan biaya tenaga kerja.

Sekarang mari kita uraikan pernyataan di atas. Ketika tim proyek terdiri dari satu anggota, anggota tersebut tidak akan pernah menganggur karena kekurangan pekerjaan, tetapi proyek akan memakan waktu terlalu lama untuk diselesaikan. Di sisi lain, ketika ada terlalu banyak anggota, proyek akan selesai dalam waktu yang jauh lebih singkat, tetapi seringkali selama durasi proyek beberapa anggota harus menganggur karena kekurangan pekerjaan.

Durasi proyek seperti yang dihitung oleh model COCOMO, semua developer tetap sibuk dengan pekerjaan selama seluruh periode pengembangan. Setiap kali proyek harus diselesaikan dalam waktu yang lebih pendek dari perkiraan durasi dengan menggunakan COCOMO, beberapa waktu menganggur di pihak developer akan ada. Waktu menganggur seperti itu akan menghasilkan peningkatan biaya pengembangan. Sebuah tim berukuran optimal untuk sebuah proyek adalah tim di mana setiap developer setiap saat selama pengembangan tidak duduk diam menunggu pekerjaan, tetapi pada saat yang sama terdiri dari anggota sebanyak mungkin untuk mengurangi waktu pengembangan. Kita bisa menganggap durasi yang diberikan oleh COCOMO disebut sebagai durasi optimal. Disebut durasi optimal, jika proyek diupayakan selesai dalam waktu yang lebih singkat, maka upaya yang dibutuhkan akan meningkat dengan cepat. Ini mungkin tampak sebagai paradoks—bagaimanapun juga, ini adalah produk yang sama yang akan dikembangkan, meskipun dalam waktu yang lebih singkat, lalu mengapa upaya yang diperlukan harus meningkat dengan cepat? Hal ini dapat dijelaskan dengan fakta bahwa untuk setiap produk pada titik mana pun selama pengembangan proyek, ada batasan jumlah aktivitas paralel yang dapat diidentifikasi dan dilaksanakan secara bermakna. Jadi, jika lebih banyak developer dikerahkan daripada ukuran optimal, beberapa developer harus menganggur, karena pada titik tertentu dalam pengembangan atau lainnya, tidak mungkin untuk menugaskan mereka pekerjaan sama sekali. Waktu menganggur ini akan muncul sebagai upaya yang lebih tinggi dan biaya yang lebih besar.

Estimasi ukuran staf

Mengingat estimasi untuk upaya pengembangan proyek dan waktu pengembangan nominal, dapatkah tingkat staf yang dibutuhkan ditentukan dengan pembagian sederhana dari estimasi upaya dengan estimasi durasi? Jawabannya adalah tidak". Ini akan menjadi resep yang sempurna untuk penundaan proyek dan kelebihan biaya.

Contoh 3.2 Asumsikan bahwa ukuran produk perangkat lunak tipe organik telah diperkirakan 32.000 baris kode sumber. Asumsikan bahwa gaji rata-rata seorang developer perangkat lunak adalah Rp. 15.000.000 per bulan. Tentukan upaya yang diperlukan untuk mengembangkan produk perangkat lunak, waktu pengembangan nominal, dan biaya untuk

mengembangkan produk. Dari rumus dasar estimasi COCOMO untuk perangkat lunak organik: Upaya = 2,4 × (32)1,05 = 91 PM. Waktu pengembangan nominal = 2,5 × (91)0,38 = 14 bulan Biaya staf yang dibutuhkan untuk mengembangkan produk = 91 × Rp. 15.000.000 = Rp. 1.465.000.000

COCOMO Menengah

Model dasar COCOMO mengasumsikan bahwa usaha dan waktu pengembangan adalah fungsi dari ukuran produk saja. Namun, sejumlah parameter proyek lain selain ukuran produk mempengaruhi upaya serta waktu yang dibutuhkan untuk mengembangkan produk. Misalnya upaya untuk mengembangkan suatu produk akan bervariasi tergantung pada kecanggihan lingkungan pengembangan. Oleh karena itu, untuk mendapatkan estimasi yang akurat dari upaya dan durasi proyek, efek dari semua parameter yang relevan harus diperhitungkan. Model COCOMO perantara mengakui fakta ini dan menyempurnakan perkiraan awal. Model COCOMO perantara menyempurnakan perkiraan awal yang diperoleh dengan menggunakan ekspresi COCOMO dasar dengan meningkatkan atau menurunkan perkiraan berdasarkan evaluasi serangkaian atribut pengembangan perangkat lunak.

Model COCOMO perantara menggunakan satu set 15 driver biaya (pengganda) yang ditentukan berdasarkan berbagai atribut pengembangan perangkat lunak. Penggerak biaya ini dikalikan dengan perkiraan biaya dan upaya awal (diperoleh dari COCOMO dasar) untuk menaikkan atau menurunkannya secara tepat. Misalnya, jika praktik pemrograman modern digunakan, perkiraan awal diperkecil dengan perkalian dengan pemicu biaya yang memiliki nilai kurang dari 1. Jika ada persyaratan keandalan yang ketat pada produk perangkat lunak, perkiraan awal ditingkatkan. Boehm mengharuskan manajer proyek untuk menilai 15 parameter yang berbeda untuk proyek tertentu pada skala satu sampai tiga. Untuk setiap penilaian parameter proyek seperti itu, dia telah menyarankan pemicu biaya (atau pengganda) yang sesuai untuk menyempurnakan perkiraan awal. Secara umum, pemicu biaya yang diidentifikasi oleh Boehm dapat diklasifikasikan sebagai atribut dari item berikut:

- **Produk**: Karakteristik produk yang dipertimbangkan meliputi kompleksitas yang melekat pada produk, persyaratan keandalan produk, dll.
- **Komputer**: Karakteristik komputer yang dipertimbangkan termasuk kecepatan eksekusi yang dibutuhkan, ruang penyimpanan yang dibutuhkan, dll.
- **Personil**: Atribut personel pengembangan yang dipertimbangkan meliputi tingkat pengalaman personel, kemampuan pemrograman mereka, kemampuan analisis, dll.
- **Lingkungan pengembangan**: Atribut lingkungan pengembangan menangkap fasilitas pengembangan yang tersedia untuk pengembang. Parameter penting yang diperhatikan adalah kecanggihan alat otomatisasi (CASE) yang digunakan untuk pengembangan perangkat lunak.

Kita hanya membahas ide-ide dasar di balik model COCOMO perantara. Sebuah diskusi rinci tentang model COCOMO menengah berada di luar cakupan buku ini dan pembaca yang tertarik dapat merujuk [Boehm81].

COCOMO Lengkap

Kelemahan utama dari model COCOMO dasar dan menengah adalah bahwa mereka menganggap produk perangkat lunak sebagai satu kesatuan yang homogen. Namun, sebagian besar sistem besar terdiri dari beberapa sub-sistem yang lebih kecil. Sub-sistem ini seringkali memiliki karakteristik yang sangat berbeda. Sebagai contoh, beberapa sub-sistem dapat dianggap sebagai tipe organik, beberapa semi-terpisah, dan beberapa bahkan tertanam. Tidak hanya kompleksitas pengembangan yang melekat pada subsistem yang berbeda, tetapi untuk beberapa subsistem persyaratan keandalan mungkin tinggi, untuk beberapa tim

pengembangan mungkin tidak memiliki pengalaman pengembangan serupa sebelumnya, dan seterusnya.

Model COCOMO yang lengkap mempertimbangkan perbedaan karakteristik subsistem ini dan memperkirakan upaya dan waktu pengembangan sebagai jumlah perkiraan untuk masing-masing subsistem. Dengan kata lain, biaya untuk mengembangkan setiap sub-sistem diperkirakan secara terpisah, dan biaya sistem yang lengkap ditentukan sebagai biaya subsistem. Pendekatan ini mengurangi margin kesalahan dalam estimasi akhir. Mari kita perhatikan proyek pengembangan berikut sebagai contoh penerapan model COCOMO yang lengkap. Produk sistem informasi manajemen terdistribusi (MIS) untuk organisasi yang memiliki kantor di beberapa tempat di seluruh negeri dapat memiliki sub-komponen berikut:

- Bagian basis data
- Bagian antarmuka pengguna grafis (GUI)
- Bagian komunikasi

Dari jumlah tersebut, bagian komunikasi dapat dianggap sebagai perangkat lunak tertanam. Bagian database dapat berupa perangkat lunak semi-terpisah, dan bagian GUI perangkat lunak organik. Biaya untuk ketiga komponen ini dapat diperkirakan secara terpisah, dan dijumlahkan untuk memberikan biaya keseluruhan sistem.

Untuk lebih meningkatkan akurasi hasil, nilai parameter yang berbeda dari model dapat disesuaikan dan divalidasi terhadap database proyek historis organisasi untuk mendapatkan estimasi yang lebih akurat. Model estimasi seperti COCOMO tidak sepenuhnya akurat dan tidak memiliki justifikasi ilmiah yang lengkap. Namun, model estimasi biaya perangkat lunak seperti COCOMO diperlukan untuk pendekatan rekayasa untuk manajemen proyek perangkat lunak. Perusahaan menganggap perkiraan biaya yang dihitung memuaskan, jika ini berada dalam kisaran sekitar 80 persen dari biaya akhir. Meskipun perkiraan ini adalah perkiraan kasar — tanpa model seperti itu, seseorang hanya memiliki penilaian subjektif untuk diandalkan.

3.8 COCOMO 2

Sejak model estimasi COCOMO diusulkan pada awal 1980-an, paradigma pengembangan perangkat lunak serta karakteristik proyek pengembangan telah mengalami perubahan besar. Proyek perangkat lunak saat ini berukuran jauh lebih besar dan penggunaan kembali perangkat lunak yang ada untuk mengembangkan produk baru telah meluas. Misalnya, pengembangan berbasis komponen dan arsitektur berorientasi layanan (SoA) telah menjadi sangat populer (dibahas dalam Bab 15). Model siklus hidup baru dan paradigma pengembangan sedang digunakan untuk perangkat lunak berbasis web dan berbasis komponen. Selama tahun 1980-an jarang ada program yang interaktif, dan antarmuka pengguna grafis hampir tidak ada. Di sisi lain, produk perangkat lunak saat ini sangat interaktif dan mendukung antarmuka pengguna grafis yang rumit. Upaya yang dihabiskan untuk mengembangkan bagian GUI seringkali sama dengan upaya yang dihabiskan untuk mengembangkan fungsionalitas perangkat lunak yang sebenarnya. Untuk membuat COCOMO cocok dalam skenario yang diubah, Boehm mengusulkan COCOMO 2 [Boehm95] pada tahun 1995.

COCOMO 2 menyediakan tiga model untuk mencapai estimasi biaya yang semakin akurat. Ini dapat digunakan untuk memperkirakan biaya proyek pada fase yang berbeda dari produk perangkat lunak. Sebagai proyek berlangsung, model ini dapat diterapkan pada tahap yang berbeda dari proyek yang sama.

- **Model komposisi aplikasi**: Model ini seperti namanya, dapat digunakan untuk memperkirakan biaya pengembangan prototipe.

- **Model desain awal:** Ini mendukung estimasi biaya pada tahap desain arsitektur.
- **Model pasca-arsitektur**: Ini memberikan estimasi biaya selama tahap desain dan pengkodean terperinci.

Model pasca-arsitektur dapat dianggap sebagai pembaruan dari COCOMO asli. Dua model lainnya membantu mempertimbangkan dua faktor berikut. Sekarang hari setiap perangkat lunak interaktif dan GUI-driven. Pengembangan GUI merupakan bagian penting dari upaya pengembangan secara keseluruhan. Faktor kedua menyangkut beberapa masalah yang mempengaruhi produktivitas seperti tingkat penggunaan kembali.

Model komposisi aplikasi

Model komposisi aplikasi didasarkan pada penghitungan jumlah layar, laporan, dan modul (komponen). Masing-masing komponen ini dianggap sebagai objek (ini tidak ada hubungannya dengan konsep objek dalam paradigma berorientasi objek). Ini digunakan untuk menghitung titik objek aplikasi. Upaya diperkirakan dalam model komposisi aplikasi sebagai berikut:

1. Perkirakan jumlah layar, laporan, dan modul (komponen) dari analisis dokumen SRS.
2. Tentukan tingkat kerumitan setiap layar dan laporan, dan beri peringkat sebagai sederhana, sedang, atau sulit. Kompleksitas layar atau laporan ditentukan oleh jumlah tabel dan tampilan yang dikandungnya.
3. Gunakan nilai bobot pada Tabel 3.3 hingga 3.5.
   Bobot telah dirancang agar sesuai dengan jumlah upaya yang diperlukan untuk mengimplementasikan instance objek pada kelas kompleksitas yang ditetapkan.

Tabel 3.3 Penetapan Kompleksitas LAYAR untuk Tabel Data

Jumlah View	**Tabel < 4**	**Tabel < 8**	**Tabel ≥**
<3	Simpel	Simpel	Medium
3-7	Simple	Medium	Sulit
>8	Medium	Sulit	Sulit

Tabel 3.4 Tugas Kompleksitas Laporan untuk Tabel Data

Jumlah View	**Tabel < 4**	**Tabel < 8**	**Tabel ≥**
0 atau 1	Simpel	Simpel	Medium
2 atau 3	Simple	Medium	Sulit
4 atau lebih	Medium	Sulit	Sulit

4. Tambahkan semua nilai kompleksitas yang ditetapkan untuk instance objek bersama-sama untuk mendapatkan titik objek.

Tabel 3.5 Tabel Bobot Kompleksitas untuk Setiap Kelas untuk Setiap Jenis Objek

Jenis obyek	**Simpel**	**Medium**	**Sulit**
Layar	1	2	3
Laporan	2	5	8
Komponen 3GL	-	-	10

5. Perkirakan persentase penggunaan kembali yang diharapkan dalam sistem. Perhatikan bahwa penggunaan kembali mengacu pada jumlah perangkat lunak yang telah dikembangkan sebelumnya yang akan digunakan dalam sistem. Kemudian, evaluasi penghitungan New Object-Point (NOP) sebagai berikut,
$$NOP = \frac{(Objek - Poin)(100 - \%\ yang\ digunakan)}{100}$$
6. Tentukan produktivitas menggunakan Tabel 3.6. Produktivitas tergantung pada pengalaman developer serta kematangan lingkungan CASE yang digunakan.
7. Akhirnya, perkiraan usaha dalam orang-bulan dihitung sebagai E = NOP/PROD.

Tabel 3.6 Tabel Produktivitas

Pengalaman Developer	Sangat rendah	Rendah	Nominal	Tinggi	Sangat tinggi
Kematangan KASUS	Sangat rendah	Rendah	Nominal	Tinggi	Sangat tinggi
PRODUKTIVITAS	4	7	13	25	50

3.9 ILMU PERANGKAT LUNAK HALSTEAD—TEKNIK ANALITIS

Ilmu perangkat lunak Halstead adalah teknik analisis untuk mengukur ukuran, upaya pengembangan, dan biaya pengembangan produk perangkat lunak. Halstead menggunakan beberapa parameter program primitif untuk mengembangkan ekspresi untuk semua panjang program, volume minimum potensial, volume aktual, tingkat bahasa, upaya, dan waktu pengembangan. Untuk program tertentu, biarkan:

- h_1 adalah jumlah operator unik yang digunakan dalam program,
- h_2 adalah jumlah operan unik yang digunakan dalam program,
- N_1 adalah jumlah total operator yang digunakan dalam program,
- N_2 menjadi jumlah total operand yang digunakan dalam program.

Meskipun istilah operator dan operan memiliki arti intuitif, definisi yang tepat dari istilah ini diperlukan untuk menghindari ambiguitas. Namun, sayangnya kita tidak akan dapat memberikan definisi yang tepat dari kedua istilah tersebut. Tidak ada kesepakatan umum di antara para peneliti tentang cara yang paling berarti untuk mendefinisikan operator dan operan untuk bahasa pemrograman yang berbeda. Namun, beberapa panduan umum mengenai identifikasi operator dan operan untuk bahasa pemrograman apa pun dapat disediakan. Misalnya, penugasan, aritmatika, dan operator logika biasanya dihitung sebagai operator. Sepasang kurung, serta blok awal —pasangan akhir blok, dianggap sebagai operator tunggal. Label dianggap sebagai operator, jika digunakan sebagai target pernyataan GOTO. Konstruksi if ... then ... else ... endif dan while ... do dianggap sebagai operator tunggal. Operator urutan (penghentian pernyataan) ';' dianggap sebagai operator tunggal. Deklarasi subrutin dan deklarasi variabel terdiri dari operand. Nama fungsi dalam pernyataan pemanggilan fungsi dianggap sebagai operator, dan argumen pemanggilan fungsi dianggap sebagai operan. Namun, daftar parameter fungsi dalam pernyataan deklarasi fungsi tidak dianggap sebagai operan. Dibawah ini saya cantumkan himpunan operator dan operan untuk bahasa ANSIC. Namun, harus disadari bahwa ada ketidaksepakatan yang cukup besar di antara berbagai peneliti dalam hal ini.

Operator dan Operan untuk bahasa ANSI C

Berikut ini adalah daftar operator yang disarankan untuk bahasa ANSI C:

([. , -> * + - ~ ! ++ -- * / % + - << >> < > <= >= != == & ^ | && || = *= /= %= += -= <<= >>= &= ^= |= : ? { ; CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN and a function name in a function call

Operand adalah variabel dan konstanta yang digunakan dengan operator dalam ekspresi. Perhatikan bahwa nama variabel yang muncul dalam deklarasi tidak dianggap sebagai operan.

Contoh 3.3 Perhatikan ekspresi a = &b; a, b adalah operand dan =, & adalah operator.

Contoh 3.4 Nama fungsi dalam definisi fungsi tidak dihitung sebagai operator.

```
int func ( int a, int b )
{
        . . .
}
```

Untuk kode contoh di atas, operatornya adalah: {}, ( ) Saya tidak menganggap fungsi c, a, dan b sebagai operan, karena ini adalah bagian dari definisi fungsi.

Contoh 3.5 Perhatikan pernyataan pemanggilan fungsi: func (a, b);. Dalam hal ini, fungsi ' ,' a n d ; dianggap sebagai operator dan variabel a, b diperlakukan sebagai operan.

Panjang dan Kosakata

Panjang program seperti yang didefinisikan oleh Halstead, mengkuantifikasi penggunaan total semua operator dan operan dalam program. Jadi, panjang $N = N_1 + N_2$. Definisi Halstead tentang panjang program sebagai jumlah total operator dan operan secara kasar setuju dengan gagasan intuitif tentang panjang program sebagai jumlah total token yang digunakan dalam program. osakata program adalah jumlah operator dan operan unik yang digunakan dalam program. Jadi, kosakata program $h = h_1 + h_2$.

Volume Program

Panjang program (yaitu, jumlah total operator dan operan yang digunakan dalam kode) tergantung pada pilihan operator dan operan yang digunakan. Dengan kata lain, untuk masalah pemrograman yang sama, panjangnya akan tergantung pada gaya pemrograman. Jenis ketergantungan ini akan menghasilkan ukuran panjang yang berbeda untuk masalah yang pada dasarnya sama ketika bahasa pemrograman yang berbeda digunakan. Jadi, saat menyatakan ukuran program, bahasa pemrograman yang digunakan harus dipertimbangkan:

$$V = N \log_2 h$$

Mari kita coba memahami ide penting di balik ungkapan ini. Secara intuitif, volume program V adalah jumlah bit minimum yang diperlukan untuk mengkodekan program. Sebenarnya, untuk mewakili h pengidentifikasi yang berbeda secara unik, kita membutuhkan setidaknya log2 h bit (di mana h adalah kosakata program). Dalam skema ini, kita membutuhkan N log2 h bit untuk menyimpan program dengan panjang N. Oleh karena itu, volume V mewakili ukuran program dengan kira-kira mengkompensasi efek dari bahasa pemrograman yang digunakan.

Potensi Volume Minimum

Volume minimum potensial V* didefinisikan sebagai volume program yang paling ringkas di mana suatu masalah dapat dikodekan. Volume minimum diperoleh ketika program dapat diekspresikan menggunakan instruksi kode sumber tunggal, misalnya pemanggilan fungsi seperti foo();. Dengan kata lain, volume terikat dari bawah karena fakta bahwa sebuah program akan memiliki setidaknya dua operator dan tidak kurang dari jumlah operan yang diperlukan. Perhatikan bahwa operan adalah item data input dan output. Jadi, jika suatu algoritma beroperasi pada data input dan output d1, d2, ... dn, program yang paling ringkas adalah $f(d_1, d_2, ..., d_n)$; dimana, $h_1 = 2$, $h_2 = n$. Oleh karena itu, $V^* = (2 + h_2) \log_2 (2 + h_2)$. Tingkat program L diberikan oleh L = V*/V. Konsep tingkat program L telah diperkenalkan dalam upaya

untuk mengukur tingkat abstraksi yang disediakan oleh bahasa pemrograman. Dengan menggunakan definisi ini, bahasa dapat diurutkan ke dalam level yang juga tampak benar secara intuitif.

Hasil di atas menyiratkan bahwa semakin tinggi level suatu bahasa, semakin sedikit upaya yang diperlukan untuk mengembangkan program menggunakan bahasa tersebut. Hasil ini sesuai dengan gagasan intuitif bahwa dibutuhkan lebih banyak upaya untuk mengembangkan program dalam bahasa rakitan daripada mengembangkan program dalam bahasa tingkat tinggi untuk memecahkan masalah.

Usaha dan Waktu

Upaya yang diperlukan untuk mengembangkan program dapat diperoleh dengan membagi volume program dengan tingkat bahasa pemrograman yang digunakan untuk mengembangkan kode. Jadi, usaha E = V /L, di mana E adalah jumlah diskriminasi mental yang diperlukan untuk melaksanakan program dan juga usaha yang diperlukan untuk membaca dan memahami program. Dengan demikian, upaya pemrograman E = V2/V* (karena L = V*/V) bervariasi sebagai kuadrat volume. Pengalaman menunjukkan bahwa E berkorelasi baik dengan upaya yang diperlukan untuk pemeliharaan program yang ada. Waktu programmer T = E/S, di mana S adalah kecepatan diskriminasi mental. Nilai S telah dikembangkan secara empiris dari penalaran psikologis, dan nilai yang direkomendasikan untuk aplikasi pemrograman adalah 18.

Estimasi Panjang

Meskipun panjang program dapat ditemukan dengan menghitung jumlah total operator dan operan dalam suatu program, Halstead menyarankan cara untuk menentukan panjang program dengan menggunakan jumlah operator dan operan unik yang digunakan dalam program. Dengan menggunakan metode ini, parameter program seperti panjang, volume, biaya, usaha, dll., dapat ditentukan bahkan sebelum dimulainya aktivitas pemrograman apa pun. Metodenya diringkas di bawah ini. Halstead berasumsi bahwa sangat tidak mungkin bahwa suatu program memiliki beberapa bagian yang identik—dalam terminologi bahasa formal, substring identik—dengan panjang lebih besar dari h(h menjadi kosakata program). Faktanya, begitu sepotong kode muncul secara identik di beberapa tempat, biasanya dibuat menjadi prosedur atau fungsi. Dengan demikian, kita dapat dengan aman mengasumsikan bahwa setiap program dengan panjang N terdiri dari N/h string unik dengan panjang h. Sekarang, merupakan hasil kombinatorial standar bahwa untuk sembarang alfabet dengan ukuran K, terdapat tepat Kr string yang berbeda dengan panjang r. Dengan demikian,

$$\frac{N}{\eta} \leq \eta^{\eta}$$

Atau

$$N \leq \eta^{\eta+1}$$

Karena operator dan operan biasanya bergantian dalam suatu program, kita dapat menyempurnakan batas atas lebih lanjut menjadi N hh1h1 h2h3. Selain itu, N harus mencakup tidak hanya himpunan elemen N yang terurut, tetapi juga harus mencakup semua kemungkinan subset dari himpunan terurut tersebut, yaitu himpunan daya dari N string (Alasan khusus Halstead ini sulit untuk dibenarkan!). Karena itu,

$$2^{N} = \eta\eta_1^{\eta 1}\eta_2^{\eta 2}$$

Atau mengambil logaritma di kedua sisi,

$$N = \log 2\,\eta + \log 2\left(\eta\eta_1^{\eta 1}\eta_2^{\eta 2}\right)$$

Jadi, kita mendapatkan,

$$N = \log 2\left(\eta\eta_1^{\eta 1}\eta_2^{\eta 2}\right) kira-kira, dengan\ mengabaikan \log 2\,\eta$$

Atau,

$$N = \log 2\eta_1^{\eta 1} + \log 2\eta_2^{\eta 2}$$
$$= \eta_1 \log 2\, \eta_1 + \eta_2 \log 2\eta_2$$

Bukti eksperimental yang dikumpulkan dari analisis sejumlah besar program menunjukkan bahwa panjang yang dihitung dan yang sebenarnya sangat cocok. Namun, hasilnya mungkin tidak akurat ketika program kecil dipertimbangkan secara individual.

Contoh 3.6 Mari kita perhatikan program C berikut:

```
main()
{
    int a,b,c,avg;
    scanf("%d %d %d",&a,&b,&c);
    avg=(a+b+c)/3;
    printf("avg= %d",avg);
}
```

Operator uniknya adalah: main, (), {}, int, scanf, &, “,", “;”, =, +, /, printf

Operan uniknya adalah: a,b,c,&a,&b,&c,a+b+c,avg,3,”%d %d %d”, “rata-rata=%d”

Sehingga,

η_1 = 12, η_2=11

Perkiraan Panjang = (12 * log 12 + 11 * log 11)
= (12 * 3.58 + 11 * 3.45) = (43 +38) = 81

Volume = Panjang * log(23) = 81 * 4.52 = 366

Kesimpulannya, teori Halstead mencoba memberikan definisi formal dan kuantifikasi atribut kualitatif seperti kompleksitas program, kemudahan pemahaman, dan tingkat abstraksi berdasarkan beberapa parameter tingkat rendah seperti jumlah operan, dan operator yang muncul dalam program. Ilmu perangkat lunak Halstead memberikan perkiraan kasar properti dari kumpulan besar perangkat lunak, tetapi meluas ke kasus individu agak tidak akurat.

3.10 ESTIMASI TINGKAT STAF

Setelah upaya yang diperlukan untuk menyelesaikan proyek perangkat lunak telah diperkirakan, kebutuhan staf untuk proyek tersebut dapat ditentukan. Putnam adalah orang pertama yang mempelajari masalah penentuan pola kepegawaian yang tepat untuk proyek perangkat lunak. Dia memperluas karya klasik Norden yang sebelumnya menyelidiki pola kepegawaian jenis penelitian dan pengembangan umum (R&D) jenis proyek. Untuk menghargai keunikan pola kepegawaian yang diinginkan untuk proyek perangkat lunak, pertama-tama kita harus memahami hasil Norden dan Putnam.

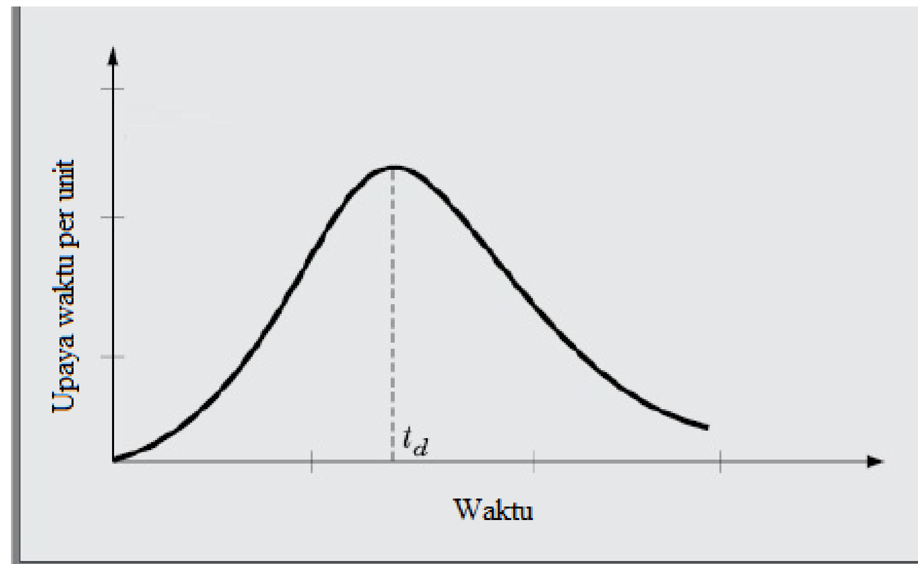
Karya Norden

Norden mempelajari pola kepegawaian dari beberapa proyek R&D. Dia menemukan bahwa pola kepegawaian jenis proyek R&D sangat berbeda dari manufaktur atau penjualan. Di outlet penjualan, jumlah staf penjualan biasanya tidak berubah seiring waktu. Misalnya, di supermarket, jumlah tenaga penjualan akan bergantung pada jumlah konter penjualan dan kira-kira konstan sepanjang waktu. Namun, pola staf jenis proyek R&D perlu berubah seiring waktu. Pada awal proyek R&D, kegiatan proyek direncanakan dan penyelidikan awal dilakukan. Selama ini, kebutuhan tenaga kerja rendah. Seiring berjalannya proyek, kebutuhan tenaga kerja meningkat, hingga mencapai puncaknya. Setelah itu, kebutuhan tenaga kerja secara bertahap berkurang. Norden menyimpulkan bahwa pola kepegawaian untuk setiap proyek R&D mulai dari tingkat rendah, meningkat hingga mencapai nilai puncak. Kemudian

mulai berkurang. Pola ini dapat didekati dengan kurva distribusi Rayleigh (lihat Gambar 3.6). Norden mewakili kurva Rayleigh dengan persamaan berikut:

$$E = \frac{K}{t_d^2} * t * e^{\frac{-t^2}{2t\,d^2}}$$

dimana, E adalah usaha yang diperlukan pada waktu t. E adalah indikasi jumlah developer (atau tingkat staf) pada waktu tertentu selama durasi proyek, K adalah area di bawah kurva, dan td adalah waktu di mana kurva mencapai nilai maksimumnya. Harus diingat bahwa hasil Norden dapat diterapkan pada proyek R&D umum dan tidak dimaksudkan untuk memodelkan pola kepegawaian proyek pengembangan perangkat lunak.



Gambar 3.6 Kurva Rayleigh.

Karya Putnam

Putnam mempelajari masalah staf proyek perangkat lunak dan menemukan bahwa pola staf untuk proyek pengembangan perangkat lunak memiliki karakteristik yang sangat mirip dengan proyek R&D lainnya. Hanya sejumlah kecil developer yang diperlukan di awal proyek untuk melaksanakan tugas perencanaan dan spesifikasi. Seiring kemajuan proyek dan pekerjaan yang lebih detail dilakukan, jumlah developer meningkat dan mencapai puncaknya selama pengujian produk. Setelah implementasi dan pengujian unit, jumlah staf proyek turun. Putnam menemukan bahwa kurva Rayleigh-Norden dapat disesuaikan untuk menghubungkan jumlah baris kode yang dikirimkan dengan upaya dan waktu yang diperlukan untuk mengembangkan produk. Dengan menganalisis sejumlah besar proyek pertahanan, Putnam memperoleh ekspresi berikut:

$$L = C_k K^{\frac{1}{3}} t_d^{\frac{4}{3}}$$

di mana istilah yang berbeda adalah sebagai berikut:

- K adalah total usaha yang dikeluarkan (dalam PM) dalam pengembangan produk dan L adalah ukuran produk di KLOC.
- t d sesuai dengan waktu sistem dan integrasi dan pengujian. Oleh karena itu, t d dapat dianggap sebagai waktu yang dibutuhkan untuk mengembangkan perangkat lunak.
- Ck adalah keadaan teknologi yang konstan dan mencerminkan kendala yang menghambat kemajuan programmer. Nilai khas C_k =2 untuk lingkungan pengembangan yang buruk (tidak ada metodologi, dokumentasi yang buruk,

dan tinjauan, dll.), C_k =8 untuk lingkungan pengembangan perangkat lunak yang baik (prinsip rekayasa perangkat lunak dipatuhi), C_k =11 untuk lingkungan yang sangat baik (selain mengikuti prinsip-prinsip rekayasa perangkat lunak, alat dan teknik otomatis digunakan). Nilai pasti C_k untuk proyek tertentu dapat dihitung dari data historis organisasi yang mengembangkannya. Putnam menyarankan bahwa peningkatan staf yang optimal pada sebuah proyek harus mengikuti kurva Rayleigh.

Untuk pemanfaatan sumber daya yang efisien serta penyelesaian proyek selama durasi yang optimal, mulai dari sejumlah kecil pengembang, harus ada penambahan staf dan setelah ukuran puncak tercapai, pengurangan staf diperlukan. Namun, peningkatan staf tidak boleh dilakukan dalam jumlah besar. Ukuran tim harus ditingkatkan atau diturunkan perlahan-lahan setiap kali diperlukan untuk mencocokkan kurva Rayleigh-Norden. Laporan pengalaman menunjukkan bahwa peningkatan yang sangat cepat dari staf proyek setiap saat selama pengembangan proyek berkorelasi dengan selip jadwal. Harus jelas bahwa tingkat tenaga kerja yang konstan sepanjang durasi proyek akan menyebabkan pemborosan usaha dan sebagai hasilnya akan meningkatkan waktu dan usaha yang diperlukan untuk mengembangkan produk. Jika jumlah developer yang konstan digunakan di semua fase proyek, beberapa fase akan kelebihan staf dan fase lainnya akan kekurangan staf yang menyebabkan penggunaan tenaga kerja yang tidak efisien, yang menyebabkan selip jadwal dan peningkatan biaya.

Jika kita memeriksa kurva Rayleigh, kita dapat melihat bahwa kira-kira 40 persen luas di bawah kurva Rayleigh berada di sebelah kiri *td* dan 60 persen luas di sebelah kanan *td*. Ini telah diverifikasi secara matematis dengan mengintegrasikan ekspresi yang diberikan oleh Putnam. Ini berarti bahwa upaya yang diperlukan untuk mengembangkan produk hingga upaya pemeliharaannya kira-kira dalam rasio 40:60.

3.11 PENJADWALAN

Penjadwalan tugas proyek merupakan kegiatan perencanaan proyek yang penting. Masalah penjadwalan, pada dasarnya, terdiri dari memutuskan tugas mana yang akan diambil kapan dan oleh siapa. Setelah jadwal telah dikerjakan dan proyek berjalan, manajer proyek memantau penyelesaian tugas yang tepat waktu dan mengambil tindakan korektif yang mungkin diperlukan setiap kali ada kemungkinan tergelincirnya jadwal. Untuk menjadwalkan aktivitas proyek, manajer proyek perangkat lunak perlu melakukan hal berikut:

1. Identifikasi semua kegiatan utama yang perlu dilakukan untuk menyelesaikan proyek.
2. Bagi setiap aktivitas menjadi tugas-tugas.
3. Tentukan ketergantungan di antara tugas-tugas yang berbeda.
4. Tetapkan perkiraan untuk jangka waktu yang diperlukan untuk menyelesaikan tugas.
5. Mewakili informasi dalam bentuk jaringan aktivitas.
6. Tentukan tanggal mulai dan berakhirnya tugas dari informasi yang direpresentasikan dalam jaringan aktivitas.
7. Tentukan jalur kritis. Jalur kritis adalah rantai tugas yang menentukan durasi proyek.
8. Alokasikan sumber daya untuk tugas.

Langkah pertama dalam menjadwalkan proyek perangkat lunak melibatkan mengidentifikasi semua aktivitas yang diperlukan untuk menyelesaikan proyek. Pengetahuan yang baik tentang seluk-beluk proyek dan proses pengembangan membantu para manajer untuk secara efektif mengidentifikasi kegiatan penting proyek. Selanjutnya, aktivitas dipecah

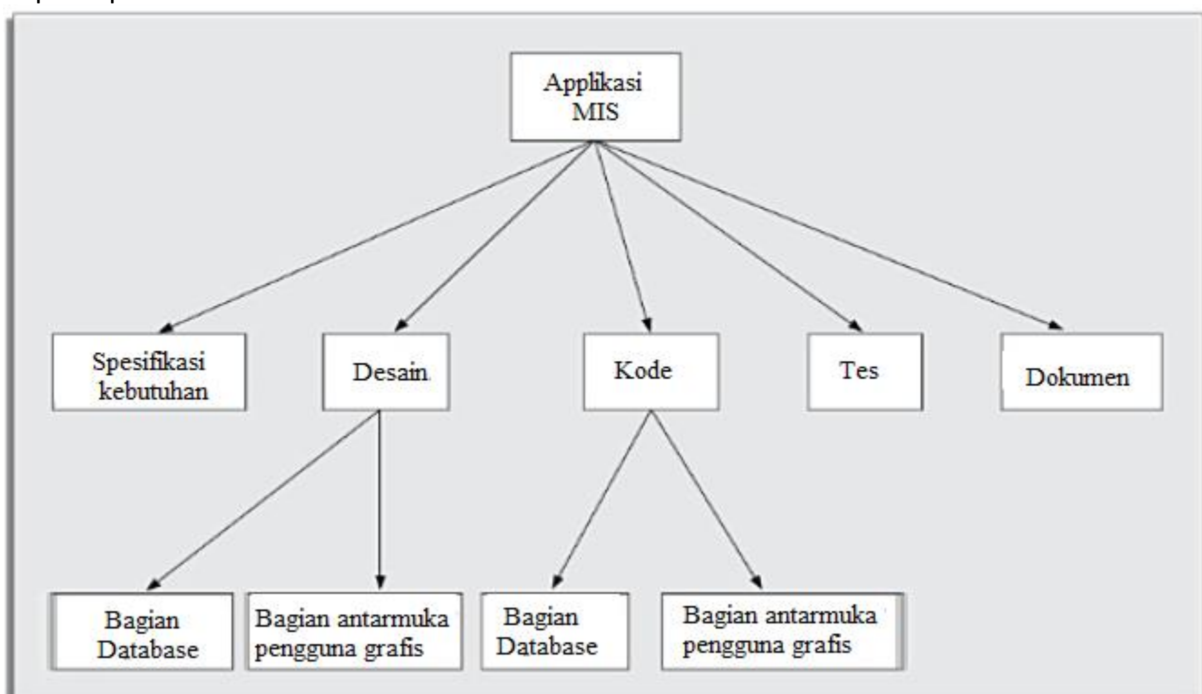
menjadi serangkaian aktivitas yang lebih kecil (subaktivitas). Subaktivitas terkecil disebut tugas yang ditugaskan ke developer yang berbeda. Unit terkecil dari aktivitas kerja yang tunduk pada manajemen perencanaan dan pengendalian disebut tugas.

Setelah manajer proyek membagi aktivitas menjadi tugas-tugas, ia harus menemukan ketergantungan di antara tugas-tugas tersebut. Ketergantungan di antara tugas-tugas yang berbeda menentukan urutan di mana tugas-tugas yang berbeda akan dilakukan. Jika tugas A memerlukan hasil tugas B yang lain, maka tugas A harus dijadwalkan setelah tugas B dan A dikatakan bergantung pada B. Secara umum, dependensi tugas menentukan urutan parsial antar tugas. Artinya, setiap tugas dapat mendahului subset dari tugas lain, tetapi beberapa tugas mungkin tidak memiliki urutan prioritas yang ditentukan di antara mereka (disebut tugas bersamaan).

Setelah representasi jaringan aktivitas telah dikerjakan, sumber daya dialokasikan untuk setiap aktivitas. Alokasi sumber daya biasanya dilakukan dengan menggunakan bagan Gantt. Setelah alokasi sumber daya dilakukan, representasi grafik evaluasi dan teknik tinjauan proyek (PERT) dikembangkan. Representasi grafik PERT berguna bagi manajer proyek untuk melakukan pemantauan dan pengendalian proyek. Sekarang mari kita bahas struktur perincian kerja, jaringan aktivitas, bagan Gantt dan PERT.

Struktur rincian kerja/*Work Breakdown Structure* (WBS)

Work Breakdown Structure (WBS) digunakan untuk secara rekursif menguraikan sekumpulan aktivitas tertentu menjadi aktivitas yang lebih kecil. Tugas adalah aktivitas kerja tingkat terendah dalam hierarki WBS. Mereka juga membentuk unit kerja dasar yang dialokasikan untuk developer dan dijadwalkan. Pertama, mari kita pahami mengapa perlu untuk memecah aktivitas proyek menjadi tugas. Setelah kegiatan proyek telah didekomposisi menjadi satu set tugas menggunakan WBS, kerangka waktu kapan setiap kegiatan akan dilakukan harus ditentukan. Akhir dari setiap aktivitas penting disebut milestone. Manajer proyek melacak kemajuan proyek dengan memantau penyelesaian tepat waktu dari tonggak sejarah. Jika dia mengamati bahwa beberapa tonggak mulai tertunda, dia dengan hati-hati memantau dan mengontrol kemajuan tugas, sehingga tenggat waktu keseluruhan masih dapat dipenuhi.



Gambar 3.7 Struktur rincian kerja dari masalah MIS.

WBS menyediakan notasi untuk merepresentasikan aktivitas, sub aktivitas, dan tugas yang diperlukan untuk menyelesaikan suatu masalah. Masing-masing diwakili menggunakan persegi panjang (lihat Gambar 3.7). Akar pohon diberi label dengan nama proyek. Setiap node dari pohon dipecah menjadi aktivitas-aktivitas yang lebih kecil yang dijadikan anak-anak dari node tersebut. Untuk menguraikan suatu aktivitas menjadi sub-aktivitas, pengetahuan yang baik tentang aktivitas dapat bermanfaat. Gambar 3.7 merepresentasikan WBS dari sistem informasi manajemen/*Management Information System* (MIS) perangkat lunak.

Berapa lama terurai?

Penguraian aktivitas dilakukan sampai salah satu dari berikut ini terpenuhi:

- Subaktivitas tingkat daun (tugas) membutuhkan sekitar dua minggu untuk berkembang.
- Kompleksitas tersembunyi terungkap, sehingga pekerjaan yang harus dilakukan dipahami dan dapat ditugaskan sebagai unit kerja ke salah satu pengembang.
- Peluang untuk menggunakan kembali komponen perangkat lunak yang ada diidentifikasi.

Memecah tugas ke tingkat yang terlalu kasar versus tingkat yang terlalu halus

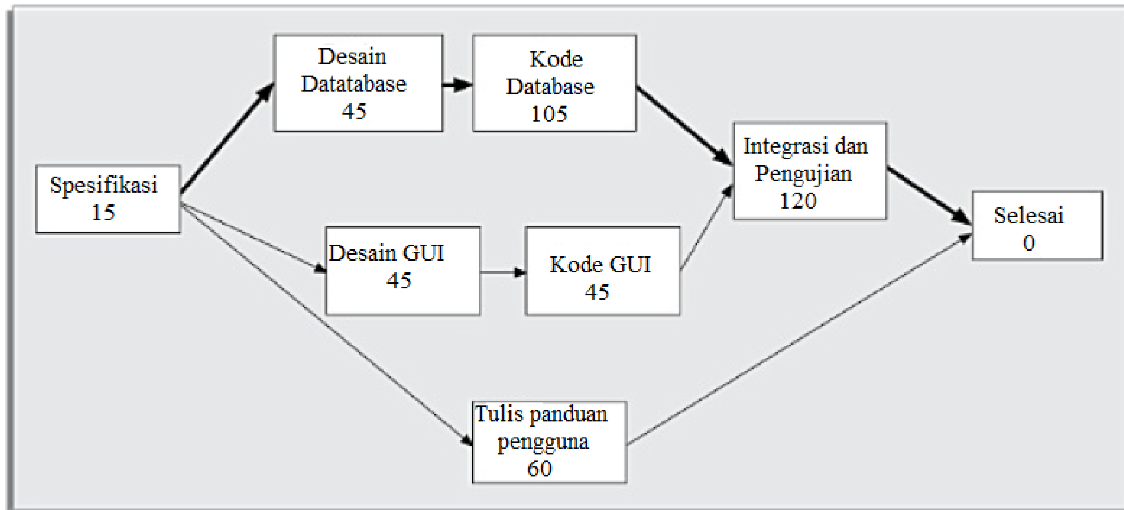
Mari kita selidiki terlebih dahulu implikasi dari melakukan dekomposisi ke tingkat yang sangat halus versus membiarkan dekomposisi pada butiran yang agak kasar. Penting untuk memecah aktivitas besar menjadi banyak tugas yang lebih kecil, karena tugas yang lebih kecil memungkinkan tonggak untuk ditempatkan pada jarak yang lebih pendek. Hal ini memungkinkan pemantauan yang ketat terhadap proyek dan tindakan korektif dapat dimulai segera setelah ada masalah yang diketahui. Namun, pembagian yang sangat halus berarti bahwa jumlah waktu yang tidak proporsional harus dihabiskan untuk mempersiapkan dan merevisi berbagai bagan.

Sekarang mari kita selidiki implikasi dari melakukan dekomposisi ke tingkat yang sangat kasar versus penguraian ke tingkat yang sangat halus. Ketika perincian tugas adalah beberapa bulan, pada saat masalah (penundaan jadwal) diketahui dan tindakan korektif dimulai, mungkin sudah terlambat bagi proyek untuk pulih. Di sisi lain, jika tugas didekomposisi menjadi granularity yang sangat kecil (masing-masing satu atau dua hari), maka jarak pencapaian menjadi terlalu dekat. Ini akan membutuhkan pemantauan status proyek yang sering dan memerlukan revisi yang sering pada dokumen rencana. Ini menjadi overhead yang tinggi pada manajer proyek dan efektivitasnya dalam pemantauan dan pengendalian proyek berkurang.

Saat memecah suatu aktivitas menjadi tugas-tugas yang lebih kecil, seorang manajer sering kali harus membuat beberapa keputusan sulit. Jika suatu aktivitas dipecah menjadi sejumlah besar sub-aktivitas yang sangat kecil, ini dapat didistribusikan ke lebih banyak pengembang. Jika pemesanan tugas memungkinkan solusi untuk ini dapat dilakukan secara mandiri (secara paralel), menjadi mungkin untuk mengembangkan produk lebih cepat (tentu saja dengan bantuan tenaga kerja tambahan!). Oleh karena itu, untuk dapat menyelesaikan sebuah proyek dalam waktu yang paling singkat, manajer perlu memecah tugas-tugas besar menjadi tugas-tugas yang lebih kecil, dengan harapan menemukan lebih banyak paralelisme. Namun, tidak berguna untuk membagi tugas ke dalam unit yang membutuhkan waktu kurang dari satu atau dua minggu untuk dieksekusi.

Contoh 3.8 Pertimbangkan sebuah proyek untuk pengembangan sistem informasi manajemen (SIM). Manajer proyek telah mengidentifikasi kegiatan pengembangan utama, spesifikasi persyaratan, desain, kode, pengujian, dan dokumen. menguraikan kegiatan menjadi tugas menggunakan teknik struktur rincian kerja.

Jawaban: Berdasarkan pengetahuan domain manajer, dia bisa sampai pada struktur rincian kerja yang ditunjukkan pada Gambar 3.8.



Gambar 3.8 Representasi jaringan aktivitas dari masalah MIS.

Activity on Edge (AoE): Dalam tugas representasi ini diasosiasikan dengan edge. Tepi juga dijelaskan dengan durasi tugas. Node dalam grafik mewakili tonggak proyek. Jaringan aktivitas awalnya direpresentasikan menggunakan representasi aktivitas di tepi (AoE). Namun, belakangan aktivitas pada simpul (AoN) menjadi populer karena representasi ini lebih mudah dipahami dan direvisi. Manajer dapat memperkirakan durasi waktu untuk tugas yang berbeda dalam beberapa cara. Satu kemungkinan adalah bahwa mereka dapat secara empiris menetapkan durasi untuk tugas yang berbeda. Namun ini mungkin bukan ide yang bagus, karena developer perangkat lunak sering kali tidak menyukai keputusan sepihak seperti itu. Namun, beberapa manajer lebih suka memperkirakan waktu untuk berbagai kegiatan sendiri. Mereka percaya bahwa jadwal yang agresif akan memotivasi para developer untuk melakukan pekerjaan yang lebih baik dan lebih cepat. Di sisi lain, eksperimen yang cermat telah menunjukkan bahwa jadwal agresif yang tidak realistis tidak hanya menyebabkan developer berkompromi pada aspek kualitas tidak berwujud, tetapi juga menyebabkan penundaan jadwal yang lebih besar dibandingkan dengan pendekatan lainnya. Alternatif yang mungkin adalah membiarkan setiap developer sendiri memperkirakan waktu untuk aktivitas yang akan ditugaskan kepadanya. Pendekatan ini dapat membantu untuk secara akurat memperkirakan durasi tugas tanpa menciptakan tekanan jadwal yang tidak semestinya.

Contoh 3.9: Tentukan representasi jaringan Aktivitas untuk proyek pengembangan MIS dari Contoh 3.7. Asumsikan bahwa manajer telah menentukan tugas yang akan diwakili dari struktur rincian kerja Gambar 3.7, dan telah menentukan durasi dan ketergantungan untuk setiap tugas seperti yang ditunjukkan pada Tabel 3.7.

Jawaban: Representasi jaringan aktivitas telah ditunjukkan pada Gambar 3.8.

Tabel 3.7 Parameter Proyek yang Dihitung dari Jaringan Aktivitas

Nomor Tugas	Tugas	Durasi	Tergantung tugas
T1	Spesifikasi	15	-
T2	Desain database	45	T1
T3	Desain GUI	30	T1
T4	Kode Database	105	T2
T5	Kode bagian GUI	45	T3

T6	Integrasi dan pengujian	120	T4 dan T5
T7	Tulis pedoman pengguna	60	T1

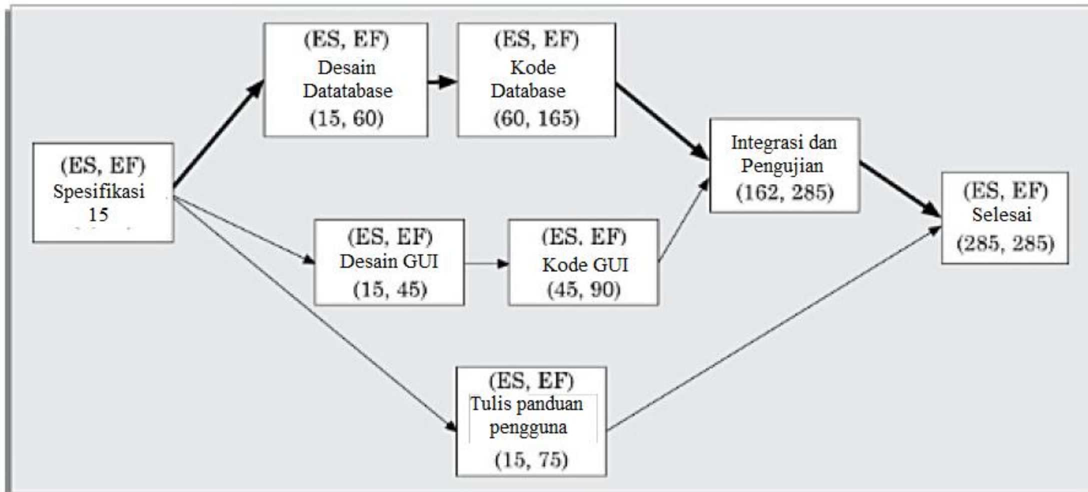
Metode Jalur Kritis/*Critical path Method*(CPM)

CPM dan PERT adalah teknik riset operasi yang dikembangkan pada akhir 1950-an. Sejak itu, mereka tetap sangat populer di kalangan manajer proyek. Akhir-akhir ini, kedua teknik ini telah digabungkan dan banyak alat manajemen proyek mendukungnya sebagai CPM/PERT. Jalur dalam grafik jaringan aktivitas adalah kumpulan simpul dan tepi yang berurutan dalam grafik ini dari simpul awal hingga simpul terakhir. Jalur kritis terdiri dari satu set tugas dependen yang perlu dilakukan secara berurutan dan yang bersama-sama membutuhkan waktu paling lama untuk diselesaikan. Tugas kritis adalah tugas dengan waktu kosong nol. Jalur dari node awal ke node akhir yang hanya berisi tugas-tugas kritis disebut jalur kritis. CPM adalah pendekatan algoritmik untuk menentukan jalur kritis dan waktu kendur untuk tugas yang tidak berada di jalur kritis melibatkan penghitungan jumlah berikut:

- **Minimum time (MT):** Ini adalah waktu minimum yang diperlukan untuk menyelesaikan proyek. Ini dihitung dengan menentukan maksimum semua jalur dari awal hingga akhir.
- **Earlyest start (ES):** Merupakan waktu maksimum dari suatu tugas dari semua jalur dari awal hingga tugas ini. ES untuk suatu tugas adalah ES dari tugas sebelumnya ditambah durasi tugas sebelumnya.
- **Lates Start Time (LST):** Ini adalah perbedaan antara MT dan maksimum semua jalur dari tugas ini hingga selesai. LST dapat dihitung dengan mengurangi durasi tugas berikutnya dari LST tugas berikutnya.
- **Earliest Finish Tme (EF):** EF untuk suatu tugas adalah jumlah waktu mulai paling awal dari tugas dan durasi tugas.
- **Latest Finish (LF):** LF menunjukkan waktu terakhir dimana tugas dapat diselesaikan tanpa mempengaruhi waktu penyelesaian akhir proyek. Sebuah tugas yang diselesaikan di luar LF-nya akan menyebabkan penundaan proyek. LF tugas dapat diperoleh dengan mengurangi maksimum semua jalur dari tugas ini hingga selesai dari MT.
- **Slack time (ST):** Waktu slack (atau waktu mengambang) adalah total waktu bahwa tugas mungkin tertunda sebelum akan mempengaruhi waktu akhir proyek. Waktu slack menunjukkan "fleksibilitas" dalam memulai dan menyelesaikan tugas. ST untuk suatu tugas adalah LS-ES dan secara ekuivalen dapat ditulis sebagai LF-EF.

Contoh 3.10 Gunakan jaringan Aktivitas Gambar 3.8 untuk menentukan ES dan EF untuk setiap tugas untuk masalah MIS Contoh 3.7.

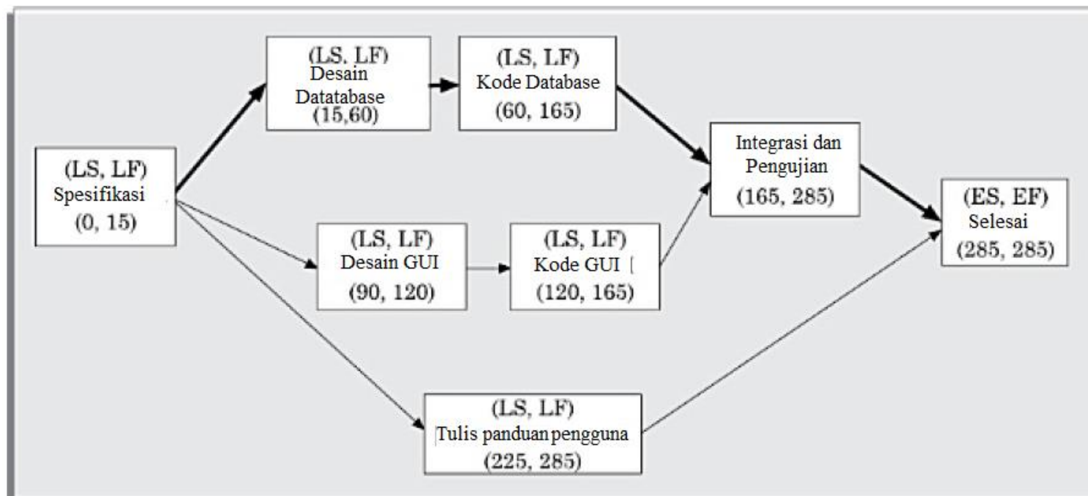
Jawaban: Jaringan aktivitas dengan nilai ES dan EF yang dihitung telah ditunjukkan pada Gambar 3.9.



Gambar 3.9 AoN untuk masalah MIS dengan (ES,EF).

Contoh 3.11 Gunakan jaringan Aktivitas Gambar 3.9 untuk menentukan LS dan LF untuk setiap tugas untuk masalah MIS Contoh 3.7.

Jawaban: Jaringan aktivitas dengan nilai LS dan LF yang dihitung telah ditunjukkan pada Gambar 3.10.



Gambar 3.10 AoN masalah MIS dengan (LS,LF).

CPM dapat digunakan untuk menentukan perkiraan durasi minimum suatu proyek dan waktu slack yang terkait dengan berbagai tugas non-kritis. Jadi setiap jalur yang durasinya sama dengan MT adalah jalur kritis. Perhatikan bahwa, mungkin ada lebih dari satu jalur kritis untuk sebuah proyek. Tugas yang termasuk dalam jalur kritis harus mendapat perhatian khusus baik oleh manajer proyek maupun personel yang ditugaskan untuk melakukan tugas tersebut. Salah satu cara adalah menggambar jalur kritis dengan garis ganda, bukan garis tunggal. Jalur kritis dapat berubah seiring berjalannya proyek. Ini mungkin terjadi ketika tugas diselesaikan baik di belakang atau lebih cepat dari jadwal. Parameter proyek untuk tugas yang berbeda untuk masalah MIS dapat dihitung sebagai berikut:

1. Hitung ES dan EF untuk setiap tugas. Gunakan aturan: ES sama dengan EF terbesar dari pendahulunya
2. Hitung LS dan LF untuk setiap tugas. Gunakan aturan: LF sama dengan LS terkecil dari penerus langsung
3. Hitung ST untuk setiap tugas. Gunakan aturan: ST=LF-EF

Pada Gambar 3.9 dan Gambar 3.10 saya menunjukkan perhitungan (ES,EF) dan (LS,LF) masing-masing. Dari parameter proyek ini untuk tugas yang berbeda untuk masalah MIS telah direpresentasikan dalam Tabel 3.8.

Tabel 3.8 Parameter Proyek yang Dihitung Dari Jaringan Aktivitas

Tugas	**ES**	**EF**	**LS**	**LF**	**ST**
Spesifikasi	0	15	0	15	0
Desain database	15	60	15	60	0
Desain GUI	15	45	90	120	75
Kode Database	60	165	60	165	0
Kode bagian GUI	45	90	120	165	75
Integrasi dan pengujian	165	285	165	285	0
Tulis pedoman pengguna	15	75	225	285	210

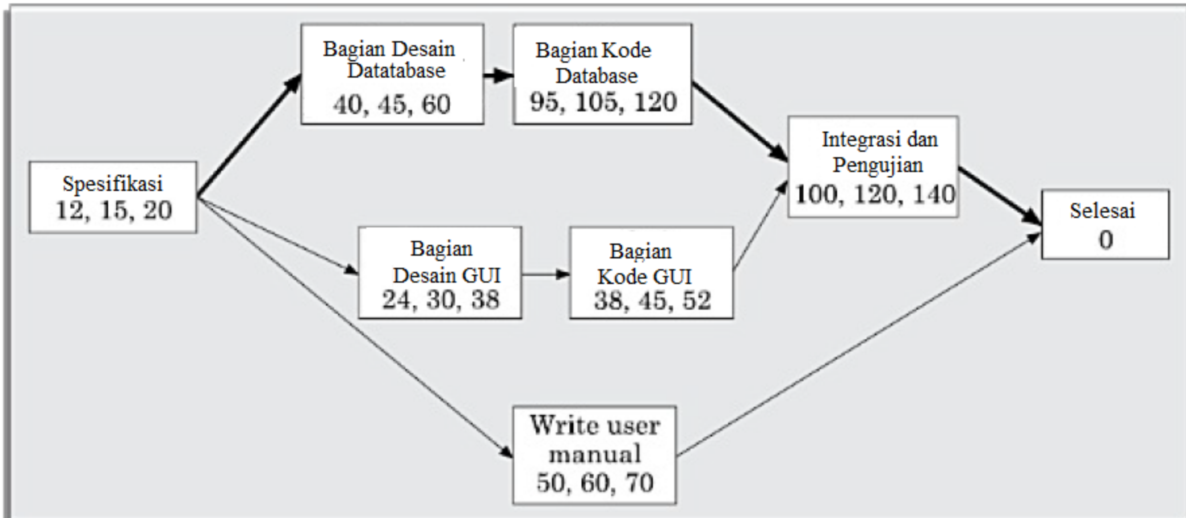
Jalur kritis adalah semua jalur yang durasinya sama dengan MT. Jalur kritis pada Gambar 3.8 ditunjukkan menggunakan panah tebal.

Grafik PERT

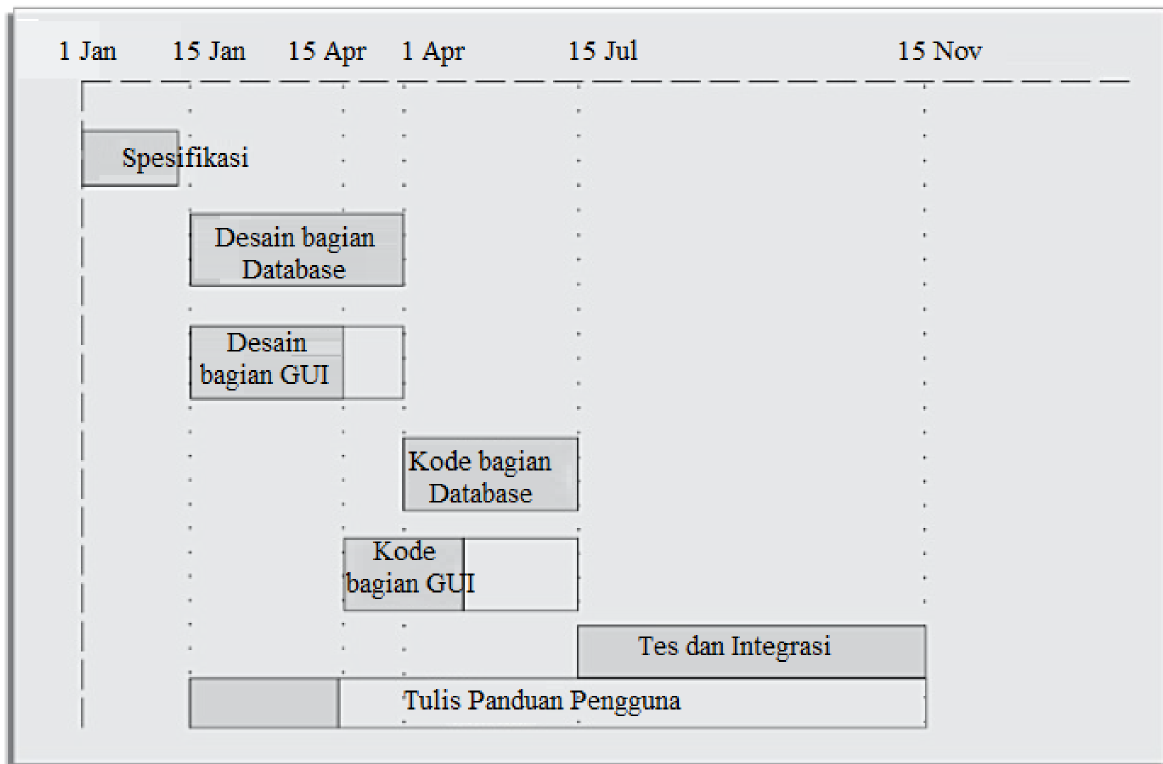
Durasi aktivitas yang dihitung menggunakan jaringan aktivitas hanya perkiraan durasi. Oleh karena itu, tidak mungkin untuk memperkirakan perkiraan kasus terburuk (pesimis) dan kasus terbaik (optimis) menggunakan diagram aktivitas. Karena, durasi sebenarnya mungkin berbeda dari perkiraan durasi, kegunaan diagram jaringan aktivitas terbatas. CPM dapat digunakan untuk menentukan durasi proyek, tetapi tidak memberikan indikasi apa pun tentang kemungkinan memenuhi jadwal tersebut.

Grafik evaluasi dan teknik peninjauan proyek/*Project evaluation and review technique* (PERT) adalah bentuk grafik aktivitas yang lebih canggih. Manajer proyek tahu bahwa ada ketidakpastian yang cukup besar tentang berapa banyak waktu yang dibutuhkan untuk menyelesaikan suatu tugas. Durasi yang ditetapkan untuk tugas oleh manajer proyek bagaimanapun juga hanyalah perkiraan. Oleh karena itu, pada kenyataannya durasi suatu kegiatan adalah variabel acak dengan beberapa distribusi probabilitas. Dalam konteks ini, grafik PERT dapat digunakan untuk menentukan waktu probabilistik untuk mencapai berbagai batu mil proyek, termasuk batu mil terakhir. Bagan PERT seperti jaringan aktivitas terdiri dari jaringan kotak dan panah. Kotak mewakili aktivitas dan panah mewakili dependensi tugas. Bagan PERT mewakili variasi statistik dalam perkiraan proyek dengan asumsi ini adalah distribusi normal. PERT memungkinkan untuk beberapa keacakan dalam waktu penyelesaian tugas, dan oleh karena itu memberikan kemampuan untuk menentukan probabilitas untuk mencapai tonggak proyek berdasarkan kemungkinan menyelesaikan setiap tugas di sepanjang jalan ke tonggak tersebut. Setiap tugas dijelaskan dengan tiga perkiraan:

- Optimis (O): Waktu penyelesaian tugas kasus terbaik.
- Perkiraan kemungkinan besar (M): Kemungkinan besar waktu penyelesaian tugas.
- Kasus terburuk (W): Kemungkinan waktu penyelesaian tugas kasus terburuk.



Gambar 3.11 Representasi grafik PERT dari masalah MIS.



Gambar 3.12 Representasi Gantt chart dari masalah MIS.

Gantt Chart

Gantt chart dinamai menurut nama pengembangnya Henry Gantt. Gantt chart adalah salah satu bentuk diagram batang. Sumbu vertikal mencantumkan semua tugas yang harus dilakukan. Batang digambar sepanjang sumbu y, satu untuk setiap tugas. Bagan Gantt yang digunakan dalam manajemen proyek perangkat lunak sebenarnya adalah versi yang disempurnakan dari bagan Gantt standar. Dalam bagan Gantt yang digunakan untuk manajemen proyek perangkat lunak, setiap batang terdiri dari bagian yang tidak diarsir dan bagian yang diarsir. Bagian bilah yang diarsir menunjukkan lamanya waktu yang diperkirakan untuk setiap tugas. Bagian yang tidak diarsir menunjukkan slack time atau lax time. Kelonggaran waktu menunjukkan kelonggaran atau keluwesan yang tersedia dalam memenuhi waktu paling akhir di mana suatu tugas harus diselesaikan. Representasi Gantt

chart untuk masalah MIS pada Gambar 3.8 ditunjukkan pada Gambar 3.12. Bagan Gantt berguna untuk perencanaan sumber daya (yaitu mengalokasikan sumber daya untuk kegiatan). Berbagai jenis sumber daya yang perlu dialokasikan untuk kegiatan termasuk staf, perangkat keras, dan perangkat lunak.

Bagan Gantt adalah jenis bagan batang khusus di mana setiap batang mewakili suatu aktivitas. Bar digambar sepanjang garis waktu. Panjang setiap batang sebanding dengan durasi waktu yang direncanakan untuk aktivitas yang bersangkutan. Representasi bagan Gantt dari jadwal proyek sangat membantu dalam merencanakan pemanfaatan sumber daya, sedangkan bagan PERT berguna untuk memantau kemajuan kegiatan secara tepat waktu. Juga, lebih mudah untuk mengidentifikasi aktivitas paralel dalam sebuah proyek menggunakan grafik PERT. Manajer proyek perlu mengidentifikasi aktivitas paralel dalam proyek untuk penugasan ke developer yang berbeda.

Pemantauan dan pengendalian proyek

Setelah proyek berjalan, manajer proyek memantau proyek secara terus menerus untuk memastikan bahwa proyek berjalan sesuai rencana. Manajer proyek menunjuk peristiwa penting tertentu seperti penyelesaian beberapa aktivitas penting sebagai tonggak sejarah. Beberapa contoh tonggak adalah sebagai berikut—sebuah tonggak dapat berupa persiapan dan peninjauan dokumen SRS, penyelesaian pengkodean dan pengujian unit, dll. Setelah tonggak tercapai, manajer proyek dapat mengasumsikan bahwa beberapa kemajuan terukur telah dibuat. Jika keterlambatan dalam mencapai tonggak diprediksi, maka tindakan korektif mungkin harus diambil. Ini mungkin memerlukan pengerjaan ulang semua jadwal dan menghasilkan jadwal baru.

Seperti yang telah disebutkan, grafik PERT sangat berguna dalam pemantauan dan pengendalian proyek. Lintasan dalam graf ini adalah himpunan simpul dan tepi yang berurutan dari simpul awal hingga simpul terakhir. Jalur kritis dalam grafik ini adalah jalur di mana setiap tonggak penting untuk memenuhi tenggat waktu proyek. Dengan kata lain, jika ada penundaan yang terjadi di sepanjang jalur kritis, seluruh proyek akan tertunda. Oleh karena itu, penting untuk mengidentifikasi semua jalur kritis dalam jadwal—mengikuti jadwal tugas yang muncul di jalur kritis adalah sangat penting untuk memenuhi tanggal pengiriman. Harap dicatat bahwa mungkin ada lebih dari satu jalur kritis dalam jadwal. Tugas di sepanjang jalur kritis disebut tugas kritis. Tugas-tugas kritis perlu dipantau secara ketat dan tindakan korektif perlu dimulai segera setelah keterlambatan diketahui. Jika perlu, seorang manajer dapat mengalihkan sumber daya dari tugas non-kritis ke tugas kritis sehingga semua tonggak sepanjang jalur kritis terpenuhi.

Beberapa alat tersedia yang dapat membantu Anda untuk mengetahui jalur kritis dalam jadwal yang tidak dibatasi, tetapi mencari tahu jadwal yang optimal dengan keterbatasan sumber daya dan dengan sejumlah besar tugas paralel adalah masalah yang sangat sulit. Ada beberapa produk komersial untuk mengotomatisasi teknik penjadwalan yang tersedia. Alat populer untuk membantu menggambar grafik terkait jadwal termasuk perangkat lunak MS-Project yang tersedia di komputer pribadi.

3.12 ORGANISASI DAN STRUKTUR TIM

Biasanya setiap organisasi pengembangan perangkat lunak menangani beberapa proyek setiap saat. Organisasi perangkat lunak menugaskan tim developer yang berbeda untuk menangani proyek perangkat lunak yang berbeda. Berkenaan dengan organisasi staf, ada dua isu penting—Bagaimana struktur organisasi secara keseluruhan? Dan, bagaimana tim proyek individu terstruktur? Ada beberapa cara standar di mana organisasi perangkat lunak dan tim dapat terstruktur.

Struktur organisasi

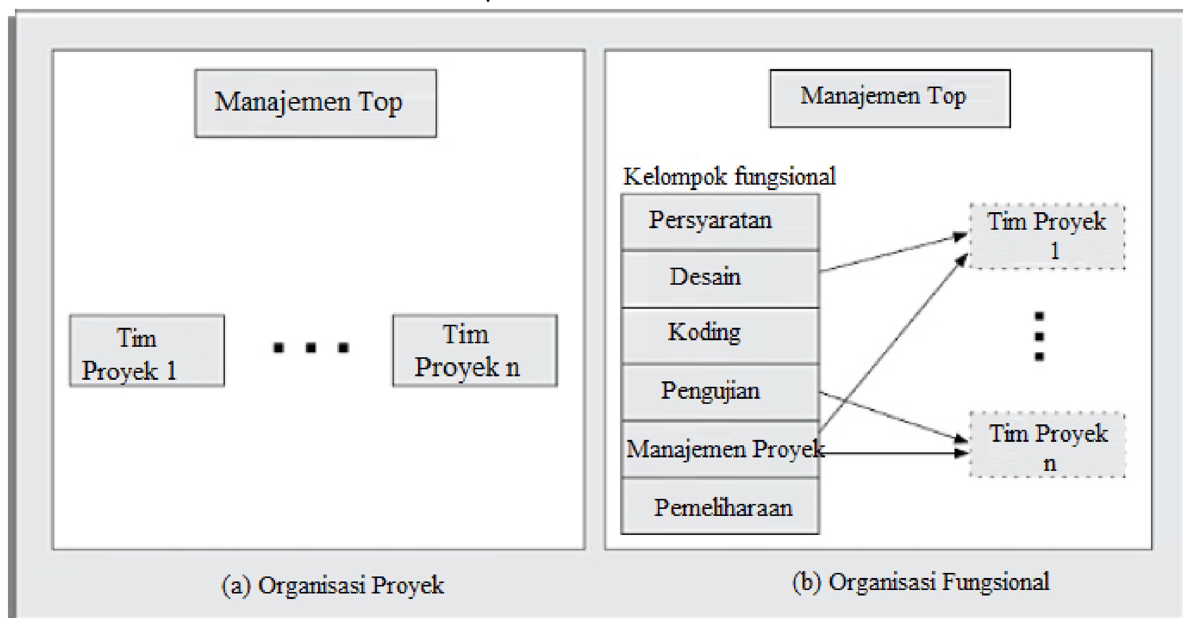
Pada dasarnya ada tiga cara luas di mana organisasi pengembangan perangkat lunak dapat terstruktur — format fungsional, format proyek, dan format matriks.

Format fungsional

Dalam format fungsional, staf pengembangan dibagi berdasarkan kelompok fungsional tertentu di mana mereka berasal. Format ini secara skematis telah ditunjukkan pada Gambar 3.13(a). Proyek-proyek yang berbeda meminjam developer dari berbagai kelompok fungsional untuk fase tertentu dari proyek dan mengembalikan mereka ke kelompok fungsional setelah selesainya fase. Akibatnya, tim programmer yang berbeda dari kelompok fungsional yang berbeda melakukan fase proyek yang berbeda. Misalnya, satu tim mungkin melakukan spesifikasi persyaratan, yang lain melakukan desain, dan seterusnya. Produk yang selesai sebagian berpindah dari satu tim ke tim lain saat produk berkembang. Oleh karena itu, format fungsional memerlukan komunikasi yang cukup di antara tim yang berbeda dan pengembangan dokumen berkualitas baik karena pekerjaan satu tim harus dipahami dengan jelas oleh tim berikutnya yang mengerjakan proyek. Oleh karena itu, organisasi fungsional mengamanatkan dokumentasi berkualitas baik untuk dihasilkan setelah setiap kegiatan.

Format proyek

Dalam format proyek, staf pengembangan dibagi berdasarkan proyek tempat mereka bekerja (Lihat Gambar 3.13(b)). Satu set developer ditugaskan untuk setiap proyek pada awal proyek, dan tetap bersama proyek sampai penyelesaian proyek. Dengan demikian, tim yang sama melakukan semua aktivitas siklus hidup. Keuntungan dari format proyek adalah menyediakan rotasi pekerjaan. Artinya, setiap developer melakukan aktivitas siklus hidup yang berbeda dalam sebuah proyek. Namun, ini menghasilkan pemanfaatan tenaga kerja yang buruk, karena tim proyek penuh dibentuk sejak awal proyek, dan sangat sedikit pekerjaan untuk tim selama fase awal siklus hidup.



Gambar 3.13 Representasi skematis dari organisasi fungsional dan proyek.

Format fungsional versus proyek

Meskipun komunikasi yang lebih besar di antara anggota tim mungkin tampak sebagai overhead yang dapat dihindari, format fungsional memiliki banyak keuntungan. Keuntungan utama dari organisasi fungsional adalah:

- Kemudahan penempatan staf

- Produksi dokumen berkualitas baik
- Spesialisasi pekerjaan
- Penanganan masalah yang terkait dengan pergantian tenaga kerja3 secara efisien.

Organisasi fungsional memungkinkan developer untuk menjadi spesialis dalam peran tertentu, mis. analisis persyaratan, desain, pengkodean, pengujian, pemeliharaan, dll. Mereka melakukan peran ini berulang kali untuk proyek yang berbeda dan mengembangkan wawasan mendalam untuk pekerjaan mereka. Ini juga menghasilkan lebih banyak perhatian yang diberikan pada dokumentasi yang tepat di akhir fase karena kebutuhan yang lebih besar untuk komunikasi yang jelas antara tim yang melakukan fase yang berbeda. Organisasi fungsional juga memberikan solusi yang efisien untuk masalah kepegawaian. Masalah staf proyek berkurang secara signifikan karena personel dapat dibawa ke proyek sesuai kebutuhan, dan dikembalikan ke kelompok fungsional ketika mereka tidak lagi dibutuhkan. Ini mungkin adalah keuntungan paling penting dari organisasi fungsional.

Struktur organisasi proyek memaksa manajer untuk menerima jumlah developer yang hampir konstan selama seluruh durasi proyeknya. Hal ini mengakibatkan developer tidak bekerja pada fase awal pengembangan perangkat lunak dan berada di bawah tekanan luar biasa pada fase pengembangan selanjutnya. Keuntungan lebih lanjut dari organisasi fungsional adalah lebih efektif dalam menangani masalah pergantian tenaga kerja. Ini karena developer dapat didatangkan dari kumpulan fungsional saat dibutuhkan. Selain itu, organisasi ini mengamanatkan produksi dokumen berkualitas baik, sehingga developer baru dapat dengan cepat terbiasa dengan pekerjaan yang sudah dilakukan.

Terlepas dari beberapa keuntungan penting dari organisasi fungsional, itu tidak terlalu populer di industri perangkat lunak. Paradoks yang tampak ini tidak sulit untuk dijelaskan. Kita dapat dengan mudah mengidentifikasi tiga poin berikut:

- Format proyek memberikan rotasi pekerjaan kepada anggota tim. Artinya, setiap anggota tim mengambil peran sebagai desainer, koder, penguji, dll selama proyek berlangsung. Di sisi lain, mengingat kekurangan keterampilan saat ini, akan sangat sulit bagi organisasi fungsional untuk mengisi slot untuk beberapa peran seperti kelompok pemeliharaan, pengujian, dan pengkodean.
- Masalah lain dengan organisasi fungsional adalah jika sebuah organisasi menangani proyek yang membutuhkan pengetahuan tentang area domain khusus, maka pakar domain ini tidak dapat dibawa masuk dan keluar dari proyek untuk fase yang berbeda, kecuali jika perusahaan menangani sejumlah besar proyek semacam itu.
- Untuk alasan yang jelas format fungsional tidak cocok untuk organisasi kecil yang hanya menangani satu atau dua proyek.

Format matriks

Sebuah organisasi matriks dimaksudkan untuk memberikan keuntungan dari kedua struktur fungsional dan proyek. Dalam organisasi matriks, kumpulan spesialis fungsional ditugaskan untuk proyek yang berbeda sesuai kebutuhan. Dengan demikian, penyebaran spesialis fungsional yang berbeda dalam proyek yang berbeda dapat direpresentasikan dalam matriks (lihat Gambar 3.14) Pada Gambar 3.14 mengamati bahwa anggota yang berbeda dari spesialisasi fungsional ditugaskan untuk proyek yang berbeda. Oleh karena itu dalam organisasi matriks, manajer proyek perlu berbagi tanggung jawab untuk proyek dengan sejumlah manajer fungsional individu.

	Project			
Functional group	#1	#2	#3	
#1	2	0	3	Functional manager 1
#2	0	5	3	Functional manager 2
#3	0	4	2	Functional manager 3
#4	1	4	0	Functional manager 4
#5	0	4	6	Functional manager 5
	Project manager 1	Project manager 2	Project manager 3	

Gambar 3.14 Organisasi matriks.

Organisasi matriks dapat dicirikan sebagai lemah atau kuat, tergantung pada otoritas relatif dari manajer fungsional dan manajer proyek. Dalam matriks fungsional yang kuat, manajer fungsional memiliki wewenang untuk menugaskan pekerja ke proyek dan manajer proyek harus menerima personel yang ditugaskan. Dalam matriks yang lemah, manajer proyek mengontrol anggaran proyek, dapat menolak pekerja dari kelompok fungsional, atau bahkan memutuskan untuk mempekerjakan pekerja luar. Dua masalah penting yang sering dialami oleh organisasi matriks adalah:

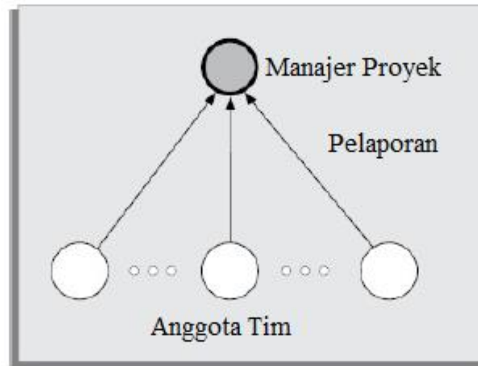
- Konflik antara manajer fungsional dan manajer proyek atas alokasi pekerja.
- Pergeseran pekerja yang sering dalam mode pemadam kebakaran karena krisis terjadi di berbagai proyek.

Struktur Tim

Struktur tim membahas organisasi tim proyek individu. Mari kita periksa kemungkinan cara di mana tim proyek individu diatur. Dalam teks ini, kita hanya akan mempertimbangkan tiga struktur tim formal-demokratis, kepala programmer, dan organisasi tim kontrol campuran, meskipun beberapa variasi lain dari struktur ini dimungkinkan. Proyek dengan kompleksitas dan ukuran tertentu seringkali membutuhkan struktur tim khusus untuk kerja yang efisien.

Ketua tim programmer

Dalam organisasi tim ini, seorang insinyur senior memberikan kepemimpinan teknis dan ditunjuk sebagai kepala pemrogram. Kepala programmer membagi tugas menjadi banyak tugas yang lebih kecil dan menugaskannya kepada anggota tim. Dia juga memverifikasi dan mengintegrasikan produk yang dikembangkan oleh anggota tim yang berbeda. Struktur tim kepala programmer ditunjukkan pada Gambar 3.15. Programmer utama memberikan otoritas, dan struktur ini bisa dibilang lebih efisien daripada tim demokratis untuk masalah yang dipahami dengan baik. Namun, tim kepala programmer mengarah ke moral tim yang lebih rendah, karena anggota tim bekerja di bawah pengawasan konstan kepala pemrogram. Ini juga menghambat pemikiran asli mereka. Tim programmer kepala tunduk pada kegagalan titik tunggal karena terlalu banyak tanggung jawab dan wewenang diberikan kepada programmer utama. Artinya, sebuah proyek mungkin sangat menderita, jika kepala programmer meninggalkan organisasi atau menjadi tidak tersedia karena beberapa alasan lain.

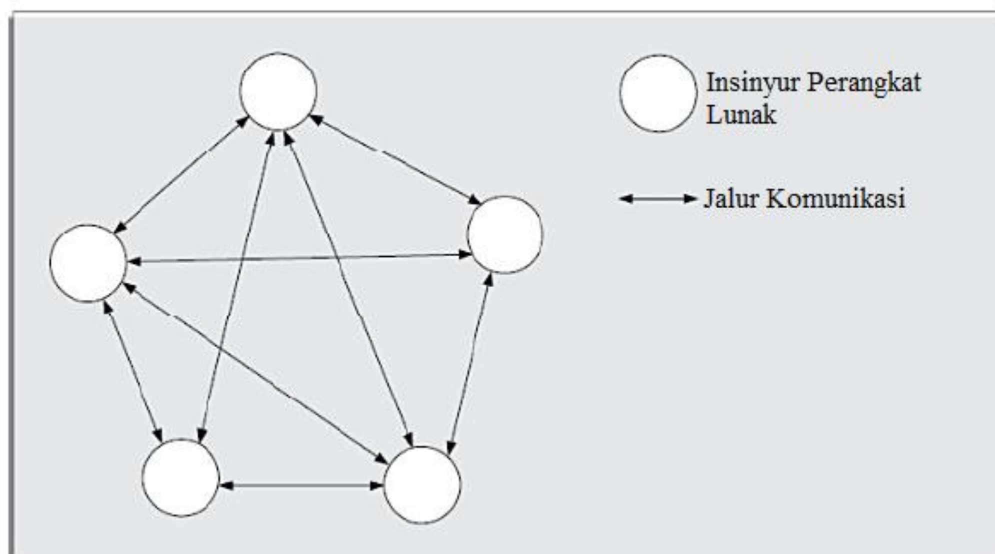


Gambar 3.15 Struktur tim kepala programmer.

Tim programmer kepala mungkin merupakan cara paling efisien untuk menyelesaikan proyek sederhana dan kecil karena programmer kepala dapat dengan cepat membuat desain yang memuaskan dan meminta programmer untuk mengkodekan modul yang berbeda dari solusi desainnya. Sekarang mari kita coba memahami jenis proyek yang cocok untuk organisasi tim programmer kepala. Misalkan sebuah organisasi telah berhasil menyelesaikan banyak proyek SIM sederhana. Kemudian, untuk proyek MIS serupa, struktur tim kepala programmer dapat diadopsi. Struktur tim programmer kepala bekerja dengan baik ketika tugas berada dalam jangkauan intelektual satu individu. Namun, bahkan untuk masalah yang sederhana dan mudah dipahami, organisasi harus selektif dalam mengadopsi struktur kepala pemrogram. Struktur tim programmer kepala tidak boleh digunakan kecuali pentingnya penyelesaian awal melebihi faktor-faktor lain seperti moral tim, pengembangan pribadi, dll.

Tim Demokrat

Struktur tim demokratis, seperti namanya, tidak memaksakan hierarki tim formal (lihat Gambar 3.16). Biasanya, seorang manajer memberikan kepemimpinan administratif. Pada waktu yang berbeda, anggota kelompok yang berbeda memberikan kepemimpinan teknis.

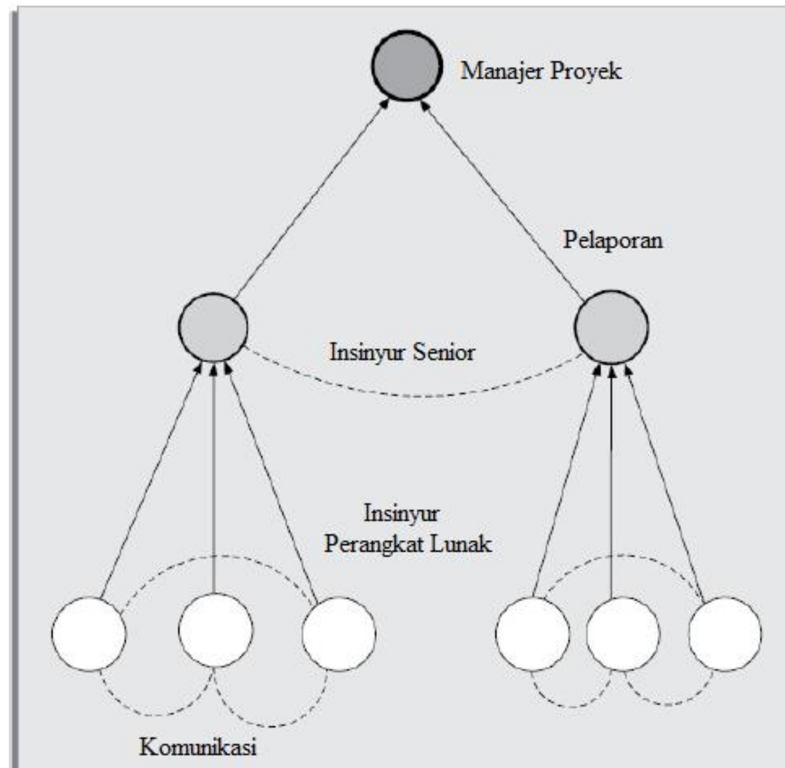


Gambar 3.16 Struktur tim Demokrat.

Dalam organisasi yang demokratis, anggota tim memiliki semangat kerja dan kepuasan kerja yang lebih tinggi. Akibatnya, ia menderita lebih sedikit pergantian tenaga kerja. Meskipun tim demokratis kurang produktif dibandingkan dengan tim programmer utama, struktur tim demokratis sesuai untuk masalah yang kurang dipahami, karena sekelompok developer dapat menemukan solusi yang lebih baik daripada satu individu seperti dalam tim

programmer utama. Struktur tim yang demokratis cocok untuk proyek berorientasi penelitian yang membutuhkan kurang dari lima atau enam pengembang. Untuk proyek berukuran besar, organisasi demokrasi murni cenderung menjadi kacau. Organisasi tim demokratis mendorong pemrograman tanpa ego karena programmer dapat berbagi dan meninjau pekerjaan satu sama lain. Untuk menghargai konsep pemrograman tanpa ego, kita perlu memahami konsep ego dari perspektif psikologis.

Sebagian besar dari Anda mungkin pernah mendengar tentang seniman temperamental yang sangat bangga dengan apa pun yang mereka buat. Biasanya, psikologi manusia membuat seorang individu bangga dengan segala sesuatu yang ia ciptakan dengan menggunakan pemikiran orisinal. Pengembangan perangkat lunak membutuhkan pemikiran orisinal juga, meskipun dari jenis yang berbeda. Psikologi manusia membuat seseorang terlibat secara emosional dengan ciptaannya dan menghalanginya dari pemeriksaan objektif atas ciptaannya. Sama seperti seniman temperamental, programmer merasa sangat sulit untuk menemukan bug dalam program mereka sendiri atau kekurangan dalam desain mereka sendiri. Oleh karena itu, cara terbaik untuk menemukan masalah dalam desain atau kode adalah meminta seseorang untuk meninjaunya.



Gambar 3.17 Struktur tim campuran.

Seringkali, harus menjelaskan program seseorang kepada orang lain memberi seseorang cukup objektivitas untuk mencari tahu apa yang mungkin salah. Pengamatan ini adalah ide dasar di balik penelusuran kode yang akan dibahas dalam Bab 10. Penerapannya, adalah untuk mendorong tim demokratis untuk berpikir bahwa desain, kode, dan hasil lainnya adalah milik seluruh kelompok. Ini disebut pemrograman tanpa ego karena mencoba untuk menghindari programmer menginvestasikan banyak ego dalam aktivitas pengembangan yang mereka lakukan dalam pengaturan yang demokratis. Namun, struktur tim yang demokratis memiliki satu kelemahan—anggota tim mungkin membuang banyak waktu untuk berdebat tentang hal-hal sepele karena kurangnya otoritas dalam tim untuk menyelesaikan perdebatan.

Organisasi tim kontrol campuran

Organisasi tim kontrol campuran, seperti namanya, mengacu pada ide-ide dari organisasi demokratis dan organisasi kepala-programmer. Organisasi tim kontrol campuran ditunjukkan pada Gambar 3.17. Organisasi tim ini menggabungkan pelaporan hierarkis dan pengaturan demokratis. Pada Gambar 3.17, jalur komunikasi ditampilkan sebagai garis putus-putus dan struktur pelaporan ditampilkan menggunakan panah padat. Organisasi tim kontrol campuran cocok untuk ukuran tim yang besar. Penataan demokratis di tingkat developer senior digunakan untuk menguraikan masalah menjadi bagian-bagian kecil. Setiap pengaturan demokratis di tingkat programmer mencoba solusi untuk satu bagian. Dengan demikian, organisasi tim ini sangat cocok untuk menangani program yang besar dan kompleks. Struktur tim ini sangat populer dan digunakan di banyak perusahaan pengembangan perangkat lunak.

3.13 STAF

Manajer proyek perangkat lunak biasanya bertanggung jawab untuk memilih tim mereka. Oleh karena itu, mereka perlu mengidentifikasi developer perangkat lunak yang baik untuk keberhasilan proyek. Kesalahpahaman umum yang dipegang oleh manajer sebagaimana dibuktikan dalam praktik kepegawaian, perencanaan dan penjadwalan mereka, adalah asumsi bahwa satu insinyur perangkat lunak sama produktifnya dengan yang lain. Namun, percobaan telah mengungkapkan bahwa terdapat variabilitas besar produktivitas antara developer perangkat lunak terburuk dan terbaik dalam skala 1 sampai 30. Bahkan, developer terburuk kadang-kadang bahkan mengurangi produktivitas keseluruhan tim, dan dengan demikian berlaku menunjukkan produktivitas negatif. Oleh karena itu, memilih developer perangkat lunak yang baik sangat penting untuk keberhasilan suatu proyek.

Siapa insinyur perangkat lunak yang baik?

Di masa lalu, beberapa penelitian tentang ciri-ciri seorang insinyur perangkat lunak yang baik telah dilakukan. Semua studi ini secara kasar menyetujui atribut berikut yang harus dimiliki oleh developer perangkat lunak yang baik:

- Paparan teknik sistematis, yaitu keakraban dengan prinsip-prinsip rekayasa perangkat lunak.
- Pengetahuan teknis yang baik tentang area proyek (Pengetahuan domain)
- Kemampuan pemrograman yang baik.
- Kemampuan komunikasi yang baik. Keterampilan ini terdiri dari keterampilan lisan, tertulis, dan interpersonal.
- Motivasi tinggi.
- Pengetahuan yang baik tentang dasar-dasar ilmu komputer
- Intelijen.
- Kemampuan untuk bekerja dalam tim.
- Disiplin, dll.

Studi menunjukkan bahwa atribut ini bervariasi sebanyak 1:30 untuk kandidat miskin dan cerdas. Eksperimen yang dilakukan oleh Sackman [1968] menunjukkan bahwa rasio jam pengkodean untuk programmer terburuk hingga terbaik adalah 25:1, dan rasio jam debug adalah 28:1. Juga, kemampuan seorang insinyur perangkat lunak untuk sampai pada desain perangkat lunak dari deskripsi masalah sangat bervariasi sehubungan dengan parameter kualitas dan waktu. Pengetahuan teknis di bidang proyek (domain knowledge) merupakan faktor penting yang menentukan produktivitas seseorang untuk proyek tertentu, dan kualitas produk yang ia kembangkan. Seorang programmer yang memiliki pengetahuan menyeluruh tentang aplikasi database (misalnya MIS) dapat menjadi developer komunikasi data yang buruk. Kurangnya keakraban dengan area aplikasi dapat mengakibatkan rendahnya produktivitas dan kualitas produk yang buruk.

Karena pengembangan perangkat lunak adalah aktivitas kelompok, sangat penting bagi developer perangkat lunak untuk memiliki tiga jenis keterampilan komunikasi utama—Lisan, Tertulis, dan Interpersonal. developer perangkat lunak tidak hanya perlu berkomunikasi secara efektif dengan rekan satu timnya (misalnya ulasan, penelusuran, dan komunikasi tim lainnya) tetapi mungkin juga harus berkomunikasi dengan pelanggan untuk mengumpulkan persyaratan produk. Keterampilan interpersonal yang buruk menghambat aktivitas vital ini dan sering kali muncul sebagai kualitas produk yang buruk dan produktivitas yang rendah. developer perangkat lunak juga kadang-kadang diminta untuk membuat presentasi kepada manajer dan pelanggan. Ini membutuhkan keterampilan komunikasi yang berbeda (keterampilan komunikasi lisan). developer perangkat lunak juga diharapkan untuk mendokumentasikan pekerjaannya (desain, kode, pengujian, dll.) serta menulis manual pengguna, manual pelatihan, manual instalasi, manual pemeliharaan, dll. Ini membutuhkan keterampilan komunikasi tertulis yang baik.

Tingkat motivasi developer perangkat lunak adalah faktor penting lain yang berkontribusi pada kualitas dan produktivitas kerjanya. Meskipun tidak ada studi sistematis yang dilaporkan dalam hal ini, secara umum disepakati bahwa developer yang cerdas pun dapat berubah menjadi berkinerja buruk ketika mereka kekurangan motivasi. developer rata-rata yang dapat bekerja dengan satu jalur pikiran dapat mengungguli developer lain. Tetapi motivasi adalah fenomena kompleks yang membutuhkan kontrol yang cermat. Bagi sebagian besar developer perangkat lunak, insentif yang lebih tinggi dan kondisi kerja yang lebih baik hanya memiliki pengaruh terbatas pada tingkat motivasi mereka. Motivasi sebagian besar ditentukan oleh sifat-sifat pribadi, keluarga dan latar belakang sosial, dll.

3.14 MANAJEMEN RISIKO

Setiap proyek rentan terhadap sejumlah besar risiko. Tanpa manajemen risiko yang efektif, bahkan proyek yang direncanakan dengan sangat cermat pun bisa gagal. Risiko adalah setiap peristiwa atau keadaan yang tidak menguntungkan yang diantisipasi yang dapat terjadi saat proyek sedang berlangsung. Kita perlu membedakan antara risiko yang merupakan masalah yang mungkin terjadi dari masalah yang saat ini sedang dihadapi oleh sebuah proyek. Jika risiko menjadi nyata, masalah yang diantisipasi menjadi kenyataan dan tidak lagi menjadi risiko. Jika risiko menjadi nyata, hal itu dapat berdampak buruk pada proyek dan menghambat keberhasilan dan penyelesaian proyek tepat waktu. Oleh karena itu, manajer proyek perlu mengantisipasi dan mengidentifikasi berbagai risiko yang rentan terhadap suatu proyek, sehingga rencana kontinjensi dapat disiapkan sebelumnya untuk menampung setiap risiko. Dalam konteks ini, manajemen risiko bertujuan untuk mengurangi kemungkinan suatu risiko menjadi nyata serta mengurangi dampak dari suatu risiko yang menjadi nyata. Manajemen risiko terdiri dari tiga aktivitas penting—identifikasi risiko, penilaian risiko, dan mitigasi risiko.

Identifikasi resiko

Manajer proyek perlu mengantisipasi risiko dalam suatu proyek sedini mungkin. Segera setelah risiko diidentifikasi, rencana manajemen risiko yang efektif dibuat, sehingga kemungkinan dampak risiko diminimalkan. Jadi, identifikasi risiko sejak dini itu penting. Identifikasi risiko agak mirip dengan manajer proyek yang mencatat mimpi buruknya. Misalnya, manajer proyek mungkin khawatir apakah vendor yang Anda minta untuk mengembangkan modul tertentu mungkin tidak menyelesaikan pekerjaan mereka tepat waktu, apakah mereka akan menyerahkan pekerjaan berkualitas buruk, apakah beberapa personel kunci Anda mungkin meninggalkan organisasi, dll. Semua risiko seperti itu yang mungkin mempengaruhi proyek harus diidentifikasi dan didaftar.

Sebuah proyek dapat dikenakan berbagai macam risiko. Agar dapat secara sistematis mengidentifikasi risiko-risiko penting yang mungkin mempengaruhi suatu proyek, penting untuk mengkategorikan risiko-risiko ke dalam kelas-kelas yang berbeda. Manajer proyek kemudian dapat memeriksa risiko mana dari setiap kelas yang relevan dengan proyek. Ada tiga kategori utama risiko yang dapat mempengaruhi proyek perangkat lunak: risiko proyek, risiko teknis, dan risiko bisnis.

Risiko proyek: Risiko proyek menyangkut berbagai bentuk anggaran, jadwal, personel, sumber daya, dan masalah terkait pelanggan. Risiko proyek yang penting adalah selip jadwal. Karena, perangkat lunak tidak berwujud, sangat sulit untuk memantau dan mengendalikan proyek perangkat lunak. Sangat sulit untuk mengendalikan sesuatu yang tidak dapat dilihat. Untuk setiap proyek manufaktur, seperti pembuatan mobil, manajer proyek dapat melihat produk terbentuk. Misalnya, dia dapat melihat bahwa mesinnya dipasang, setelah itu pintu dipasang, mobilnya dicat, dll. Dengan demikian dia dapat secara akurat menilai kemajuan pekerjaan dan mengendalikannya, jika dia menemukan ada aktivitas yang sedang berlangsung di suatu tempat. lebih lambat dari yang direncanakan. Ketidaktampakan produk yang sedang dikembangkan merupakan alasan penting mengapa banyak proyek perangkat lunak mengalami risiko tergelincirnya jadwal.

Risiko teknis: Risiko teknis menyangkut potensi desain, implementasi, antarmuka, pengujian, dan masalah pemeliharaan. Risiko teknis juga mencakup spesifikasi yang ambigu, spesifikasi yang tidak lengkap, spesifikasi yang berubah, ketidakpastian teknis, dan keusangan teknis. Sebagian besar risiko teknis terjadi karena kurangnya pengetahuan tim pengembangan tentang produk. **Risiko bisnis:** Jenis risiko ini mencakup risiko membangun produk unggulan yang tidak diinginkan siapa pun, kehilangan komitmen anggaran, dll.

Klasifikasi risiko dalam sebuah proyek

Contoh 3.12 Mari kita perhatikan produk komunikasi bergerak berbasis satelit yang dibahas dalam Studi Kasus 2.2. Manajer proyek dapat mengidentifikasi beberapa risiko dalam proyek ini. Mari kita mengklasifikasikannya dengan tepat.

- Bagaimana jika biaya proyek meningkat dan melampaui apa yang diperkirakan?: **Risiko proyek.**
- Bagaimana jika ponsel yang dikembangkan menjadi terlalu besar ukurannya untuk dibawa dengan nyaman?: **Risiko bisnis.**
- Bagaimana jika kemudian diketahui bahwa tingkat radiasi yang berasal dari ponsel berbahaya bagi manusia?: **Risiko bisnis**.
- Bagaimana jika hand-off panggilan antar satelit menjadi terlalu sulit untuk diterapkan?: **Risiko teknis.**

Agar dapat berhasil meramalkan dan mengidentifikasi berbagai risiko yang mungkin memengaruhi proyek perangkat lunak, sebaiknya Anda memiliki daftar bencana perusahaan. Daftar ini akan berisi semua peristiwa buruk yang telah terjadi pada proyek perangkat lunak perusahaan selama bertahun-tahun termasuk peristiwa yang dapat diletakkan di depan pintu pelanggan. Daftar ini dapat dibaca oleh manajer proyek untuk mengetahui beberapa risiko yang mungkin rentan terhadap proyek. Daftar bencana seperti itu telah ditemukan untuk membantu dalam melakukan analisis risiko yang lebih baik.

Tugas beresiko

Tujuan penilaian risiko adalah untuk menentukan peringkat risiko dalam hal potensi kerusakannya. Untuk penilaian risiko, pertama-tama setiap risiko harus dinilai dengan dua cara:

- Kemungkinan suatu risiko menjadi nyata (r).
- Konsekuensi dari masalah yang terkait dengan risiko itu (s).

Berdasarkan kedua faktor tersebut, maka prioritas masing-masing risiko dapat dihitung sebagai berikut:

$$p = r \times s$$

dimana, p adalah prioritas risiko yang harus ditangani, r adalah probabilitas risiko menjadi nyata, dan s adalah tingkat keparahan kerusakan yang disebabkan karena risiko menjadi nyata. Jika semua risiko yang teridentifikasi diprioritaskan, maka risiko yang paling mungkin dan merusak dapat ditangani terlebih dahulu dan prosedur pengurangan risiko yang lebih komprehensif dapat dirancang untuk risiko tersebut.

Mitigasi risiko

Setelah semua risiko yang teridentifikasi dari suatu proyek telah dinilai, rencana dibuat untuk memuat risiko yang paling merusak dan paling mungkin terjadi terlebih dahulu. Berbagai jenis risiko memerlukan prosedur penahanan yang berbeda. Faktanya, sebagian besar risiko memerlukan kecerdikan yang cukup besar di pihak manajer proyek dalam menangani risiko.

Ada tiga strategi utama untuk pengendalian risiko:

- **Hindari risiko**: Risiko dapat dihindari dengan beberapa cara. Risiko sering muncul karena kendala proyek dan dapat dihindari dengan memodifikasi kendala yang sesuai. Berbagai kategori kendala yang biasanya menimbulkan risiko adalah:
    - Risiko terkait proses: Risiko ini muncul karena jadwal kerja, anggaran, dan pemanfaatan sumber daya yang agresif.
    - Risiko terkait produk: Risiko ini muncul karena komitmen terhadap fitur produk yang menantang (misalnya waktu respons satu detik, dll.), kualitas, keandalan, dll.
    - Risiko terkait teknologi: Risiko ini muncul karena komitmen untuk menggunakan teknologi tertentu (misalnya, komunikasi satelit).

Beberapa contoh penghindaran risiko adalah sebagai berikut: Berdiskusi dengan pelanggan untuk mengubah persyaratan untuk mengurangi ruang lingkup pekerjaan, memberikan insentif kepada developer untuk menghindari risiko pergantian tenaga kerja, dll.

- **Transfer risiko:** Strategi ini melibatkan mendapatkan komponen berisiko yang dikembangkan oleh pihak ketiga, membeli perlindungan asuransi, dll.
- **Pengurangan risiko**: Ini melibatkan perencanaan cara untuk menahan kerusakan akibat risiko. Misalnya, jika ada risiko bahwa beberapa personel kunci akan pergi, perekrutan baru mungkin direncanakan. Teknik pengurangan risiko yang paling penting untuk risiko teknis adalah membangun prototipe yang mencoba teknologi yang Anda coba gunakan. Misalnya, jika Anda menggunakan kompiler untuk mengenali perintah pengguna, Anda harus membuat kompiler untuk bahasa perintah yang kecil dan sangat primitif terlebih dahulu.

Ada beberapa strategi untuk mengatasi risiko. Untuk memilih strategi yang paling tepat untuk menangani risiko, manajer proyek harus mempertimbangkan biaya penanganan risiko dan pengurangan risiko yang sesuai. Untuk ini kita dapat menghitung leverage risiko dari berbagai risiko. Leverage risiko adalah selisih eksposur risiko dibagi dengan biaya untuk mengurangi risiko. Lebih formal,

$$Leverage\ risiko = \frac{Eksposur\ resiko\ sebelum\ dikurangi - eksposur\ resiko\ setelah\ dikurangi}{biaya\ pengurangan}$$

Meskipun kita sudah mengidentifikasi tiga cara umum untuk menangani risiko, penanganan risiko yang efektif tidak dapat dicapai hanya dengan mengikuti prosedur yang ditetapkan secara mekanis, tetapi membutuhkan banyak kecerdikan dari pihak manajer proyek. Sebagai contoh, mari kita pertimbangkan opsi yang tersedia untuk memuat jenis risiko penting yang terjadi di banyak proyek perangkat lunak—yaitu selip jadwal.

Contoh penanganan risiko selip jadwal

Risiko yang berkaitan dengan selip jadwal muncul terutama karena sifat perangkat lunak yang tidak berwujud. Untuk proyek seperti membangun rumah, progresnya dapat dengan mudah dilihat dan dinilai oleh manajer proyek. Jika dia menemukan bahwa proyek tersebut tertinggal, maka tindakan korektif dapat dimulai. Mengingat bahwa pengembangan perangkat lunak itu sendiri tidak terlihat, langkah pertama dalam mengelola risiko tergelincirnya jadwal, adalah meningkatkan visibilitas produk perangkat lunak. Visibilitas produk perangkat lunak dapat ditingkatkan dengan menghasilkan dokumen yang relevan selama proses pengembangan dan membuat dokumen ini ditinjau oleh tim yang sesuai.

Tonggak sejarah harus ditempatkan secara berkala untuk memberi manajer indikasi kemajuan yang teratur. Penyelesaian fase proses pembangunan yang diikuti tidak perlu menjadi satu-satunya tonggak. Setiap fase dapat dipecah menjadi tugas-tugas berukuran wajar dan tonggak dapat dikaitkan dengan tugas-tugas ini. Sebuah tonggak tercapai, setelah dokumentasi yang dihasilkan sebagai bagian dari tugas rekayasa perangkat lunak diproduksi dan berhasil ditinjau. Tonggak sejarah tidak perlu ditempatkan untuk setiap aktivitas. Aturan praktis perkiraan b adalah untuk menetapkan tonggak setiap 10 sampai 15 hari. Jika tonggak ditempatkan terlalu dekat satu sama lain maka biaya overhead dalam mengelola tonggak akan terlalu banyak.

3.15 MANAJEMEN KONFIGURASI PERANGKAT LUNAK

Hasil (juga disebut sebagai kiriman) dari upaya pengembangan perangkat lunak yang besar biasanya terdiri dari sejumlah besar objek, misalnya, kode sumber, dokumen desain, dokumen SRS, dokumen uji, manual pengguna, dll. Objek ini biasanya disebut dan dimodifikasi oleh sejumlah developer perangkat lunak sepanjang siklus hidup perangkat lunak. Keadaan setiap objek yang dapat dikirim berubah seiring dengan kemajuan perkembangan dan juga saat bug dideteksi dan diperbaiki. Konfigurasi perangkat lunak adalah keadaan semua hasil proyek pada titik waktu apa pun; dan manajemen konfigurasi perangkat lunak berkaitan dengan pelacakan dan pengendalian konfigurasi perangkat lunak secara efektif selama siklus hidupnya. Saat perangkat lunak diubah, revisi dan versi baru dibuat. Sebelum kita membahas manajemen konfigurasi, kita harus jelas tentang perbedaan antara versi dan revisi produk perangkat lunak.

Revisi perangkat lunak versus versi

Versi baru dari perangkat lunak dibuat ketika ada perubahan signifikan dalam fungsionalitas, teknologi, atau perangkat keras yang digunakannya, dll. Di sisi lain, rilis baru dibuat jika hanya ada perbaikan bug, peningkatan kecil pada fungsionalitas , kegunaan, dll. Bahkan pengiriman awal mungkin terdiri dari beberapa versi dan lebih banyak versi mungkin ditambahkan nanti. Misalnya, satu versi paket komputasi matematis mungkin berjalan di mesin berbasis Unix, versi lain di Microsoft Windows, dan seterusnya. Saat perangkat lunak dirilis dan digunakan oleh pelanggan, ditemukan kesalahan yang perlu diperbaiki. Peningkatan fungsi perangkat lunak mungkin juga diperlukan. Rilis perangkat lunak baru adalah sistem yang ditingkatkan yang dimaksudkan untuk menggantikan yang lama. Seringkali sistem digambarkan sebagai versi m, rilis n; atau hanya mn. Secara formal, relasi histori adalah versi

yang dapat didefinisikan antar objek. Relasi ini dapat dibagi menjadi dua subrelasi yaitu revisi dari dan merupakan varian dari.

Kebutuhan Manajemen Konfigurasi Perangkat Lunak

Ada beberapa alasan untuk meletakkan objek di bawah manajemen konfigurasi. Berikut ini adalah beberapa masalah penting yang dapat muncul, jika manajemen konfigurasi tidak digunakan: setiap developer perangkat lunak memiliki salinan pribadi dari suatu objek (misalnya kode sumber). Ketika seorang developer membuat perubahan pada salinan lokalnya, ia diharapkan untuk memberitahukan perubahan yang telah ia buat kepada developer lain, sehingga perubahan yang diperlukan dalam antarmuka dapat dilakukan secara seragam di semua modul. Namun, tidak hanya akan menghabiskan waktu yang berharga dari pengembang, tetapi sering kali developer mungkin membuat perubahan pada antarmuka di salinan lokalnya dan lupa untuk memberi tahu rekan satu tim tentang perubahan tersebut. Ini membuat salinan objek yang berbeda tidak konsisten. Akhirnya, ketika modul yang berbeda diintegrasikan, itu tidak berfungsi. Oleh karena itu, ketika beberapa anggota tim bekerja untuk mengembangkan aplikasi, mereka perlu mengerjakan satu salinan aplikasi, jika tidak, inkonsistensi dapat muncul.

Masalah yang terkait dengan akses bersamaan: Mungkin alasan paling penting untuk manajemen konfigurasi adalah untuk mengontrol akses ke berbagai objek yang dapat dikirim. Kecuali jika disiplin ketat diberlakukan terkait pembaruan dan penyimpanan objek yang berbeda, beberapa masalah dapat muncul. Asumsikan bahwa hanya satu salinan modul program yang dipertahankan, dan beberapa developer sedang mengerjakannya. Dua developer dapat secara bersamaan melakukan perubahan pada fungsi yang berbeda dari modul yang sama, dan sambil menyimpan saling menimpa. Masalah serupa dapat terjadi untuk objek yang dapat dikirim lainnya.

Menyediakan lingkungan pengembangan yang stabil: Ketika pekerjaan proyek sedang berlangsung, anggota tim membutuhkan lingkungan yang stabil untuk membuat kemajuan. Misalkan satu developer mencoba mengintegrasikan modul A, dengan modul B dan C; karena jika developer modul C terus mengubah C; ini bisa sangat membuat frustrasi jika perubahan pada modul C memaksa kompilasi ulang modul. Ketika manajemen konfigurasi yang efektif diterapkan, manajer membekukan objek untuk membentuk garis dasar. Garis dasar adalah status semua objek di bawah kendali konfigurasi. Ketika salah satu objek di bawah kontrol konfigurasi diubah, garis dasar baru akan terbentuk.

Ketika setiap anggota tim perlu mengubah salah satu objek di bawah kontrol konfigurasi, dia akan diberikan salinan item dasar. Pemohon membuat perubahan pada salinan pribadinya. Hanya setelah pemohon selesai dengan semua modifikasi pada salinan pribadinya, konfigurasi diperbarui dan garis dasar baru segera terbentuk. Ini menetapkan dasar bagi orang lain untuk digunakan dan diandalkan. Juga, baseline dapat diarsipkan secara berkala (pengarsipan berarti menyalin ke tempat yang aman seperti penyimpanan jarak jauh), sehingga baseline terakhir dapat dipulihkan ketika terjadi bencana.

Sistem akuntansi dan pemeliharaan informasi status: Sistem akuntansi menunjukkan melacak siapa yang membuat perubahan tertentu pada suatu objek dan kapan perubahan itu dilakukan.

Menangani varian: Adanya varian produk perangkat lunak menyebabkan beberapa masalah khusus. Misalkan Anda memiliki beberapa varian dari modul yang sama, dan menemukan bahwa ada bug di salah satunya. Kemudian, itu harus diperbaiki di semua versi dan revisi. Untuk melakukannya secara efisien, Anda tidak perlu memperbaikinya di setiap versi dan revisi perangkat lunak secara terpisah. Membuat perubahan pada satu program harus tercermin dengan tepat di semua versi dan revisi yang relevan.

Aktivitas Manajemen Konfigurasi

Manajemen konfigurasi dilakukan melalui dua kegiatan utama:

- **Identifikasi konfigurasi**: Ini melibatkan memutuskan bagian mana dari sistem yang harus dilacak.
- **Kontrol konfigurasi**: Ini memastikan bahwa perubahan pada sistem terjadi dengan lancar. Biasanya, manajer proyek melakukan aktivitas manajemen konfigurasi dengan menggunakan alat manajemen konfigurasi. Selain itu, alat manajemen konfigurasi membantu melacak berbagai objek yang dapat dikirim, sehingga manajer proyek dapat dengan cepat dan jelas menentukan status proyek saat ini. Alat manajemen konfigurasi memungkinkan developer untuk mengubah berbagai komponen secara terkendali.

Identifikasi konfigurasi

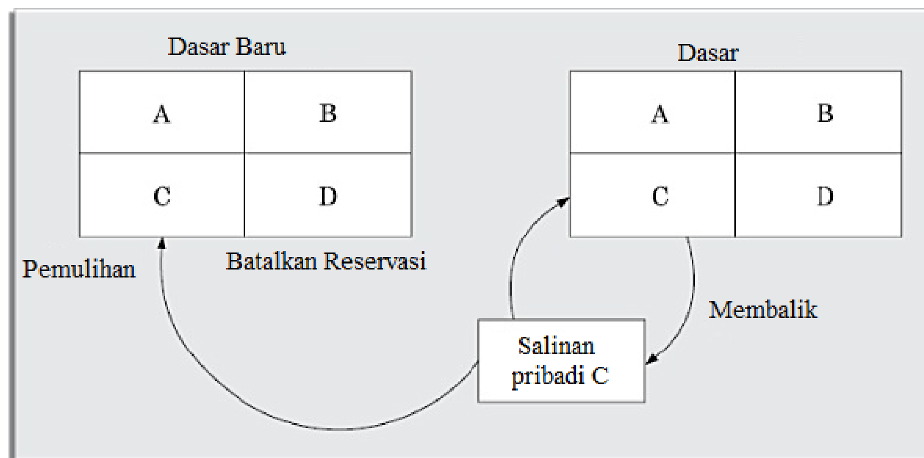
Manajer proyek biasanya mengklasifikasikan objek yang terkait dengan pengembangan perangkat lunak ke dalam tiga kategori utama—terkendali, pra-kontrol, dan tidak terkendali. Objek yang dikontrol adalah objek yang sudah berada di bawah kontrol konfigurasi. Anggota tim harus mengikuti beberapa prosedur formal untuk mengubahnya. Objek yang dikontrol sebelumnya belum berada di bawah kontrol konfigurasi, tetapi pada akhirnya akan berada di bawah kontrol konfigurasi. Objek yang tidak dikontrol tidak tunduk pada kontrol konfigurasi. Objek yang dapat dikontrol mencakup objek yang dikontrol dan yang telah dikontrol sebelumnya. Objek yang dapat dikontrol biasanya meliputi:

- Dokumen spesifikasi persyaratan
- Dokumen desain
- Alat yang digunakan untuk membangun sistem, seperti compiler, linker, lexical analyser, parser, dll.
- Kode sumber untuk setiap modul
- Kasus uji
- Laporan masalah

Rencana manajemen konfigurasi ditulis selama fase perencanaan proyek. Ini mencantumkan semua objek yang dikendalikan. Manajer yang mengembangkan rencana harus mencapai keseimbangan antara mengendalikan terlalu banyak, dan mengendalikan terlalu sedikit. Jika terlalu banyak dikendalikan, overhead karena manajemen konfigurasi meningkat ke tingkat yang tidak masuk akal. Di sisi lain, mengendalikan terlalu sedikit dapat menyebabkan kebingungan dan inkonsistensi ketika sesuatu berubah.

Kontrol konfigurasi

Kontrol konfigurasi adalah proses mengelola perubahan pada objek yang dikontrol. Bagian kontrol konfigurasi dari sistem manajemen konfigurasi yang paling langsung memengaruhi operasi developer sehari-hari. Kontrol konfigurasi hanya mengizinkan perubahan yang diizinkan pada objek yang dikontrol untuk terjadi dan mencegah perubahan yang tidak sah. Untuk mengubah objek yang dikendalikan seperti modul, developer bisa mendapatkan salinan pribadi dari modul dengan operasi cadangan (lihat Gambar 3.18). Alat manajemen konfigurasi memungkinkan hanya satu orang untuk memesan modul setiap saat. Setelah objek dicadangkan, itu tidak mengizinkan orang lain untuk memesan modul ini sampai modul yang dicadangkan dipulihkan. Jadi, dengan mencegah lebih dari satu developer untuk secara bersamaan memesan modul, masalah yang terkait dengan akses bersamaan diselesaikan.



Gambar 3.18 Operasi pencadangan dan pemulihan dalam kontrol konfigurasi.

Mari kita lihat bagaimana sebuah objek di bawah kontrol konfigurasi dapat diubah. developer yang perlu mengubah modul terlebih dahulu membuat permintaan cadangan. Setelah perintah cadangan berhasil dijalankan, salinan pribadi modul dibuat di direktori lokalnya. Kemudian, dia melakukan semua perubahan yang diperlukan pada salinan pribadinya. Setelah menyelesaikan semua perubahan yang diperlukan dengan memuaskan, perubahan tersebut perlu dipulihkan dalam repositori manajemen konfigurasi. Namun, memulihkan modul yang diubah ke konfigurasi sistem memerlukan izin dari *change control board* (CCB). CCB biasanya dibentuk dari antara anggota tim pengembangan. Untuk setiap perubahan yang perlu dilakukan, CCB meninjau perubahan yang dilakukan pada objek yang dikontrol dan mengesahkan beberapa hal tentang perubahan tersebut:

1. Perubahan memiliki motivasi yang baik.
2. Pengembang telah mempertimbangkan dan mendokumentasikan dampak dari perubahan tersebut.
3. Perubahan berinteraksi dengan baik dengan perubahan yang dibuat oleh developer lain.
4. Orang yang tepat (CCB) telah memvalidasi perubahan, misalnya, seseorang telah menguji kode yang diubah, dan telah memverifikasi bahwa perubahan tersebut konsisten dengan persyaratan.

Papan kontrol perubahan (CCB) terdengar seperti sekelompok orang. Namun, kecuali untuk proyek yang sangat besar, fungsi dewan kendali perubahan biasanya dilakukan oleh manajer proyek itu sendiri atau beberapa anggota senior dari tim pengembangan. Setelah CCB meninjau perubahan modul, manajer proyek memperbarui baseline lama melalui operasi pemulihan (lihat Gambar 3.18). Alat kontrol konfigurasi tidak mengizinkan developer untuk mengganti objek yang telah dia pesan dengan salinan lokalnya kecuali dia mendapat otorisasi dari CCB. Dengan membatasi kemampuan developer untuk mengganti objek yang dicadangkan, lingkungan yang stabil tercapai. Karena alat manajemen konfigurasi memungkinkan hanya satu developer untuk bekerja pada satu modul pada satu waktu, masalah penimpaan yang tidak disengaja dihilangkan. Juga, karena hanya manajer yang dapat memperbarui baseline setelah persetujuan CCB, perubahan yang tidak disengaja pada item konfigurasi dihilangkan.

Sistem kontrol kode sumber (SCCS) dan RCS

SCCS dan RCS adalah dua alat manajemen konfigurasi populer yang tersedia di sebagian besar sistem Unix. SCCS atau RCS dapat digunakan untuk mengontrol dan mengelola berbagai versi file teks. SCCS dan RCS tidak menangani file biner (yaitu, file yang dapat

dieksekusi, dokumen, file yang berisi diagram, dll.) SCCS dan RCS menyediakan cara yang efisien untuk menyimpan versi yang meminimalkan jumlah ruang disk yang ditempati. Misalkan, MOD modul hadir dalam 3 versi—MOD1.1, MOD1.2 dan MOD1.3. Kemudian, SCCS dan RCS menyimpan modul asli MOD1.1 bersama dengan perubahan yang diperlukan untuk mengubah MOD1.1 menjadi MOD1.2 dan MOD1.2 menjadi MOD1.3. Perubahan yang diperlukan untuk mengubah setiap file dasar ke versi berikutnya disimpan dan disebut delta. Alasan utama di balik menyimpan delta daripada menyimpan file revisi penuh adalah untuk menghemat ruang disk.

Fasilitas kontrol perubahan yang disediakan oleh SCCS dan RCS mencakup kemampuan untuk memasukkan batasan pada kumpulan individu yang dapat membuat versi baru, dan fasilitas untuk memeriksa komponen masuk dan keluar (yaitu, operasi cadangan dan pemulihan). developer individu memeriksa komponen dan memodifikasinya. Setelah mereka membuat semua perubahan yang diperlukan pada modul dan setelah perubahan ditinjau, mereka memeriksa modul yang diubah ke SCCS atau RCS. Revisi dilambangkan dengan angka dalam urutan menaik, misalnya 1.1, 1.2, 1.3, dll. Juga dimungkinkan untuk membuat varian atau revisi komponen dengan membuat percabangan dalam riwayat pengembangan.

3.16 RENCANA LAINNYA

Selain estimasi biaya, penjadwalan, dan rencana kepegawaian, manajer proyek merencanakan beberapa hal lain selama tahap perencanaan proyek. Tugas penting pada tahap ini adalah pemilihan model proses pengembangan yang sesuai. Kita telah membahas di Bab 2 bahwa masalah yang berbeda memerlukan baik mengadopsi model proses yang sama sekali berbeda atau menyesuaikan model proses standar yang diadopsi oleh perusahaan. Misalnya, pekerjaan pengembangan rutin mungkin memerlukan model air terjun untuk diikuti, dan proyek yang ambisius dan menantang secara teknis mungkin memerlukan model pengembangan evolusioner atau prototipe untuk diadopsi. Juga, tergantung pada jenis proyek, beberapa fase siklus hidup dapat dihilangkan, dimodifikasi, atau fase baru dapat ditambahkan ke model siklus hidup yang dipilih. Misalnya, untuk proyek pemeliharaan perangkat lunak, fase desain dapat dimodifikasi ke tahap modifikasi desain. Jadi selama tahap perencanaan proyek, manajer proyek mungkin harus memilih model proses yang sesuai dan melakukan penyesuaian yang diperlukan.

3.17 RINGKASAN

- Dalam bab ini, kita memeriksa tanggung jawab utama seorang manajer proyek perangkat lunak. Saya menyebutkan bahwa berbagai tanggung jawab manajer proyek dapat diklasifikasikan ke dalam dua kelas besar:
  - Perencanaan proyek
  - Pemantauan dan pengendalian proyek.
- Kegiatan perencanaan proyek dilakukan sebelum kegiatan pembangunan dimulai. Aktivitas perencanaan tersebut adaah estimasi, penjadwalan, dan penempatan staf.
- Rencana lain yang harus dilakukan manajer proyek selama tahap perencanaan proyek, seperti analisis risiko, rencana konfigurasi, dan penyesuaian proses.
- Kegiatan pemantauan dan pengendalian proyek dilakukan setelah pekerjaan pembangunan dimulai.
- Meskipun beberapa teknik sistematis tersedia bagi manajer proyek untuk melakukan perencanaan proyek; untuk pemantauan dan pengendalian proyek, pengalaman dan penilaian subjektif sangat penting.

- Konfigurasi perangkat lunak adalah keadaan item yang dapat dikirim pada setiap titik waktu. Tanpa manajemen konfigurasi yang tepat, sebuah proyek akan mengalami banyak jenis masalah. Untuk alasan ini, manajemen konfigurasi perangkat lunak merupakan persyaratan penting dan wajib di hampir semua prinsip jaminan kualitas perangkat lunak.

3.18 LATIHAN

1. Pilih opsi yang benar:
   a. Usaha diukur dengan menggunakan salah satu dari unit berikut:Orang
      i. Orang-bulan
      ii. Bulan
      iii. Rupiah
   b. Model estimasi COCOMO dapat digunakan untuk memperkirakan salah satu dari berikut ini:
      i. LOK
      ii. Usaha
      iii. Poin fungsi
      iv. Kepadatan cacat
   c. Apa urutan yang benar di mana manajer proyek perangkat lunak memperkirakan berbagai parameter proyek saat menggunakan COCOMO:
      i. Biaya, usaha, durasi, ukuran
      ii. Biaya, durasi, usaha, ukuran
      iii. Ukuran, usaha, durasi, biaya
      iv. Ukuran, biaya, usaha, durasi
   d. Manakah dari berikut ini yang BUKAN merupakan faktor untuk "Baris kode" yang dianggap sebagai metrik ukuran yang buruk:
      i. Ini bergantung pada bahasa pemrograman.
      ii. Ini menghukum pengkodean yang efisien dan ringkas
      iii. Hal ini tergantung programmer.
      iv. Tergantung pada kompleksitas persyaratan.
   e. Manakah dari parameter proyek berikut yang pertama kali diperkirakan oleh manajer proyek:
      i. Biaya
      ii. Usaha
      iii. Ukuran
      iv. Durasi
   f. Manakah dari bagan berikut ini yang paling berguna untuk menguraikan aktivitas proyek menjadi tugas-tugas yang lebih kecil yang dapat dikelola dengan lebih bermakna:
      i. Grafik PERT
      ii. Grafik GANTT
      iii. Representasi jaringan tugas
      iv. Struktur rincian pekerjaan
   g. Manakah dari berikut ini yang merupakan contoh model estimasi biaya multivariabel?
      i. COCOMO Dasar
      ii. COCOMO Menengah
      iii. COCOMO Lengkap

    iv. Teknik Delphi
h. Jika sebuah produk perangkat lunak berukuran S membutuhkan waktu m bulan untuk dikembangkan, maka menurut model estimasi COCOMO, berapa lama (dalam bulan) waktu yang dibutuhkan untuk mengembangkan produk berukuran 2 × S?
    i. Lebih dari 2 × m bulan
    ii. Lebih dari 3 × m bulan
    iii. Kurang dari 2 × m bulan
    iv. Lebih dari 4 × m bulan
i. Manakah dari pernyataan berikut yang benar tentang model COCOMO.
    i. Biaya adalah atribut paling mendasar dari produk perangkat lunak, berdasarkan ukuran dan durasi proyek yang diukur.
    ii. Ukuran adalah atribut paling mendasar dari produk perangkat lunak, berdasarkan mana biaya dan durasi proyek diukur.
    iii. Upaya adalah atribut paling mendasar dari produk perangkat lunak, berdasarkan ukuran dan biaya proyek yang diukur.
    iv. Durasi adalah atribut paling mendasar dari produk perangkat lunak, berdasarkan ukuran dan upaya proyek yang diukur.
j. Untuk proyek pengembangan perangkat lunak tertentu, estimasi upaya 100 orang-bulan diperoleh dengan menggunakan model COCOMO. Ini berarti bahwa proyek harus diselesaikan dengan:
    i. Mempekerjakan 100 orang selama 1 bulan
    ii. Mempekerjakan 1 orang selama 100 bulan
    iii. Mempekerjakan 10 orang selama 10 bulan
    iv. Jumlah orang yang dipekerjakan selama fase proyek yang berbeda akan sesuai dengan distribusi Raleigh
k. Manakah dari berikut ini yang paling menggambarkan manajemen konfigurasi dalam rekayasa perangkat lunak?
    i. Manajemen pengaturan parameter konfigurasi dalam perangkat lunak.
    ii. Manajemen objek yang mengontrol pengaturan parameter konfigurasi sistem.
    iii. Manajemen status berbagai hasil proyek.
    iv. Konfigurasi kegiatan pengelolaan tergantung pada jenis proyek.
l. Bagaimana "versi" aplikasi berbeda dari "rilis"-nya?
    i. Rilis adalah perubahan kecil dari rilis sebelumnya.
    ii. Versi adalah perubahan kecil yang dibuat untuk rilis sebelumnya.
    iii. Rilis pada dasarnya sama dengan versi.
    iv. Rilis adalah rilis yang tersedia untuk pelanggan sedangkan versi untuk penggunaan internal.
m. Jika suatu proyek sudah tertunda, maka penambahan tenaga kerja untuk menyelesaikannya paling cepat adalah:
    i. Selalu kontra produktif
    ii. Dapat membantu sampai batas yang sangat terbatas
    iii. Cara paling efektif untuk mengatasi situasi
    iv. Dapat menyebabkan penyelesaian proyek dalam waktu singkat

2. Tuliskan lima tanggung jawab utama seorang manajer proyek perangkat lunak.
3. Identifikasi faktor-faktor yang membuat proyek perangkat lunak jauh lebih sulit untuk dikelola, dibandingkan dengan banyak jenis proyek lainnya seperti proyek untuk meletakkan jalan beton 100 km di jalan non-beton yang ada.

4. Pada titik mana dalam siklus hidup pengembangan perangkat lunak (SDLC), aktivitas manajemen proyek dimulai? Kapan ini berakhir? Identifikasi kegiatan manajemen proyek yang penting.
5. Apa yang dimaksud dengan 'ukuran' proyek perangkat lunak? Mengapa seorang manajer proyek perlu memperkirakan ukuran proyek? Bagaimana perkiraan ukurannya?
6. Apa yang Anda pahami dengan perencanaan jendela geser? Jelaskan dengan menggunakan beberapa contoh jenis proyek yang sangat cocok dengan bentuk perencanaan ini. Apakah perencanaan jendela geser sesuai untuk proyek kecil? Apa kelebihannya dibandingkan perencanaan konvensional?
7. Apa yang Anda pahami dengan visibilitas produk dalam konteks pengembangan perangkat lunak? Mengapa penting untuk meningkatkan visibilitas produk selama pengembangan perangkat lunak? Bagaimana visibilitas produk dapat ditingkatkan.
8. Apa saja kategori yang berbeda dari proyek pengembangan perangkat lunak menurut model estimasi COCOMO? Berikan contoh proyek pengembangan produk perangkat lunak yang termasuk dalam masing-masing kategori ini.
9. Jelaskan secara singkat perbedaan utama antara model estimasi COCOMO asli dan model estimasi COCOMO 2.
10. Apakah yang Anda maksud: ukuran proyek Apa metrik populer untuk mengukur ukuran proyek? Bagaimana ukuran proyek dapat diperkirakan selama tahap perencanaan proyek?
11. Jelaskan secara singkat estimasi ukuran proyek menggunakan Delphi dan teknik penilaian ahli. Bandingkan keuntungan dan kerugian dari dua teknik estimasi ukuran proyek berikut — penilaian ahli dan teknik Delphi.
12. Mengapa sulit untuk secara akurat memperkirakan upaya yang diperlukan untuk menyelesaikan suatu proyek? Jelaskan secara singkat berbagai metode estimasi usaha yang tersedia. Mana yang paling disarankan untuk digunakan dan mengapa?
13. Untuk jumlah baris kode yang sama dan ukuran tim pengembangan yang sama, beri peringkat proyek perangkat lunak berikut sesuai dengan perkiraan waktu pengembangannya. Sebutkan secara singkat alasan di balik jawaban Anda.
    a. Editor teks
    b. Sistem daftar gaji karyawan
    c. Sistem operasi untuk komputer baru
14. Sebagai manajer proyek perangkat lunak untuk mengembangkan produk untuk aplikasi bisnis, jika Anda memperkirakan upaya yang diperlukan untuk penyelesaian proyek adalah 50 orang-bulan, dapatkah Anda menyelesaikan proyek dengan mempekerjakan 50 developer untuk jangka waktu satu bulan? Justifikasi jawaban Anda.
15. Jelaskan secara singkat model COCOMO 2. Dalam aspek apa itu merupakan peningkatan dari model COCOMO asli.
16. Nyatakan apakah pernyataan berikut BENAR atau SALAH. Berikan alasan untuk jawaban Anda.
    a. Sebagai manajer proyek, akan bermanfaat bagi Anda untuk mengurangi durasi proyek hingga setengahnya asalkan pelanggan setuju untuk membayar kebutuhan tenaga kerja yang meningkat.
    b. Organisasi perangkat lunak mencapai pemanfaatan tenaga kerja yang lebih efisien dengan mengadopsi struktur organisasi berbasis proyek dibandingkan dengan organisasi berbasis fungsi.
    c. Untuk pengembangan produk yang sama, semakin besar ukuran tim pengembangan perangkat lunak, semakin cepat pengembangan produk. (untuk

kesederhanaan, asumsikan bahwa semua developer sama-sama mahir dan memiliki pengalaman yang persis sama).

d. Jumlah personel pengembangan yang dibutuhkan untuk setiap proyek pengembangan perangkat lunak dapat ditentukan dengan membagi total (perkiraan) upaya dengan total (perkiraan) durasi proyek.
e. Organisasi tim demokratis sangat cocok untuk menangani proyek yang kompleks dan menantang.
f. Dimungkinkan untuk melakukan manajemen konfigurasi untuk proyek perangkat lunak tanpa menggunakan alat otomatis.
g. Menurut model COCOMO, biaya adalah atribut paling mendasar dari produk perangkat lunak, berdasarkan ukuran dan upaya yang diperkirakan.
h. Ukuran proyek, seperti yang digunakan dalam COCOMO adalah ukuran kode akhir yang dapat dieksekusi dalam byte.
i. Teknik estimasi Delphi biasanya memberikan estimasi ukuran proyek yang lebih akurat dibandingkan dengan teknik expert judgement.
j. Struktur tim yang demokratis adalah yang paling cocok dibandingkan dengan jenis struktur tim lainnya. untuk mengembangkan produk perangkat lunak yang sangat besar.
k. Ketika tugas di sepanjang jalur kritis diselesaikan dalam waktu kurang dari perkiraan semula, itu akan menghasilkan penyelesaian proyek secara keseluruhan lebih cepat.
l. Tugas yang diselesaikan setelah waktu penyelesaian terakhir (LF) akan muncul sebagai penundaan penyelesaian proyek pada waktu yang sesuai.
m. Manajer proyek biasanya menggunakan grafik GANTT untuk melakukan alokasi sumber daya, sedangkan grafik PERT digunakan untuk memantau dan mengendalikan kemajuan proyek.

17. Apa yang dimaksud dengan tonggak sejarah dalam pengembangan perangkat lunak? Mengapa dianggap bermanfaat untuk memiliki tonggak dalam pengembangan perangkat lunak?
18. Bagaimana urutan estimasi berikut dalam teknik estimasi COCOMO: biaya, tenaga, durasi, ukuran? Mewakili urutan prioritas di antara aktivitas-aktivitas ini menggunakan diagram jaringan tugas.
19. Jelaskan mengapa waktu pengembangan produk perangkat lunak dengan ukuran tertentu tetap hampir sama, terlepas dari apakah itu jenis organik, semi-terpisah, atau tertanam.
20. Jelaskan mengapa menurut model COCOMO, ketika ukuran perangkat lunak ditingkatkan dua kali lipat, waktu untuk mengembangkan produk biasanya meningkat kurang dari dua kali lipat.
21. Misalkan Anda telah memperkirakan waktu pengembangan nominal produk perangkat lunak berukuran sedang menjadi 5 bulan. Anda juga telah memperkirakan bahwa itu akan menelan biaya Rp. 50.000 untuk mengembangkan produk perangkat lunak. Sekarang, pelanggan datang dan memberi tahu Anda bahwa dia ingin Anda mempercepat waktu pengiriman sebesar 10 persen. Berapa biaya tambahan yang akan Anda kenakan kepada pelanggan untuk pengiriman yang dipercepat ini? Terlepas dari apakah Anda membutuhkan lebih sedikit waktu atau lebih banyak waktu untuk mengembangkan produk, Anda pada dasarnya mengembangkan produk yang sama. Lalu mengapa upaya itu bergantung pada durasi Anda mengembangkan produk?
22. Jelaskan bagaimana model Putnam dapat digunakan untuk menghitung perubahan biaya proyek dengan durasi proyek. Apa kerugian utama menggunakan model Putnam

untuk menghitung biaya tambahan yang dikeluarkan karena kompresi jadwal? Bagaimana Anda bisa mengatasinya?

23. Misalkan Anda sedang mengembangkan produk perangkat lunak jenis organik. Anda telah memperkirakan ukuran produk menjadi sekitar 100.000 baris kode. Hitung usaha nominal dan waktu pengembangan.
24. Untuk program C berikut, perkirakan ukuran panjang dan volume Halstead. Bandingkan ukuran panjang dan volume Halstead dengan ukuran LOC.
25. Apa yang diwakili oleh metrik volume Halstead secara konseptual? Bagaimana menurut Halstead upaya bergantung pada volume program?
26. Apa keuntungan relatif menggunakan LOC atau metrik titik fungsi untuk mengukur ukuran produk perangkat lunak untuk perencanaan proyek perangkat lunak?
27. Buat daftar kekurangan penting dari LOC untuk digunakan sebagai metrik ukuran perangkat lunak untuk melaksanakan estimasi proyek.
28. Jelaskan mengapa menambahkan lebih banyak tenaga kerja ke proyek yang sudah terlambat membuatnya lebih lambat.
29. Apa yang Anda pahami dengan rincian kerja dalam manajemen proyek? Mengapa rincian pekerjaan penting untuk manajemen proyek yang efektif? Bagaimana pemecahan pekerjaan dicapai? Masalah apa yang mungkin terjadi jika tugas dipecah menjadi granularity yang terlalu halus atau tugas dipecah menjadi granularity yang terlalu kasar?
30. Jelaskan kapan Anda harus menggunakan bagan PERT dan kapan Anda harus menggunakan bagan Gantt saat Anda menjalankan tugas sebagai manajer proyek.
31. Bagaimana Gantt chart berguna dalam manajemen proyek perangkat lunak? Masalah apa yang mungkin dihadapi, jika pemantauan dan pengendalian proyek dilakukan dengan menggunakan bagan Gantt?
32. Apa yang dimaksud dengan dasar dalam konteks manajemen konfigurasi perangkat lunak? Jelaskan bagaimana baseline dapat diperbarui untuk membentuk baseline baru?
33. Misalkan Anda adalah manajer proyek perangkat lunak. Jelaskan hanya dengan menggunakan satu atau dua kalimat mengapa Anda tidak harus menghitung jumlah developer yang dibutuhkan untuk proyek sebagai pembagian sederhana dari perkiraan usaha (dalam orang-bulan) dengan perkiraan durasi nominal (dalam bulan).
34. Buat daftar item penting yang harus didiskusikan oleh dokumen rencana manajemen proyek perangkat lunak (SPMP).
35. Misalkan Anda adalah manajer proyek dari tim pengembangan produk yang besar dan Anda harus membuat pilihan antara organisasi tim programmer yang demokratis dan kepala, yang mana yang akan Anda adopsi untuk tim Anda? Jelaskan alasan di balik jawaban Anda.
36. Bandingkan keuntungan relatif dari pendekatan fungsional dan proyek dari organisasi pusat pengembangan. Misalkan Anda adalah chief executive officer (CEO) dari pusat pengembangan perangkat lunak. Struktur organisasi mana yang akan Anda pilih untuk organisasi Anda? Mengapa?
37. Sebutkan berbagai cara di mana tim pengembangan perangkat lunak diatur. Untuk pengembangan produk komunikasi seluler berbasis satelit yang menantang, jenis organisasi tim proyek mana yang akan Anda rekomendasikan? Justifikasi jawaban Anda.
38. apakah Anda setuju dengan pernyataan berikut? “Sedikit, jika ada, organisasi di dunia nyata yang murni fungsional, proyek, atau matriks.” Justifikasi jawaban Anda.

39. Jelaskan keuntungan dari organisasi fungsional atas organisasi proyek. Juga jelaskan mengapa rumah pengembangan perangkat lunak lebih memilih untuk menggunakan organisasi proyek daripada organisasi fungsional.
40. Dalam konteks manajemen konfigurasi perangkat lunak, jawablah pertanyaan berikut:
    a. Apa yang Anda pahami tentang konfigurasi perangkat lunak?
    b. Apa yang dimaksud dengan manajemen konfigurasi perangkat lunak?
    c. Bagaimana Anda dapat mengatur konfigurasi perangkat lunak (hanya menyebutkan nama aktivitas utama yang terlibat)?
    d. Mengapa manajemen konfigurasi perangkat lunak penting untuk keberhasilan proyek pengembangan produk perangkat lunak besar (tulis hanya alasan penting)?
    e. Apa itu *change control board* (CCB) dan apa perannya dalam manajemen konfigurasi perangkat lunak?
41. Mengapa proyek perangkat lunak lebih rentan terhadap selip jadwal dibandingkan dengan jenis proyek lainnya?
42. Di unit apa Anda dapat mengukur produktivitas tim pengembangan perangkat lunak? Sebutkan tiga faktor penting yang mempengaruhi produktivitas tim pengembangan perangkat lunak.
43. Sebutkan tiga jenis risiko umum yang mungkin dialami oleh proyek perangkat lunak biasa. Jelaskan bagaimana Anda dapat mengidentifikasi risiko yang rentan terhadap proyek Anda. Misalkan Anda adalah manajer proyek dari proyek pengembangan perangkat lunak yang besar, tunjukkan langkah-langkah utama yang akan Anda ikuti untuk mengelola risiko dalam proyek perangkat lunak Anda.
44. Keterlambatan jadwal adalah bentuk risiko yang sangat umum yang harus dihadapi oleh hampir setiap manajer proyek. Jelaskan dalam 3 sampai 4 kalimat bagaimana Anda akan mengelola risiko tergelincirnya jadwal sebagai manajer proyek dari proyek menengah.
45. Jelaskan bagaimana Anda dapat memilih teknik pengurangan risiko terbaik ketika ada banyak cara untuk mengurangi risiko.
46. Apa jenis risiko penting yang mungkin diderita proyek? Bagaimana Anda mengidentifikasi risiko yang rentan terhadap proyek selama proyek tahap perencanaan proyek?
47. Sebagai manajer proyek, identifikasi karakteristik yang akan Anda cari dalam developer perangkat lunak saat mencoba memilih personel untuk tim Anda.
48. Apa itu pemrograman tanpa ego? Bagaimana itu bisa terwujud?
49. Benarkah produk perangkat lunak selalu dapat dikembangkan lebih cepat dengan memiliki tim pengembangan yang lebih besar (Anda dapat berasumsi bahwa semua developer sama-sama mahir dan memiliki pengalaman yang persis sama)? Justifikasi jawaban Anda.
50. Misalkan Anda telah ditunjuk sebagai manajer proyek dari sebuah proyek besar, identifikasi kegiatan yang akan Anda lakukan untuk merencanakan proyek Anda. Jelaskan urutan di mana Anda akan melakukan aktivitas ini dengan menggunakan notasi jaringan tugas. Apa saja faktor yang membuat sulit untuk memperkirakan secara akurat biaya proyek perangkat lunak?
51. Misalkan Anda adalah manajer proyek dari proyek pengembangan besar. Manajemen puncak menginformasikan bahwa Anda harus berurusan dengan ukuran tim yang tetap (yaitu, jumlah developer yang konstan) selama proyek Anda berlangsung. Apa dampak dari keputusan ini terhadap proyek Anda?

52. Angka produktivitas rata-rata industri untuk developer hanya 10 LOC/hari. Apa alasan produktivitas rendah seperti itu? Bisakah kita menghubungkan ini dengan keterampilan pemrograman developer yang buruk?
53. Apa yang Anda pahami tentang manajemen konfigurasi perangkat lunak? Bagaimana pelaksanaannya? Masalah apa yang akan Anda hadapi jika Anda mengembangkan beberapa versi produk yang sama sesuai dengan permintaan klien, dan Anda tidak menggunakan alat manajemen konfigurasi apa pun?
54. Apa perbedaan antara revisi dan versi produk perangkat lunak? Apa yang Anda pahami dengan istilah change control dan version control? Mengapa ini diperlukan? Jelaskan bagaimana perubahan dan kontrol versi dicapai menggunakan alat manajemen konfigurasi.
55. Diskusikan bagaimana SCCS atau RCS dapat digunakan untuk mengelola konfigurasi kode sumber secara efisien. Apa saja kekurangan dari SCCS dan RCS?
56. Pertimbangkan proyek perangkat lunak dengan 5 tugas T1-T5. Durasi 5 tugas (dalam hari) masing-masing adalah 15, 10, 12, 25 dan 10. T2 dan T4 dapat dimulai ketika T1 selesai. T3 dapat dimulai ketika T2 selesai. T5 dapat dimulai ketika T3 dan T4 selesai. Kapan tanggal mulai terakhir tugas T3? Berapa waktu senggang tugas T4?
57. Mengapa manajer proyek perlu menguraikan tugas proyek menggunakan struktur rincian kerja (WBS)? Sampai tingkat perincian apa tugas-tugas didekomposisi? Jelaskan jawaban Anda.
58. Misalkan Anda adalah manajer proyek dari tim kecil yang mengembangkan aplikasi bisnis. Asumsikan bahwa tim Anda memiliki pengalaman dalam mengembangkan beberapa produk serupa. Jika Anda diminta untuk membuat pilihan antara organisasi tim programmer yang demokratis dan kepala, mana yang akan Anda adopsi untuk tim Anda? Jelaskan alasan di balik jawaban Anda.
59. Apa yang Anda pahami tentang risiko proyek? Bagaimana risiko dapat diidentifikasi secara efektif oleh manajer proyek? Bagaimana risiko dapat dikelola?
60. Misalkan Anda ditunjuk sebagai manajer proyek dari sebuah proyek untuk mengembangkan produk perangkat lunak pengolah kata komersial yang menyediakan fitur yang sebanding dengan perangkat lunak MS-WORD, mengembangkan struktur rincian kerja (WBS). Jelaskan jawaban Anda.
61. Apa saja parameter proyek berbeda yang menentukan biaya proyek? Apa faktor penting yang membuat sulit untuk secara akurat memperkirakan biaya proyek perangkat lunak? Jika Anda seorang manajer proyek yang mengajukan penawaran untuk pengembangan produk kepada pelanggan, apakah Anda akan mengutip perkiraan biaya menggunakan COCOMO sebagai harga dalam penawaran Anda? Jelaskan jawaban Anda.

BAB 4
ANALISIS KEBUTUHAN DAN SPESIFIKASI

Semua model siklus hidup yang digerakkan oleh rencana menetapkan persyaratan yang tepat dari pelanggan harus dipahami dan didokumentasikan sebelum mulai mengembangkan perangkat lunak. Di masa lalu, banyak proyek menderita karena developer mulai menerapkan sesuatu tanpa menentukan apakah mereka membangun keinginan pelanggan. Memulai pekerjaan pengembangan tanpa memahami dan mendokumentasikan persyaratan dengan benar meningkatkan jumlah perubahan berulang dalam fase siklus hidup selanjutnya, dan dengan demikian secara mengkhawatirkan mendorong biaya pengembangan. Ini juga menjadi dasar ketidakpuasan pelanggan dan perselisihan sengit antara pelanggan-pengembang dan pertempuran hukum yang berlarut-larut. Tidak heran jika developer berpengalaman menganggap analisis persyaratan dan spesifikasi sebagai fase yang sangat penting dari siklus hidup pengembangan perangkat lunak dan melakukannya dengan sangat hati-hati.

Developer berpengalaman membutuhkan banyak waktu untuk memahami persyaratan yang tepat dari pelanggan dan untuk mendokumentasikannya dengan cermat. Mereka tahu bahwa tanpa pemahaman yang jelas tentang masalah dan dokumentasi yang tepat, sangatlah tidak mungkin untuk mengembangkan solusi yang memuaskan. Untuk semua jenis proyek pengembangan perangkat lunak, ketersediaan dokumen persyaratan kualitas yang baik telah diakui sebagai faktor kunci dalam keberhasilan penyelesaian proyek. Dokumen persyaratan yang baik tidak hanya membantu membentuk pemahaman yang jelas tentang berbagai fitur yang diperlukan dari perangkat lunak, tetapi juga berfungsi sebagai dasar untuk berbagai aktivitas yang dilakukan selama fase siklus hidup selanjutnya.

Ketika software dikembangkan dalam mode kontrak untuk beberapa organisasi lain (yaitu, proyek outsourcing), peran penting yang dimainkan oleh dokumentasi persyaratan yang tepat tidak dapat dilebih-lebihkan. Bahkan ketika sebuah organisasi mengembangkan produk perangkat lunak generik, situasinya tidak jauh berbeda karena beberapa personel dari departemen pemasaran organisasi itu sendiri bertindak sebagai pelanggan. Oleh karena itu, untuk semua jenis proyek pengembangan perangkat lunak, perumusan persyaratan yang tepat dan dokumentasi yang efektif sangat penting. Namun, untuk proyek layanan perangkat lunak yang sangat kecil, metode agile menganjurkan pengembangan persyaratan secara bertahap.

Ikhtisar analisis persyaratan dan fase spesifikasi

Analisis persyaratan dan fase spesifikasi dimulai setelah tahap studi kelayakan selesai dan proyek dinyatakan layak secara finansial dan layak secara teknis. Analisis persyaratan dan fase spesifikasi berakhir ketika dokumen spesifikasi persyaratan telah dikembangkan dan ditinjau. Dokumen spesifikasi kebutuhan biasanya disebut sebagai dokumen spesifikasi kebutuhan perangkat lunak (SRS). Tujuan dari fase analisis kebutuhan dan spesifikasi dapat dinyatakan secara singkat sebagai berikut. Tujuan dari fase analisis dan spesifikasi kebutuhan adalah untuk memahami dengan jelas kebutuhan pelanggan dan secara sistematis mengatur persyaratan ke dalam dokumen yang disebut dokumen Spesifikasi Persyaratan Perangkat Lunak (SRS).

Siapa yang melakukan analisis dan spesifikasi persyaratan?

Aktivitas analisis dan spesifikasi kebutuhan biasanya dilakukan oleh beberapa anggota tim pengembangan yang berpengalaman dan biasanya mengharuskan mereka untuk meluangkan waktu di lokasi pelanggan. Para insinyur yang mengumpulkan dan menganalisis

persyaratan pelanggan dan kemudian menulis dokumen spesifikasi persyaratan dikenal sebagai analis sistem dalam bahasa industri perangkat lunak. Analis sistem mengumpulkan data yang berkaitan dengan produk yang akan dikembangkan dan menganalisis data yang dikumpulkan untuk membuat konsep apa yang sebenarnya perlu dilakukan. Setelah memahami persyaratan pengguna yang tepat, analis menganalisis persyaratan untuk menghilangkan inkonsistensi, anomali, dan ketidaklengkapan. Mereka kemudian melanjutkan untuk menulis dokumen spesifikasi persyaratan perangkat lunak (SRS). Dokumen SRS merupakan hasil akhir dari tahap analisis kebutuhan dan spesifikasi.

Bagaimana dokumen SRS divalidasi?

Setelah dokumen SRS siap, pertama-tama ditinjau secara internal oleh tim proyek untuk memastikan bahwa dokumen tersebut secara akurat menangkap semua kebutuhan pengguna, dan dapat dimengerti, konsisten, tidak ambigu, dan lengkap. Dokumen SRS kemudian diberikan kepada pelanggan untuk ditinjau. Setelah pelanggan meninjau dokumen SRS dan menyetujuinya, dokumen tersebut menjadi dasar untuk semua kegiatan pengembangan di masa depan dan juga berfungsi sebagai dokumen kontrak antara pelanggan dan organisasi pengembangan.

Apa kegiatan utama yang dilakukan selama fase analisis kebutuhan dan spesifikasi?

Analisis persyaratan dan fase spesifikasi terutama melibatkan pelaksanaan dua kegiatan penting berikut:

- Pengumpulan dan analisis persyaratan
- Spesifikasi kebutuhan

4.1 PENGUMPULAN DAN ANALISIS PERSYARATAN

Set lengkap persyaratan hampir tidak pernah tersedia dalam bentuk dokumen tunggal dari pelanggan. Bahkan, tidak realistis untuk mengharapkan pelanggan menghasilkan dokumen komprehensif yang berisi deskripsi yang tepat tentang apa yang diinginkannya. Selanjutnya, persyaratan lengkap jarang diperoleh dari perwakilan pelanggan mana pun. Oleh karena itu, persyaratan harus dikumpulkan oleh analis dari beberapa sumber sedikit demi sedikit. Persyaratan yang dikumpulkan ini perlu dianalisis untuk menghilangkan beberapa jenis masalah yang sering terjadi pada persyaratan yang telah dikumpulkan sedikit demi sedikit dari sumber yang berbeda. Secara konseptual kita dapat membagi pengumpulan persyaratan dan aktivitas analisis menjadi dua tugas terpisah:

- Pengumpulan persyaratan
- Analisa Kebutuhan

Pengumpulan Persyaratan

Pengumpulan persyaratan juga dikenal sebagai elisitasi persyaratan. Tujuan utama dari tugas pengumpulan persyaratan adalah untuk mengumpulkan persyaratan dari para pemangku kepentingan. Pemangku kepentingan adalah sumber persyaratan dan biasanya seseorang, atau sekelompok orang yang secara langsung atau tidak langsung berkaitan dengan perangkat lunak. Pengumpulan persyaratan mungkin terdengar seperti tugas yang sederhana. Namun, dalam praktiknya sangat sulit untuk mengumpulkan semua informasi yang diperlukan dari sejumlah besar pemangku kepentingan dan dari informasi yang tersebar di beberapa bagian dokumen. Mengumpulkan persyaratan ternyata menjadi sangat menantang jika tidak ada model kerja dari perangkat lunak yang sedang dikembangkan.

Misalkan seorang pelanggan ingin mengotomatisasi beberapa aktivitas di organisasinya yang saat ini dilakukan secara manual. Dalam hal ini, model kerja sistem (yaitu, sistem manual) ada. Ketersediaan model kerja biasanya sangat membantu dalam pengumpulan persyaratan. Misalnya, jika proyek melibatkan otomatisasi aktivitas akuntansi

yang ada dari suatu organisasi, maka tugas analis sistem menjadi jauh lebih mudah karena ia dapat segera memperoleh formulir input dan output serta rincian prosedur operasional. Dalam konteks ini, pertimbangkan bahwa diperlukan untuk mengembangkan perangkat lunak untuk mengotomatisasi kegiatan pembukuan yang terlibat dalam pengoperasian kantor tertentu. Dalam hal ini, analis harus mempelajari bentuk input dan output dan kemudian memahami bagaimana output dihasilkan dari data input. Namun, jika sebuah proyek melibatkan pengembangan sesuatu yang baru yang tidak ada model kerjanya, maka aktivitas pengumpulan persyaratan menjadi semakin sulit. Dengan tidak adanya sistem kerja, lebih banyak imajinasi dan kreativitas diperlukan di pihak analis sistem.

Biasanya bahkan sebelum mengunjungi situs pelanggan, aktivitas pengumpulan persyaratan dimulai dengan mempelajari dokumen yang ada untuk mengumpulkan semua informasi yang mungkin tentang sistem yang akan dikembangkan. Selama kunjungan ke situs pelanggan, analis biasanya mewawancarai pengguna akhir dan perwakilan pelanggan,1melakukan aktivitas pengumpulan persyaratan seperti survei kuesioner, analisis tugas, analisis skenario, dan analisis formulir. Mengingat bahwa banyak pelanggan tidak paham komputer, mereka menggambarkan kebutuhan mereka dengan sangat samar. Analis yang baik berbagi pengalaman dan keahlian mereka dengan pelanggan dan memberikan sarannya untuk mendefinisikan fungsionalitas tertentu secara lebih komprehensif, membuat fungsionalitas lebih umum dan lebih lengkap. Berikut ini, adalah cara-cara penting untuk seorang analis berpengalaman dalam mengumpulkan persyaratan:

1. **Mempelajari dokumentasi yang ada**: Analis biasanya mempelajari semua dokumen yang tersedia mengenai sistem yang akan dikembangkan sebelum mengunjungi situs pelanggan. Pelanggan biasanya memberikan dokumen pernyataan tujuan (SoP) kepada pengembang. Biasanya dokumen-dokumen ini mungkin membahas isu-isu seperti konteks di mana perangkat lunak diperlukan, tujuan dasar, pemangku kepentingan, fitur perangkat lunak serupa yang dikembangkan di tempat lain, dll.
2. **Wawancara:** Biasanya, ada banyak kategori pengguna perangkat lunak yang berbeda. Setiap kategori pengguna biasanya memerlukan serangkaian fitur yang berbeda dari perangkat lunak. Oleh karena itu, penting bagi analis untuk terlebih dahulu mengidentifikasi berbagai kategori pengguna dan kemudian menentukan persyaratan masing-masing. Misalnya, kategori yang berbeda dari pengguna perangkat lunak otomatisasi perpustakaan dapat menjadi anggota perpustakaan, pustakawan, dan akuntan. Anggota perpustakaan ingin menggunakan perangkat lunak untuk menanyakan ketersediaan buku dan menerbitkan serta mengembalikan buku. Pustakawan mungkin ingin menggunakan perangkat lunak untuk menentukan buku yang terlambat, membuat rekening anggota, menghapus rekening anggota, dll. Personel rekening mungkin menggunakan perangkat lunak untuk meminta fungsionalitas mengenai aspek keuangan seperti total biaya yang dikumpulkan dari anggota, pengadaan buku belanja pegawai, belanja gaji pegawai, dan lain-lain.

Untuk mensistematisasikan metode pengumpulan kebutuhan ini, teknik Delphi dapat diikuti. Dalam teknik ini, analis mengkonsolidasikan persyaratan seperti yang dipahami olehnya dalam dokumen dan kemudian mengedarkannya untuk komentar dari berbagai kategori pengguna. Berdasarkan umpan balik mereka, dia menyempurnakan dokumennya. Prosedur ini diulang sampai pengguna yang berbeda menyetujui set persyaratan.

3. **Analisis tugas**: Pengguna biasanya memiliki tampilan kotak hitam dari perangkat lunak dan menganggap perangkat lunak sebagai sesuatu yang menyediakan

serangkaian layanan (fungsi). Layanan yang didukung oleh perangkat lunak juga disebut tugas. Sehingga, perangkat lunak melakukan berbagai tugas pengguna. Dalam konteks ini, analis mencoba mengidentifikasi dan memahami berbagai tugas yang harus dilakukan oleh perangkat lunak. Untuk setiap tugas yang diidentifikasi, analis mencoba merumuskan langkah-langkah berbeda yang diperlukan untuk mewujudkan fungsionalitas yang diperlukan dengan berkonsultasi dengan pengguna. Misalnya, untuk layanan penerbitan buku, langkahnya mungkin—mengotentikasi pengguna, memeriksa jumlah buku yang diterbitkan kepada pelanggan dan menentukan apakah jumlah buku maksimum yang dapat dipinjam anggota ini telah tercapai, memeriksa apakah buku telah dipesan. , posting rincian masalah buku di catatan anggota, dan akhirnya cetak slip masalah buku yang dapat ditunjukkan oleh anggota di konter keamanan untuk mengeluarkan buku dari tempat perpustakaan. Analisis tugas membantu analis untuk memahami seluk beluk berbagai tugas pengguna dan untuk mewakili setiap tugas sebagai hierarki subtugas.

Analisis skenario: Sebuah tugas dapat memiliki banyak skenario operasi. Skenario tugas yang berbeda dapat terjadi ketika tugas dipanggil dalam situasi yang berbeda. Untuk berbagai jenis skenario tugas, perilaku perangkat lunak dapat berbeda. Misalnya, skenario yang mungkin untuk tugas penerbitan buku dari perangkat lunak otomatisasi perpustakaan mungkin:

- Buku berhasil diterbitkan kepada anggota dan slip penerbitan buku dicetak.
- Buku ini dicadangkan, dan karenanya tidak dapat diterbitkan untuk anggota.

Jumlah maksimum buku yang dapat diterbitkan untuk anggota telah tercapai, dan tidak ada lagi buku yang dapat diterbitkan untuk anggota. Untuk berbagai tugas yang diidentifikasi, skenario eksekusi yang mungkin diidentifikasi dan rincian setiap skenario diidentifikasi dengan berkonsultasi dengan pengguna. Untuk setiap skenario yang diidentifikasi, detail mengenai respons sistem, kondisi pasti di mana skenario terjadi, dll. ditentukan dengan berkonsultasi dengan pengguna.

Analisis formulir: Analisis formulir adalah kegiatan pengumpulan persyaratan yang penting dan efektif yang dilakukan oleh analis, ketika proyek melibatkan otomatisasi sistem manual yang ada. Selama pengoperasian sistem manual, biasanya beberapa formulir harus diisi oleh para pemangku kepentingan, dan pada gilirannya mereka menerima beberapa pemberitahuan (biasanya formulir yang diisi secara manual). Pada form analysis form yang keluar dan format notifikasi yang dihasilkan dianalisa untuk menentukan data yang masuk ke sistem dan data yang keluar dari sistem. Untuk set input data yang berbeda ke sistem, bagaimana data input ini akan digunakan oleh sistem untuk menghasilkan data output yang sesuai ditentukan dari pengguna.

Studi kasus 4.1 Pengumpulan persyaratan untuk otomatisasi pekerjaan kantor di departemen CSE

Informasi akademik, inventaris, dan keuangan di departemen CSE (Ilmu Komputer dan Teknik) dari suatu institut tertentu dilakukan secara manual oleh dua pegawai kantor, seorang penjaga toko, dan dua petugas. Departemen memiliki kekuatan mahasiswa 500 dan kekuatan guru 30. Kepala departemen (HoD) ingin mengotomatisasi pekerjaan kantor. Mengingat rendahnya anggaran yang ia miliki, ia mempercayakan pekerjaan itu kepada tim relawan mahasiswa.

Untuk pengumpulan persyaratan, anggota tim yang bertanggung jawab untuk analisis dan spesifikasi persyaratan (analis) pertama kali diberi pengarahan oleh HoD tentang aktivitas spesifik yang akan diotomatisasi. HoD menyebutkan bahwa tiga aspek utama pekerjaan

kantor perlu diotomatisasi—aktivitas terkait toko, aktivitas penilaian siswa, dan aktivitas manajemen cuti siswa. Itu perlu bagi analis untuk memenuhi kategori pengguna lainnya. HoD memperkenalkan analis (seorang mahasiswa) kepada staf kantor. Analis pertama-tama berdiskusi dengan dua panitera mengenai tanggung jawab (tugas) khusus mereka yang diperlukan untuk diotomatisasi. Untuk setiap tugas, mereka meminta panitera untuk menjelaskan kepada mereka tentang langkah-langkah di mana ini dilakukan. Analis juga menanyakan tentang berbagai skenario yang mungkin muncul untuk setiap tugas. Analis mengumpulkan semua jenis formulir yang digunakan oleh mahasiswa dan staf departemen untuk mendaftarkan berbagai jenis informasi ke kantor (misalnya pendaftaran kursus siswa, penilaian kursus) atau permintaan untuk beberapa layanan tertentu (misalnya masalah item dari toko). Ia juga mengumpulkan sampel dari berbagai jenis dokumen (keluaran) yang sedang disiapkan oleh panitera. Beberapa di antaranya memiliki formulir cetak khusus yang diisi oleh juru tulis secara manual, dan yang lainnya dimasukkan menggunakan spreadsheet, dan kemudian dicetak pada printer laser. Untuk setiap formulir keluaran, analis berkonsultasi dengan juru tulis mengenai bagaimana entri yang berbeda ini dihasilkan dari data masukan.

Analis bertemu dengan penjaga toko dan menanyakan tentang prosedur masalah material, prosedur entri buku besar toko, dan prosedur untuk meningkatkan indentasi pada berbagai vendor. Dia juga mengumpulkan salinan dari semua formulir yang relevan yang digunakan oleh penjaga toko. Analis juga mewawancarai perwakilan mahasiswa dan fakultas. Karena diperlukan untuk mengotomatisasi aktivitas kantor yang ada, analis dapat dengan mudah memperoleh format yang tepat dari data input, data output, dan deskripsi yang tepat dari prosedur kantor yang ada.

Analisa Kebutuhan

Setelah pengumpulan persyaratan selesai, analis menganalisis persyaratan yang dikumpulkan untuk membentuk pemahaman yang jelas tentang persyaratan pelanggan yang tepat dan untuk menghilangkan masalah apa pun dalam persyaratan yang dikumpulkan. Wajar untuk mengharapkan bahwa data yang dikumpulkan dari berbagai pemangku kepentingan mengandung beberapa kontradiksi, ambiguitas, dan ketidaklengkapan, karena setiap pemangku kepentingan biasanya hanya memiliki pandangan sebagian dan tidak lengkap dari perangkat lunak. Oleh karena itu, perlu untuk mengidentifikasi semua masalah dalam persyaratan dan menyelesaikannya melalui diskusi lebih lanjut dengan pelanggan. Tujuan utama dari aktivitas analisis persyaratan adalah untuk menganalisis persyaratan yang dikumpulkan untuk menghilangkan semua ambiguitas, ketidaklengkapan, dan inkonsistensi dari persyaratan pelanggan yang dikumpulkan dan untuk mendapatkan pemahaman yang jelas tentang perangkat lunak yang akan dikembangkan.

Untuk melaksanakan analisis kebutuhan secara efektif, analis pertama perlu mengembangkan pemahaman yang jelas tentang masalah. Pertanyaan dasar berikut yang berkaitan dengan proyek harus dipahami dengan jelas oleh analis sebelum melakukan analisis:

- Apa masalahnya?
- Mengapa penting untuk menyelesaikan masalah?
- Apa sebenarnya input data ke sistem dan apa sebenarnya output data oleh sistem?
- Apa kemungkinan prosedur yang perlu diikuti untuk memecahkan masalah?
- Apa kemungkinan kompleksitas yang mungkin muncul saat memecahkan masalah?
- Jika ada perangkat lunak atau perangkat keras eksternal yang harus dihubungkan dengan perangkat lunak yang dikembangkan, lalu format pertukaran data apa yang harus dilakukan dengan sistem eksternal?

Setelah analis memahami persyaratan pelanggan yang tepat, ia melanjutkan untuk mengidentifikasi dan menyelesaikan berbagai masalah yang ia deteksi dalam persyaratan yang dikumpulkan. Selama analisis persyaratan, analis perlu mengidentifikasi dan menyelesaikan tiga jenis masalah utama dalam persyaratan:

- Anomali
- Inkonsistensi
- Ketidaklengkapan

Mari kita periksa berbagai jenis masalah persyaratan ini secara rinci.

Anomali: Ini adalah anomali adalah ambiguitas dalam persyaratan. Ketika suatu persyaratan anomali, beberapa interpretasi persyaratan itu mungkin. Setiap anomali dalam salah satu persyaratan dapat menyebabkan pengembangan sistem yang salah, karena persyaratan anomali dapat ditafsirkan dalam beberapa cara selama pengembangan. Berikut ini adalah dua contoh persyaratan anomali:

Contoh 4.1 Saat mengumpulkan persyaratan untuk aplikasi kontrol proses, persyaratan berikut diungkapkan oleh pemangku kepentingan tertentu: Ketika suhu menjadi tinggi, pemanas harus dimatikan. Harap dicatat bahwa kata-kata seperti “tinggi”, “rendah”, “bagus”, “buruk”, dll. adalah indikasi dari persyaratan yang ambigu karena tidak memiliki kuantifikasi dan dapat ditafsirkan secara subjektif. Jika ambang batas di mana suhu dapat dianggap tinggi tidak ditentukan, maka hal itu dapat ditafsirkan secara berbeda oleh developer yang berbeda.

Contoh 4.2 Dalam studi kasus 4.1, misalkan seorang pegawai kantor menjelaskan persyaratan berikut: selama penghitungan nilai akhir, jika ada mahasiswa yang mendapat nilai cukup rendah dalam satu semester, maka orang tuanya perlu diberitahu. Ini jelas merupakan persyaratan yang ambigu karena tidak memiliki kriteria yang didefinisikan dengan baik tentang apa yang dapat dianggap sebagai "nilai yang cukup rendah".

Inkonsistensi: Dua persyaratan dikatakan tidak konsisten, jika salah satu persyaratan bertentangan dengan yang lain. Sayap berikut adalah dua contoh persyaratan yang tidak konsisten:

Contoh 4.3 Pertimbangkan dua persyaratan berikut yang dikumpulkan dari dua pemangku kepentingan yang berbeda dalam proyek pengembangan aplikasi kontrol proses. Tungku harus dimatikan ketika suhu tungku naik di atas 500°C.

- Ketika suhu tungku naik di atas 500°C, pancuran air harus dinyalakan dan tungku harus tetap menyala.
- Persyaratan yang diungkapkan oleh kedua pemangku kepentingan tersebut jelas tidak konsisten.

Contoh 4.4 Dalam studi kasus 4.1 anggaplah salah satu juru tulis memberikan persyaratan berikut— seorang mahasiswa yang mendapatkan nilai gagal dalam tiga mata pelajaran atau lebih harus mengulang mata kuliah tersebut selama satu semester penuh, dan dia tidak dapat mengkredit mata kuliah lain mana pun saat mengulang mata kuliah tersebut. Misalkan petugas lain menyatakan persyaratan berikut — tidak ada ketentuan bagi mahasiswa mana pun untuk mengulang satu semester; mahasiswa harus menghapus subjek dengan mengambilnya sebagai mata pelajaran tambahan di semester berikutnya. Ada inkonsistensi yang jelas antara persyaratan yang diberikan oleh kedua pemangku kepentingan.

Ketidaklengkapan: Seperangkat persyaratan yang tidak lengkap adalah persyaratan di mana beberapa persyaratan telah diabaikan. Kurangnya fitur ini akan dirasakan oleh pelanggan jauh di kemudian hari, mungkin saat menggunakan perangkat lunak. Seringkali, ketidaklengkapan disebabkan oleh ketidakmampuan pelanggan untuk memvisualisasikan sistem yang akan dikembangkan dan untuk mengantisipasi semua fitur yang diperlukan. Analis

berpengalaman dapat mendeteksi sebagian besar fitur yang hilang ini dan menyarankannya kepada pelanggan untuk pertimbangan dan persetujuannya untuk dimasukkan ke dalam persyaratan. Berikut ini adalah dua contoh persyaratan yang tidak lengkap:

Contoh 4.5 Misalkan untuk studi kasus 4.1, salah satu panitera menyatakan sebagai berikut—Jika seorang mahasiswa memperoleh nilai rata-rata (IPK) kurang dari 6, maka orang tua mahasiswa harus diberitahu tentang kinerja yang disesalkan melalui (pos) surat maupun melalui email. Namun, pada pemeriksaan semua persyaratan, ditemukan bahwa tidak ada ketentuan yang dapat digunakan untuk memasukkan alamat pos atau email orang tua mahasiswa ke dalam sistem. Fitur yang memungkinkan memasukkan id email dan alamat pos orang tua mahasiswa hilang, sehingga membuat persyaratan tidak lengkap.

Contoh 4.6 Dalam perangkat lunak otomatisasi pabrik kimia, misalkan salah satu persyaratannya adalah jika suhu internal reaktor melebihi 200C maka bel alarm harus dibunyikan. Namun, pada pemeriksaan semua persyaratan, ditemukan bahwa tidak ada ketentuan untuk mengatur ulang bel alarm setelah suhu diturunkan di salah satu persyaratan. Ini jelas merupakan persyaratan yang tidak lengkap.

Dapatkah seorang analis mendeteksi semua masalah yang ada dalam persyaratan yang dikumpulkan?

Banyak inkonsistensi, anomali, dan ketidaklengkapan dapat dideteksi dengan mudah, sementara beberapa lainnya memerlukan studi yang terfokus pada persyaratan khusus. Namun, beberapa masalah dalam persyaratan bisa sangat halus dan luput dari pandangan mata yang paling berpengalaman sekalipun. Banyak dari anomali dan inkonsistensi halus ini dapat dideteksi, jika persyaratan ditentukan dan dianalisis menggunakan metode formal. Setelah sistem telah ditentukan secara formal, dapat secara sistematis (dan bahkan secara otomatis) dianalisis untuk menghilangkan semua masalah dari spesifikasi. Kita akan membahas konsep dasar spesifikasi sistem formal. Meskipun penggunaan teknik formal tidak tersebar luas, praktik saat ini adalah secara formal hanya menentukan bagian kritis keselamatan dari suatu sistem.

4.2 SPESIFIKASI KEBUTUHAN PERANGKAT LUNAK (SRS)

Setelah analis mengumpulkan semua informasi yang diperlukan mengenai perangkat lunak yang akan dikembangkan, dan telah menghilangkan semua ketidaklengkapan, inkonsistensi, dan anomali dari spesifikasi, ia mulai mengatur persyaratan secara sistematis dalam bentuk dokumen SRS. Dokumen SRS biasanya berisi semua kebutuhan pengguna dalam bentuk terstruktur meskipun informal. Di antara semua dokumen yang dihasilkan selama siklus hidup pengembangan perangkat lunak, dokumen SRS mungkin adalah dokumen yang paling penting dan paling sulit untuk ditulis. Salah satu alasan kesulitan ini adalah bahwa dokumen SRS diharapkan dapat memenuhi kebutuhan berbagai macam audiens.

Pengguna Dokumen SRS

Biasanya sejumlah besar orang yang berbeda membutuhkan dokumen SRS untuk tujuan yang sangat berbeda. Beberapa kategori penting pengguna dokumen SRS dan kebutuhan penggunaannya adalah sebagai berikut:

- **Pengguna, pelanggan, dan personel pemasaran**: Pemangku kepentingan ini perlu mengacu pada dokumen SRS untuk memastikan bahwa sistem seperti yang dijelaskan dalam dokumen akan memenuhi kebutuhan mereka. Ingat bahwa pelanggan mungkin bukan pengguna perangkat lunak, tetapi mungkin seseorang yang dipekerjakan atau ditunjuk oleh pengguna. Untuk produk generik, tenaga pemasaran perlu memahami persyaratan yang dapat mereka jelaskan kepada pelanggan.

- **Pengembang perangkat lunak**: developer perangkat lunak mengacu pada dokumen SRS untuk memastikan bahwa mereka mengembangkan persis apa yang dibutuhkan oleh pelanggan.
- **Insinyur uji**: Teknisi uji menggunakan dokumen SRS untuk memahami fungsionalitas, dan berdasarkan ini, tulis kasus uji untuk memvalidasi kerjanya. Mereka membutuhkan fungsionalitas yang diperlukan harus dijelaskan dengan jelas, dan data input dan output harus diidentifikasi dengan tepat.
- **Penulis dokumentasi pengguna**: Penulis dokumentasi pengguna perlu membaca dokumen SRS untuk memastikan bahwa mereka memahami fitur produk dengan cukup baik untuk dapat menulis manual pengguna.
- **Manajer proyek**: Manajer proyek mengacu pada dokumen SRS untuk memastikan bahwa mereka dapat memperkirakan biaya proyek dengan mudah dengan mengacu pada dokumen SRS dan berisi semua informasi yang diperlukan untuk merencanakan proyek.
- **Insinyur pemeliharaan**: Dokumen SRS membantu teknisi pemeliharaan memahami fungsionalitas yang didukung oleh sistem. Pengetahuan yang jelas tentang fungsionalitas dapat membantu mereka memahami desain dan kode. Juga, pemahaman yang tepat tentang fungsionalitas yang didukung memungkinkan mereka untuk menentukan modifikasi spesifik pada fungsionalitas sistem yang diperlukan untuk tujuan tertentu.

Banyak insinyur perangkat lunak dalam sebuah proyek menganggap dokumen SRS sebagai dokumen referensi. Namun, seringkali lebih tepat untuk menganggap dokumen SRS sebagai dokumentasi kontrak antara tim pengembangan dan pelanggan. Bahkan, dokumen SRS dapat digunakan untuk menyelesaikan perselisihan antara developer dan pelanggan yang mungkin timbul di masa mendatang. Dokumen SRS bahkan dapat digunakan sebagai dokumen hukum untuk menyelesaikan perselisihan antara pelanggan dan developer di pengadilan. Setelah pelanggan menyetujui dokumen SRS, tim pengembangan melanjutkan untuk mengembangkan perangkat lunak dan memastikan bahwa itu sesuai dengan semua persyaratan yang disebutkan dalam dokumen SRS.

Mengapa Menghabiskan Waktu dan Sumber Daya untuk Mengembangkan Dokumen SRS?

Dokumen SRS yang diformulasikan dengan baik menemukan berbagai penggunaan selain penggunaan utama yang dimaksudkan sebagai dasar untuk memulai pekerjaan pengembangan perangkat lunak. Fungsi penting dari dokumen SRS yang diformulasikan dengan baik adalah sebagai berikut:

- **Membentuk kesepakatan antara pelanggan dan pengembang**: Sebuah dokumen SRS yang baik menetapkan panggung bagi pelanggan untuk membentuk harapan mereka tentang perangkat lunak dan developer tentang apa yang diharapkan dari perangkat lunak.
  - **Mengurangi pengerjaan ulang di masa depan**: Proses persiapan dokumen SRS memaksa para pemangku kepentingan untuk memikirkan semua persyaratan sebelum desain dan pengembangan berlangsung. Ini mengurangi desain ulang, pengodean ulang, dan pengujian ulang nanti. Tinjauan yang cermat terhadap dokumen SRS dapat mengungkapkan kelalaian, kesalahpahaman, dan inkonsistensi di awal siklus pengembangan.
  - **Memberikan dasar untuk memperkirakan biaya dan jadwal**: Manajer proyek biasanya memperkirakan ukuran perangkat lunak dari analisis dokumen SRS. Berdasarkan perkiraan ini mereka membuat perkiraan lain seperti upaya yang diperlukan untuk mengembangkan perangkat lunak dan total biaya

pengembangan. Dokumen SRS juga berfungsi sebagai dasar negosiasi harga dengan pelanggan. Manajer proyek juga menggunakan dokumen SRS untuk penjadwalan kerja.

- **Menyediakan dasar untuk validasi dan verifikasi**: Dokumen SRS menyediakan dasar yang dapat digunakan untuk memeriksa kepatuhan perangkat lunak yang dikembangkan. Ini juga digunakan oleh para insinyur pengujian untuk membuat rencana pengujian.
- **Memfasilitasi perpanjangan masa depan:** Dokumen SRS biasanya berfungsi sebagai dasar untuk merencanakan peningkatan di masa depan.

Sebelum kita membahas tentang cara menulis dokumen SRS, pertama-tama kita membahas karakteristik dokumen SRS yang baik dan perangkap yang harus dihindari secara sadar saat menulis dokumen SRS.

Ciri-ciri Dokumen SRS yang Baik

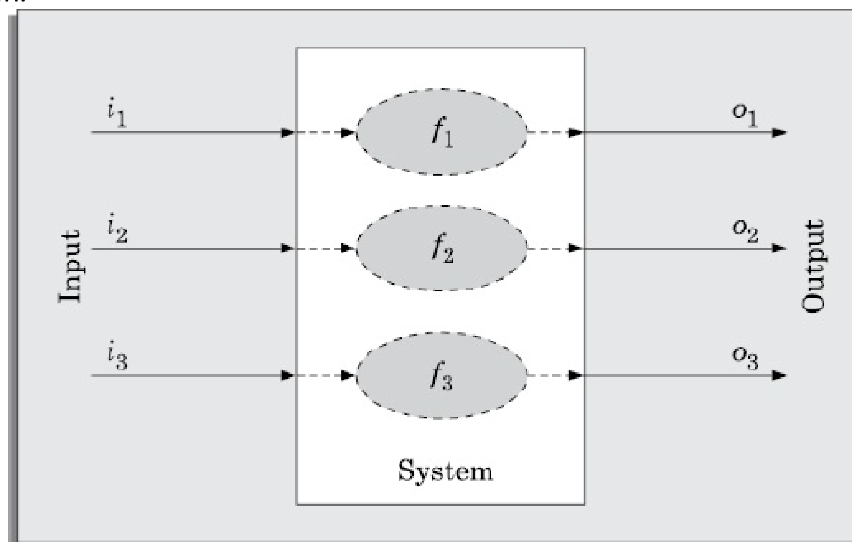
Keterampilan menulis dokumen SRS yang baik biasanya berasal dari pengalaman yang diperoleh dari menulis dokumen SRS untuk banyak proyek. Namun, analis harus menyadari kualitas yang diinginkan yang harus dimiliki setiap dokumen SRS yang baik. Praktik yang Direkomendasikan IEEE untuk Spesifikasi Persyaratan Perangkat Lunak[IEEE830] menjelaskan konten dan kualitas spesifikasi persyaratan perangkat lunak (SRS) yang baik. Beberapa kualitas yang diinginkan dari dokumen SRS adalah sebagai berikut:

- **Ringkas**: Dokumen SRS harus ringkas dan pada saat yang sama tidak ambigu, konsisten, dan lengkap. Deskripsi yang bertele-tele dan tidak relevan mengurangi keterbacaan dan juga meningkatkan kemungkinan kesalahan dalam dokumen.
- **Implementasi-independen**: SRS harus bebas dari desain dan keputusan implementasi kecuali keputusan tersebut mencerminkan persyaratan yang sebenarnya. Seharusnya hanya menentukan apa yang harus dilakukan sistem dan menahan diri untuk tidak menyatakan bagaimana melakukan ini. Ini berarti bahwa dokumen SRS harus menentukan perilaku sistem yang terlihat secara eksternal dan tidak membahas masalah implementasi. Tampilan ini dengan spesifikasi persyaratan yang ditulis, telah ditunjukkan pada Gambar 4.1. Perhatikan bahwa pada Gambar 4.1, dokumen SRS menjelaskan output yang dihasilkan untuk berbagai jenis input dan deskripsi pemrosesan yang diperlukan untuk menghasilkan output dari input (ditunjukkan dalam elips) dan kerja internal perangkat lunak tidak dibahas sama sekali .
- **Traceable**: Harus dimungkinkan untuk melacak kebutuhan spesifik ke elemen desain yang mengimplementasikannya dan sebaliknya. Demikian pula, harus dimungkinkan untuk melacak persyaratan ke segmen kode yang mengimplementasikannya dan kasus uji yang menguji persyaratan ini dan sebaliknya. Ketertelusuran juga penting untuk memverifikasi hasil fase sehubungan dengan fase sebelumnya dan untuk menganalisis dampak perubahan persyaratan pada elemen desain dan kode.
- **Dapat dimodifikasi:** Pelanggan sering mengubah persyaratan selama pengembangan pengembangan perangkat lunak karena berbagai alasan. Oleh karena itu, dalam praktiknya dokumen SRS mengalami beberapa kali revisi selama pengembangan perangkat lunak. Juga, dokumen SRS sering dimodifikasi setelah proyek selesai untuk mengakomodasi peningkatan dan evolusi di masa depan. Untuk mengatasi perubahan persyaratan, dokumen SRS harus mudah dimodifikasi. Untuk ini, dokumen SRS harus terstruktur dengan baik. Dokumen yang terstruktur dengan baik mudah dipahami dan dimodifikasi. Penjabaran

kebutuhan yang tersebar di banyak tempat dalam dokumen SRS mungkin tidak salah—tetapi cenderung membuat kebutuhan sulit untuk dipahami dan juga setiap modifikasi pada kebutuhan akan menjadi sulit karena akan membutuhkan perubahan yang harus dilakukan dalam jumlah besar. tempat dalam dokumen.

- **Identifikasi respons terhadap kejadian yang tidak diinginkan**: Dokumen SRS harus membahas respons sistem terhadap berbagai kejadian yang tidak diinginkan dan kondisi luar biasa yang mungkin timbul.
- **Dapat diverifikasi**: Semua persyaratan sistem seperti yang didokumentasikan dalam dokumen SRS harus dapat diverifikasi. Ini berarti bahwa seharusnya memungkinkan untuk merancang kasus uji berdasarkan deskripsi fungsionalitas, apakah persyaratan telah dipenuhi atau tidak dalam suatu implementasi. Persyaratan seperti "sistem harus ramah pengguna" tidak dapat diverifikasi. Di sisi lain, persyaratan—“Ketika nama buku dimasukkan, perangkat lunak harus menampilkan apakah buku tersebut tersedia untuk diterbitkan atau telah dipinjamkan” dapat diverifikasi. Setiap fitur dari sistem yang diperlukan yang tidak dapat diverifikasi harus dicantumkan secara terpisah di tujuan bagian implementasi dokumen SRS.

Dokumen SRS harus menjelaskan sistem yang akan dikembangkan sebagai kotak hitam, dan harus menentukan hanya perilaku sistem yang terlihat secara eksternal. Untuk alasan ini, dokumen SRS juga disebut spesifikasi kotak hitam dari perangkat lunak yang sedang dikembangkan.



Gambar 4.1 Tampilan kotak hitam dari suatu sistem saat melakukan serangkaian fungsi.

Atribut Dokumen SRS Buruk

Dokumen SRS yang ditulis oleh pemula sering mengalami berbagai masalah. Seperti dibahas sebelumnya, masalah yang paling merusak adalah ketidaklengkapan, ambiguitas, dan kontradiksi. Ada banyak jenis masalah lain yang mungkin dialami oleh dokumen spesifikasi. Dengan mengetahui masalah ini, seseorang dapat mencoba menghindarinya saat menulis dokumen SRS. Beberapa kategori masalah penting yang banyak dialami oleh dokumen SRS adalah sebagai berikut:

Over-spesifikasi: Ini terjadi ketika analis mencoba untuk mengatasi aspek "bagaimana" dalam dokumen SRS. Misalnya, dalam masalah otomatisasi perpustakaan, seseorang tidak boleh menentukan apakah catatan keanggotaan perpustakaan perlu disimpan diindeks pada

nama depan anggota atau pada nomor identifikasi (ID) anggota perpustakaan. Spesifikasi yang berlebihan membatasi kebebasan desainer dalam mencapai solusi desain yang baik.

Referensi ke depan: Seseorang tidak boleh mengacu pada aspek-aspek yang akan dibahas lebih lanjut dalam dokumen SRS. Referensi ke depan sangat mengurangi keterbacaan spesifikasi.

Pemikiran angan-angan: Jenis masalah ini menyangkut deskripsi aspek-aspek yang akan sulit untuk diterapkan.

Kebisingan: Istilah kebisingan mengacu pada keberadaan materi yang tidak secara langsung relevan dengan proses pengembangan perangkat lunak. Sebagai contoh, dalam fungsi register pelanggan, misalkan analis menulis bahwa departemen pendaftaran pelanggan diawaki oleh pegawai yang melapor untuk bekerja antara jam 8 pagi dan 5 sore, 7 hari seminggu. Informasi ini dapat disebut noise karena hampir tidak ada gunanya bagi developer perangkat lunak dan akan mengacaukan dokumen SRS yang tidak perlu, mengalihkan perhatian dari poin-poin penting.

Beberapa “sins” lain dari dokumen SRS dapat didaftar dan digunakan untuk mencegah penulisan dokumen SRS yang buruk dan juga digunakan sebagai checklist untuk meninjau dokumen SRS.

Kategori Penting Persyaratan Pelanggan

Dokumen SRS yang baik, harus dengan benar mengkategorikan dan mengatur persyaratan ke dalam bagian yang berbeda [IEEE830]. Sesuai dengan pedoman IEEE 830, kategori penting dari kebutuhan pengguna adalah sebagai berikut. Dokumen SRS harus dengan jelas mendokumentasikan aspek-aspek perangkat lunak berikut ini:

- Persyaratan fungsional
- Persyaratan non-fungsional
  - Kendala desain dan implementasi
  - Diperlukan antarmuka eksternal
  - Persyaratan non-fungsional lainnya
- Tujuan implementasi.

Persyaratan fungsional

Persyaratan fungsional menangkap fungsionalitas yang dibutuhkan oleh pengguna dari sistem. Mempertimbangkan perangkat lunak sama halnya dengan menawarkan satu set fungsi {fi} kepada pengguna. Fungsi-fungsi ini dapat dianggap mirip dengan fungsi matematika f : I → O, artinya suatu fungsi mengubah elemen (ii) dalam domain input (I) menjadi nilai (oi) pada output (O). Tampilan fungsional dari suatu sistem ditunjukkan secara skematis pada Gambar 4.1. Setiap fungsi fi dari sistem dapat dianggap membaca data tertentu ii, dan kemudian mentransformasikan satu set data input (ii) ke set data output yang sesuai (oi). Persyaratan fungsional sistem, harus dengan jelas menggambarkan setiap fungsi yang akan didukung sistem bersama dengan kumpulan data input dan output yang sesuai.

Persyaratan non-fungsional

Persyaratan non-fungsional adalah kewajiban yang tidak dapat dinegosiasikan yang harus didukung oleh perangkat lunak. Persyaratan non-fungsional menangkap persyaratan pelanggan yang tidak dapat dinyatakan sebagai fungsi (yaitu, menerima data input dan menghasilkan data output). Persyaratan non-fungsional biasanya membahas aspek mengenai antarmuka eksternal, antarmuka pengguna, pemeliharaan, portabilitas, kegunaan, jumlah maksimum pengguna bersamaan, waktu, dan throughput (transaksi per detik, dll.). Persyaratan non-fungsional dapat menjadi kritis dalam arti bahwa setiap kegagalan oleh perangkat lunak yang dikembangkan untuk mencapai beberapa tingkat minimum yang ditentukan dalam persyaratan ini dapat dianggap sebagai kegagalan dan membuat perangkat

lunak tidak dapat diterima oleh pelanggan. Standar IEEE 830 merekomendasikan bahwa dari berbagai persyaratan non-fungsional, antarmuka eksternal, dan batasan desain dan implementasi harus didokumentasikan dalam dua bagian yang berbeda. Persyaratan non-fungsional yang tersisa harus didokumentasikan kemudian di bagian dan ini harus mencakup persyaratan kinerja dan keamanan. Berikut merupakan berbagai kategori persyaratan nonfungsional yang dijelaskan dalam tiga bagian berbeda:

Kendala desain dan implementasi: Kendala desain dan implementasi adalah kategori penting dari persyaratan non-fungsional yang menjelaskan item atau masalah apa pun yang akan membatasi opsi yang tersedia bagi pengembang. Beberapa contoh kendala dapat berupa—kebijakan perusahaan atau peraturan yang perlu dihormati; keterbatasan perangkat keras; antarmuka dengan aplikasi lain; teknologi, alat, dan basis data khusus yang akan digunakan; protokol komunikasi khusus yang akan digunakan; pertimbangan keamanan; konvensi desain atau standar pemrograman yang harus diikuti, dll. Pertimbangkan contoh kendala yang dapat dimasukkan dalam bagian ini—Oracle DBMS perlu digunakan karena ini akan memfasilitasi antarmuka yang mudah dengan aplikasi lain yang sudah beroperasi dalam organisasi.

Diperlukan antarmuka eksternal: Contoh antarmuka eksternal adalah— perangkat keras, perangkat lunak dan antarmuka komunikasi, antarmuka pengguna, format laporan, dll. Untuk menentukan antarmuka pengguna, setiap antarmuka antara perangkat lunak dan pengguna harus dijelaskan. Deskripsi dapat mencakup contoh gambar layar, standar GUI atau panduan gaya apa pun yang harus diikuti, batasan tata letak layar, tombol dan fungsi standar (misalnya, bantuan) yang akan muncul di setiap layar, pintasan keyboard, standar tampilan pesan kesalahan, dan sebagainya. pada. Salah satu contoh persyaratan antarmuka pengguna dari suatu perangkat lunak dapat berupa perangkat lunak itu harus dapat digunakan oleh pekerja lantai pabrik yang bahkan mungkin tidak memiliki gelar sekolah menengah. Detail desain antarmuka pengguna seperti desain layar, struktur menu, diagram navigasi, dll. harus didokumentasikan dalam dokumen spesifikasi antarmuka pengguna yang terpisah.

Persyaratan non-fungsional lainnya: Bagian ini berisi deskripsi persyaratan non-fungsional yang bukan merupakan batasan desain dan juga bukan persyaratan antarmuka eksternal. Contoh penting adalah persyaratan kinerja seperti jumlah transaksi yang diselesaikan per unit waktu. Selain persyaratan kinerja, persyaratan non-fungsional lainnya yang akan dijelaskan di bagian ini mungkin termasuk masalah keandalan, keakuratan hasil, dan masalah keamanan.

Tujuan implementasi

Bagian 'tujuan implementasi' dari dokumen SRS menawarkan beberapa saran umum mengenai perangkat lunak yang akan dikembangkan. Ini tidak mengikat pengembang, dan mereka mungkin mempertimbangkan saran ini jika memungkinkan. Misalnya, developer dapat menggunakan saran ini saat memilih di antara solusi desain yang berbeda. Sebuah tujuan, berbeda dengan persyaratan fungsional dan non-fungsional, tidak diperiksa oleh pelanggan untuk kesesuaian pada saat pengujian penerimaan.

Tujuan dari bagian implementasi mungkin mendokumentasikan masalah seperti revisi yang lebih mudah pada fungsionalitas sistem yang mungkin diperlukan di masa mendatang, dukungan yang lebih mudah untuk perangkat baru yang akan didukung di masa mendatang, masalah penggunaan kembali, dll. Ini adalah item yang mungkin disimpan oleh developer dalam pikiran mereka selama pengembangan sehingga sistem yang dikembangkan dapat memenuhi beberapa aspek yang tidak diperlukan segera. Penting untuk diingat bahwa apa pun yang akan diuji oleh pengguna dan penerimaan sistem akan tergantung pada hasil tugas

ini, biasanya dianggap sebagai persyaratan yang harus dipenuhi oleh sistem dan bukan tujuan dan sebaliknya.

Bagaimana mengklasifikasikan berbagai jenis kebutuhan?

Kita harus jelas mengenai aspek persyaratan sistem yang harus didokumentasikan sebagai persyaratan fungsional, yang didokumentasikan sebagai persyaratan non-fungsional, dan yang didokumentasikan sebagai tujuan implementasi. Aspek yang dapat dinyatakan sebagai transformasi dari beberapa data input ke beberapa data output (yaitu, fungsi sistem) harus didokumentasikan sebagai persyaratan fungsional. Persyaratan lain yang kepatuhannya oleh sistem yang dikembangkan dapat diverifikasi dengan memeriksa sistem didokumentasikan sebagai persyaratan non-fungsional. Aspek yang kepatuhannya oleh sistem yang dikembangkan tidak perlu diverifikasi tetapi hanya dimasukkan sebagai saran kepada developer didokumentasikan sebagai tujuan implementasi.

Perbedaan antara persyaratan non-fungsional dan pedoman adalah sebagai berikut. Persyaratan non-fungsional akan diuji kepatuhannya, sebelum produk yang dikembangkan diterima oleh pelanggan sedangkan pedoman, di sisi lain, adalah permintaan pelanggan yang diinginkan untuk dilakukan, tetapi tidak akan diuji selama penerimaan produk. Persyaratan fungsional membentuk dasar untuk sebagian besar desain dan metodologi pengujian. Oleh karena itu, kecuali jika persyaratan fungsional diidentifikasi dan didokumentasikan dengan benar, aktivitas desain dan pengujian tidak dapat dilakukan dengan memuaskan.

Persyaratan Fungsional

Untuk mendokumentasikan persyaratan fungsional suatu sistem, pertama-tama perlu dipelajari untuk mengidentifikasi fungsi tingkat tinggi dari sistem dengan membaca dokumentasi informal dari persyaratan yang dikumpulkan. Fungsi tingkat tinggi akan dipecah menjadi subpersyaratan yang lebih kecil. Setiap fungsi tingkat tinggi adalah contoh penggunaan sistem (use case) oleh pengguna dalam beberapa cara. Fungsi tingkat tinggi adalah fungsi yang digunakan pengguna untuk menyelesaikan pekerjaan yang bermanfaat.

Namun, definisi di atas bukanlah definisi fungsi tingkat tinggi yang sangat akurat. Misalnya, seberapa bergunakah suatu pekerjaan yang harus dilakukan oleh sistem agar dapat disebut 'pekerjaan yang berguna'? Bisakah pencetakan laporan transaksi ATM saat penarikan uang dari ATM disebut pekerjaan yang bermanfaat? Pencetakan transaksi ATM tidak boleh dianggap sebagai persyaratan tingkat tinggi, karena pengguna tidak secara khusus meminta aktivitas ini. Tanda terima akan dicetak secara otomatis sebagai bagian dari fungsi penarikan uang. Biasanya, pengguna memanggil (meminta) layanan dari setiap persyaratan tingkat tinggi. Oleh karena itu dimungkinkan untuk memperlakukan tanda terima cetak sebagai bagian dari fungsi penarikan uang daripada memperlakukannya sebagai fungsi tingkat tinggi. Oleh karena itu diperlukan bahwa untuk beberapa fungsi tingkat tinggi, kita mungkin harus memperdebatkan apakah kita ingin mempertimbangkannya sebagai fungsi tingkat tinggi atau tidak. Namun, akan menjadi mungkin untuk mengidentifikasi sebagian besar fungsi tingkat tinggi tanpa banyak kesulitan setelah mempraktikkan solusi untuk beberapa masalah latihan.

Setiap persyaratan tingkat tinggi biasanya melibatkan penerimaan beberapa data dari pengguna melalui antarmuka pengguna, mengubahnya menjadi respons yang diperlukan, dan kemudian menampilkan respons sistem dalam format yang tepat. Misalnya, dalam perangkat lunak otomatisasi perpustakaan, persyaratan fungsional tingkat tinggi mungkin berupa buku pencarian. Fungsi ini melibatkan penerimaan nama buku atau kumpulan kata kunci dari pengguna, menjalankan algoritme pencocokan pada daftar buku, dan akhirnya mengeluarkan buku yang cocok. Respons sistem yang dihasilkan dapat dalam beberapa bentuk, misalnya, tampilan pada terminal, hasil cetak, beberapa data yang ditransfer ke sistem lain, dll. Namun, dalam kasus yang menurun, persyaratan tingkat tinggi mungkin tidak melibatkan input data

apa pun ke sistem atau produksi hasil yang dapat ditampilkan. Misalnya, mungkin melibatkan menyalakan lampu, atau memulai motor dalam aplikasi tertanam.

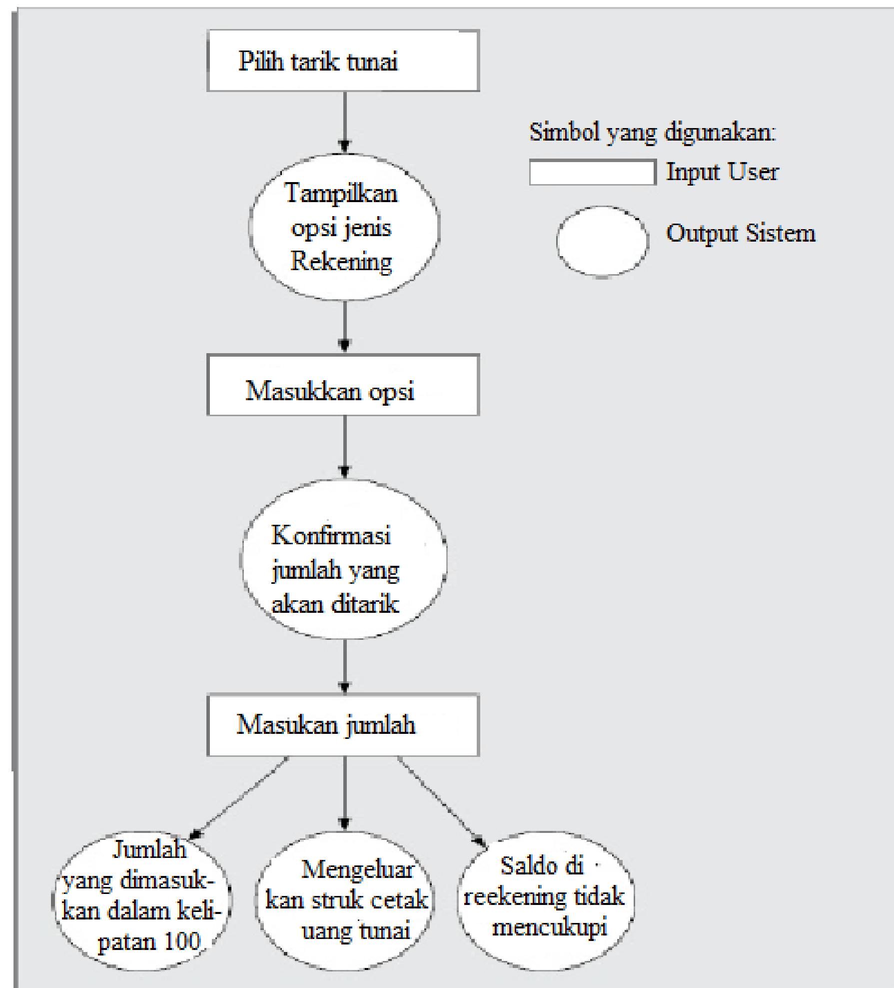
Namun, definisi di atas bukanlah definisi fungsi tingkat tinggi yang sangat akurat. Misalnya, seberapa bergunakah suatu pekerjaan yang harus dilakukan oleh sistem agar dapat disebut 'pekerjaan yang berguna'? Bisakah pencetakan laporan transaksi ATM saat penarikan uang dari ATM disebut pekerjaan yang bermanfaat? Pencetakan transaksi ATM tidak boleh dianggap sebagai persyaratan tingkat tinggi, karena pengguna tidak secara khusus meminta aktivitas ini. Tanda terima akan dicetak secara otomatis sebagai bagian dari fungsi penarikan uang. Biasanya, pengguna memanggil (meminta) layanan dari setiap persyaratan tingkat tinggi. Oleh karena itu dimungkinkan untuk memperlakukan tanda terima cetak sebagai bagian dari fungsi penarikan uang daripada memperlakukannya sebagai fungsi tingkat tinggi. Oleh karena itu diperlukan bahwa untuk beberapa fungsi tingkat tinggi, kita mungkin harus memperdebatkan apakah kita ingin mempertimbangkannya sebagai fungsi tingkat tinggi atau tidak. Namun, akan menjadi mungkin untuk mengidentifikasi sebagian besar fungsi tingkat tinggi tanpa banyak kesulitan setelah mempraktikkan solusi untuk beberapa masalah latihan.

Setiap persyaratan tingkat tinggi biasanya melibatkan penerimaan beberapa data dari pengguna melalui antarmuka pengguna, mengubahnya menjadi respons yang diperlukan, dan kemudian menampilkan respons sistem dalam format yang tepat. Misalnya, dalam perangkat lunak otomatisasi perpustakaan, persyaratan fungsional tingkat tinggi mungkin berupa buku pencarian. Fungsi ini melibatkan penerimaan nama buku atau kumpulan kata kunci dari pengguna, menjalankan algoritme pencocokan pada daftar buku, dan akhirnya mengeluarkan buku yang cocok. Respons sistem yang dihasilkan dapat dalam beberapa bentuk, misalnya, tampilan pada terminal, hasil cetak, beberapa data yang ditransfer ke sistem lain, dll. Namun, dalam kasus yang menurun, persyaratan tingkat tinggi mungkin tidak melibatkan input data apa pun ke sistem atau produksi hasil yang dapat ditampilkan. Misalnya, mungkin melibatkan menyalakan lampu, atau memulai motor dalam aplikasi tertanam.

Apakah fungsi tingkat tinggi dari suatu sistem mirip dengan fungsi matematika?

Kita semua tahu bahwa fungsi matematika mengubah data input menjadi data output. Fungsi tingkat tinggi mengubah data masukan tertentu menjadi data keluaran. Namun, kecuali untuk fungsi tingkat tinggi yang sangat sederhana, suatu fungsi jarang membaca semua data yang diperlukan sekaligus dan jarang mengeluarkan semua hasil dalam satu bidikan. Faktanya, fungsi tingkat tinggi biasanya melibatkan serangkaian interaksi antara sistem dan satu atau lebih pengguna. Contoh interaksi yang mungkin terjadi dalam persyaratan tingkat tinggi tunggal telah ditunjukkan pada Gambar 4.2. Pada Gambar 4.2, input pengguna telah diwakili oleh persegi panjang dan respon yang dihasilkan oleh sistem oleh lingkaran. Amati bahwa persegi panjang dan lingkaran bergantian dalam pelaksanaan fungsi tingkat tinggi tunggal sistem, menunjukkan serangkaian permintaan dari pengguna dan tanggapan yang sesuai dari sistem. Biasanya, ada beberapa input data awal oleh pengguna. Setelah menerima ini, sistem dapat menampilkan beberapa respons (disebut tindakan sistem). Berdasarkan ini, pengguna dapat memasukkan data lebih lanjut, dan seterusnya. Untuk fungsi tingkat tinggi apa pun, mungkin ada urutan atau skenario interaksi yang berbeda karena pengguna memilih opsi yang berbeda atau memasukkan item data yang berbeda.

Pada Gambar 4.2, skenario yang berbeda terjadi tergantung pada jumlah yang dimasukkan untuk penarikan. Skenario yang berbeda pada dasarnya adalah perilaku yang berbeda yang ditunjukkan oleh sistem untuk fungsi tingkat tinggi yang sama. Biasanya, setiap input pengguna dan tindakan sistem yang sesuai dapat dianggap sebagai sub-persyaratan dari persyaratan tingkat tinggi. Dengan demikian, setiap kebutuhan tingkat tinggi dapat terdiri dari beberapa sub-persyaratan.



Gambar 4.2 Interaksi pengguna dan sistem dalam kebutuhan fungsional tingkat tinggi.

Apakah mungkin untuk menentukan semua data input dan output dengan tepat?

Dalam dokumen spesifikasi kebutuhan, diinginkan untuk menentukan input data yang tepat ke sistem dan output data yang tepat oleh sistem. Terkadang, item data yang tepat mungkin sangat sulit untuk diidentifikasi. Hal ini terutama terjadi, ketika tidak ada model kerja dari sistem yang akan dikembangkan. Dalam kasus seperti itu, data dalam persyaratan tingkat tinggi harus dijelaskan menggunakan istilah tingkat tinggi dan mungkin sangat sulit untuk mengidentifikasi komponen yang tepat dari data ini secara akurat. Aspek lain yang harus diingat adalah bahwa data mungkin dimasukkan ke sistem secara bertahap pada titik yang berbeda dalam eksekusi. Sebagai contoh, perhatikan fungsi penarikan tunai dari mesin anjungan tunai mandiri (ATM) pada Gambar 4.2. Karena selama pelaksanaan fungsi penarikan tunai, pengguna harus memasukkan jenis akun, jumlah yang akan ditarik, sangat sulit untuk membentuk satu nama tingkat tinggi yang secara akurat menggambarkan kedua data input. Namun, data input untuk subfungsi dapat dijelaskan lebih akurat.

Bagaimana Mengidentifikasi Persyaratan Fungsional?

Persyaratan fungsional tingkat tinggi sering kali perlu diidentifikasi baik dari dokumen deskripsi masalah informal atau dari pemahaman konseptual masalah. Setiap persyaratan tingkat tinggi mencirikan cara penggunaan sistem (pemanggilan layanan) oleh beberapa pengguna untuk melakukan beberapa pekerjaan yang berarti. Ingatlah bahwa mungkin ada banyak jenis pengguna sistem dan persyaratan mereka dari sistem mungkin sangat berbeda. Jadi, seringkali berguna untuk terlebih dahulu mengidentifikasi berbagai jenis pengguna yang

mungkin menggunakan sistem dan kemudian mencoba mengidentifikasi berbagai layanan yang diharapkan dari perangkat lunak oleh berbagai jenis pengguna.

Keputusan mengenai fungsionalitas sistem mana yang dapat diambil sebagai persyaratan fungsional tingkat tinggi dan yang dapat dianggap sebagai bagian dari fungsi lain (yaitu, subfungsi) meninggalkan ruang lingkup untuk beberapa subjektivitas. Misalnya, pertimbangkan fungsi buku terbitan di Sistem Otomasi Perpustakaan. Misalkan, ketika pengguna memanggil fungsi buku terbitan, sistem akan meminta pengguna memasukkan detail setiap buku yang akan diterbitkan. Haruskah entri detail buku dianggap sebagai fungsi tingkat tinggi, atau hanya sebagai bagian dari fungsi penerbitan-buku? Banyak kali, pilihannya jelas. Tapi, terkadang itu membutuhkan pengambilan keputusan yang tidak sepele.

Bagaimana Mendokumentasikan Persyaratan Fungsional?

Setelah semua persyaratan fungsional tingkat tinggi telah diidentifikasi dan masalah persyaratan telah dihilangkan, ini didokumentasikan. Suatu fungsi dapat didokumentasikan dengan mengidentifikasi keadaan di mana data akan dimasukkan ke sistem, domain data inputnya, domain data output, dan jenis pemrosesan yang akan dilakukan pada data input untuk mendapatkan data output. Sekarang mari kita lihat spesifikasi persyaratan fungsional melalui dua contoh. Mari kita coba mendokumentasikan fungsi tarik tunai dari sistem anjungan tunai mandiri (ATM) berikut ini. Penarikan tunai adalah persyaratan tingkat tinggi. Ini memiliki beberapa sub-persyaratan yang sesuai dengan interaksi pengguna yang berbeda. Urutan interaksi pengguna ini dapat bervariasi dari satu pemanggilan ke pemanggilan lainnya tergantung pada beberapa kondisi. Urutan interaksi yang berbeda ini menangkap skenario yang berbeda. Untuk secara akurat menggambarkan persyaratan fungsional, kita harus mendokumentasikan semua skenario berbeda yang mungkin terjadi.

Contoh 4.7 (Tarik tunai dari ATM): Deskripsi informal awal tentang fungsionalitas yang diperlukan biasanya diberikan oleh pelanggan sebagai pernyataan tujuan (SoP). SoP berfungsi sebagai titik awal bagi analis dan dia melanjutkan dengan aktivitas pengumpulan persyaratan setelah pemahaman dasar tentang SoP. Namun, fungsi tarik tunai dari ATM secara intuitif jelas bagi siapa saja yang telah menggunakan ATM bank. Jadi, kita tidak perlu menyertakan deskripsi informal tentang fungsi penarikan tunai di sini dan berikut ini, kita hanya mendokumentasikan persyaratan fungsional ini.

R.1: Tarik tunai

Deskripsi:Fungsi penarikan tunai pertama-tama menentukan jenis rekening yang dimiliki pengguna dan nomor rekening tempat pengguna ingin menarik uang tunai. Ini memeriksa saldo untuk menentukan apakah jumlah yang diminta tersedia di akun. Jika saldo cukup tersedia, ia mengeluarkan uang tunai yang diperlukan, jika tidak maka akan menghasilkan pesan kesalahan.

R.1.1: Pilih opsi jumlah penarikan

Input: Opsi "Tarik jumlah" dipilih Output: Pengguna diminta untuk memasukkan jenis akun

R.1.2: Pilih jenis akun

I***n***put : Pengguna memilih opsi dari salah satu dari berikut ini—tabungan/giro/deposito.

Output: Prompt untuk memasukkan jumlah

R.1.3: Dapatkan jumlah yang dibutuhkan

Input: Jumlah yang akan ditarik dalam nilai bilangan bulat lebih besar dari 100 dan kurang dari 10.000 dalam kelipatan 100.

Output: Kas yang diminta dan laporan transaksi tercetak.

Pemrosesan: Jumlah didebit dari rekening pengguna jika saldo yang cukup tersedia, jika tidak, pesan kesalahan akan ditampilkan.

Contoh 4.8 (Pencarian ketersediaan buku di perpustakaan): Deskripsi informal awal dari fungsionalitas yang diperlukan biasanya diberikan oleh pelanggan sebagai pernyataan tujuan (SoP) berdasarkan pengumpulan persyaratan selanjutnya, analis memahami fungsionalitas tersebut. Namun, fungsionalitas ketersediaan buku pencarian secara intuitif jelas bagi siapa saja yang telah menggunakan perpustakaan.

R.1: Deskripsi Buku Pencarian

Setelah pengguna memilih opsi pencarian, dia akan diminta untuk memasukkan kata kunci. Sistem akan mencari buku dalam daftar buku berdasarkan kata kunci yang dimasukkan. Setelah melakukan pencarian, sistem akan menampilkan rincian semua buku yang judul atau nama pengarangnya cocok dengan kata kunci yang dimasukkan. Rincian buku yang akan ditampilkan meliputi: judul, nama pengarang, nama penerbit, tahun terbit, nomor ISBN, nomor katalog, dan lokasi di perpustakaan.

R.1.1: Pilih opsi pencarian

Input: Opsi “Cari”

Output: Pengguna diminta untuk memasukkan kata kunci

R.1.2: Cari dan tampilkan

Inpur: Kata kunci

Output: Rincian semua buku yang judul atau nama pengarangnya cocok dengan kata kunci yang dimasukkan oleh pengguna. Rincian buku yang ditampilkan meliputi judul buku, nama pengarang, nomor ISBN, nomor katalog, tahun terbit, jumlah eksemplar yang tersedia, dan lokasi perpustakaan.

Pengolahan: Cari daftar buku berdasarkan kata kunci:

R.2: Perbarui buku

Deskripsi: Ketika opsi “perbarui” dipilih, pengguna diminta untuk memasukkan nomor keanggotaan dan kata sandinya. Setelah validasi kata sandi, daftar buku yang dipinjamnya akan ditampilkan. Pengguna dapat memperbarui buku yang dipinjamnya dengan menunjukkannya. Buku yang diminta tidak dapat diperpanjang jika dipesan oleh pengguna lain. Dalam hal ini, pesan kesalahan akan ditampilkan.

R.2.1: Pilih opsi perpanjangan

Status: Pengguna telah login dan menu utama telah ditampilkan.

Input: pemilihan opsi "Perbarui".

Output: Pesan prompt kepada pengguna untuk memasukkan nomor keanggotaan dan kata sandinya.

R.2.2: Masuk

Status: Opsi perpanjangan telah dipilih.

Input: Nomor keanggotaan dan kata sandi.

Output: Daftar buku yang dipinjam oleh pengguna ditampilkan, dan pengguna diminta untuk memilih buku yang akan diperbarui, jika kata sandinya valid. Jika kata sandi tidak valid, pengguna diminta untuk memasukkan kembali kata sandi.

Pemrosesan: Validasi kata sandi, cari buku yang diterbitkan untuk pengguna dari daftar peminjam dan tampilkan.

Fungsi selanjutnya: R.2.3 jika kata sandi valid dan R.2.2 jika kata sandi tidak valid.

R.2.3: Perbarui buku yang dipilih

Input: Pilihan pengguna untuk buku-buku yang akan diperbaharui dari buku-buku yang dipinjamnya.

Output: Konfirmasi buku-buku yang berhasil diperbarui dan pesan permintaan maaf untuk buku-buku yang tidak dapat diperbarui.

Pemrosesan: Periksa apakah ada yang memesan buku yang diminta. Perbarui buku-buku yang dipilih oleh pengguna dalam daftar peminjam, jika tidak ada yang memesan buku-buku itu.

Untuk mengidentifikasi persyaratan tingkat tinggi dengan benar, banyak akal sehat dan kemampuan untuk memvisualisasikan berbagai skenario yang mungkin muncul dalam pengoperasian suatu fungsi diperlukan. Harap dicatat bahwa ketika salah satu aspek persyaratan, seperti status, deskripsi pemrosesan, fungsi berikutnya yang akan dieksekusi, dll terlihat jelas. Kita harus membuat *trade-off* antara mengacaukan dokumen dengan detail sepele versus kehilangan beberapa deskripsi penting.

Spesifikasi perangkat lunak besar: Jika ada sejumlah besar persyaratan fungsional (jauh lebih besar daripada yang terlihat), haruskah mereka hanya ditulis dalam daftar persyaratan bernomor panjang? Cara yang lebih baik untuk mengatur persyaratan fungsional dalam hal ini adalah dengan membagi persyaratan menjadi beberapa bagian dari persyaratan terkait. Misalnya, persyaratan fungsional perangkat lunak otomasi lembaga akademik dapat dibagi menjadi beberapa bagian seperti akun, akademisi, inventaris, publikasi, dll. Jika ada terlalu banyak persyaratan fungsional, ini harus diatur dengan benar menjadi beberapa bagian. Misalnya, berikut ini dapat menjadi bagian dalam perangkat lunak otomatisasi rumah perdagangan:

- Manajemen pelanggan
- Manajemen akun
- Manajemen pembelian
- Manajemen vendor
- Manajemen persediaan

Tingkat detail dalam spesifikasi: Bahkan untuk analis berpengalaman, dilema umum adalah dalam menentukan terlalu sedikit atau menentukan terlalu banyak. Dalam praktiknya, kita harus menentukan hanya interaksi input/output yang penting dalam suatu fungsionalitas bersama dengan pemrosesan yang diperlukan untuk menghasilkan output dari input. Namun, jika urutan interaksi ditentukan terlalu detail, maka itu menjadi kendala yang tidak perlu pada developer dan membatasi pilihan mereka dalam solusi. Di sisi lain, jika urutan interaksi tidak cukup rinci, dapat menyebabkan ambiguitas dan mengakibatkan implementasi yang tidak tepat.

Ketertelusuran

Ketertelusuran berarti memungkinkan untuk mengidentifikasi (menelusuri) komponen desain khusus yang mengimplementasikan persyaratan tertentu, bagian kode yang sesuai dengan komponen desain tertentu, dan kasus uji yang menguji persyaratan tertentu. Dengan demikian, setiap komponen kode yang diberikan dapat dilacak ke komponen desain yang sesuai, dan komponen desain dapat dilacak ke persyaratan spesifik yang diterapkannya dan sebaliknya. Analisis ketertelusuran adalah konsep penting dan sering digunakan selama pengembangan perangkat lunak. Misalnya, dengan melakukan analisis ketertelusuran, kita dapat mengetahui apakah semua persyaratan telah ditangani secara memuaskan di semua fase. Ini juga dapat digunakan untuk menilai dampak dari perubahan persyaratan. Artinya, ketertelusuran memudahkan untuk mengidentifikasi bagian mana dari desain dan kode yang akan terpengaruh, ketika terjadi perubahan persyaratan tertentu. Ini juga dapat digunakan untuk mempelajari dampak bug yang diketahui ada di bagian kode pada berbagai persyaratan, dll.

Untuk mencapai ketertelusuran, setiap kebutuhan fungsional harus diberi nomor secara unik dan konsisten. Penomoran persyaratan yang tepat memungkinkan dokumen yang berbeda untuk secara unik merujuk pada persyaratan tertentu. Contoh skema penomoran kebutuhan fungsional ditunjukkan pada Contoh 4.7 dan 4.8, di mana kebutuhan fungsional diberi nomor R.1, R.2, dst. dan subpersyaratan untuk kebutuhan R.1 diberi nomor R.1.1, R .1.2, dll.

Organisasi Dokumen SRS

Pada bagian ini, kita membahas organisasi dokumen SRS seperti yang ditentukan oleh standar IEEE 830 [IEEE 830]. Harap dicatat bahwa standar IEEE 830 dimaksudkan hanya sebagai pedoman untuk mengatur dokumen spesifikasi persyaratan menjadi beberapa bagian dan memungkinkan fleksibilitas untuk menyesuaikannya, seperti yang mungkin diperlukan untuk proyek tertentu. Tergantung pada jenis proyek yang sedang ditangani, beberapa bagian dapat dihilangkan, diperkenalkan, atau dipertukarkan yang mungkin dianggap bijaksana oleh analis. Namun, pengorganisasian dokumen SRS sebagian besar tergantung pada preferensi analis sistem itu sendiri, dan dalam hal ini ia sering dipandu oleh kebijakan dan standar yang diikuti oleh perusahaan pengembang. Juga, organisasi dokumen dan isu-isu yang dibahas di dalamnya sebagian besar tergantung pada jenis produk yang dikembangkan. Namun, terlepas dari prinsip dan jenis produk perusahaan, tiga isu dasar yang harus didiskusikan oleh dokumen SRS adalah—persyaratan fungsional, persyaratan non-fungsional, dan pedoman untuk implementasi sistem.

Bagian pendahuluan harus menjelaskan konteks di mana sistem sedang dikembangkan, dan memberikan gambaran keseluruhan sistem, dan karakteristik lingkungan. Bagian pengenalan dapat mencakup perangkat keras yang akan menjalankan sistem, perangkat yang akan berinteraksi dengan sistem, dan tingkat keterampilan pengguna. Deskripsi tingkat keterampilan pengguna adalah penting, karena desain bahasa perintah dan gaya presentasi dari berbagai dokumen sangat bergantung pada jenis pengguna yang ditargetkan. Misalnya, jika tingkat keterampilan pengguna adalah "pemula", itu berarti antarmuka pengguna harus sangat sederhana dan kokoh, sedangkan jika tingkat pengguna "maju", beberapa teknik pintas dan fitur lanjutan mungkin disediakan di antarmuka pengguna.

Diinginkan untuk menggambarkan format untuk perintah input, data input, laporan output, dan jika perlu mode interaksi. Kita sudah memppelajari bagaimana isi Bagian tentang persyaratan fungsional, persyaratan non-fungsional, dan tujuan implementasi harus ditulis.

Pengantar

Tujuan: Bagian ini harus menjelaskan di mana perangkat lunak akan digunakan dan bagaimana perangkat lunak akan digunakan.

Lingkup proyek: Bagian ini harus menjelaskan secara singkat konteks keseluruhan di mana perangkat lunak sedang dikembangkan. Misalnya, bagian dari masalah yang sedang diotomatisasi dan bagian yang perlu diotomatisasi selama evolusi perangkat lunak di masa mendatang.

Karakteristik lingkungan: Bagian ini harus secara singkat menguraikan lingkungan (perangkat keras dan perangkat lunak lain) dengan mana perangkat lunak akan berinteraksi.

Deskripsi keseluruhan organisasi dokumen SRS

Perspektif produk: Bagian ini perlu menyatakan secara singkat apakah perangkat lunak tersebut dimaksudkan untuk menggantikan sistem tertentu yang sudah ada, atau merupakan perangkat lunak baru. Jika perangkat lunak yang dikembangkan akan digunakan sebagai komponen dari sistem yang lebih besar, diagram skematik sederhana dapat diberikan untuk menunjukkan komponen utama dari sistem secara keseluruhan, interkoneksi subsistem, dan antarmuka eksternal dapat membantu.

Fitur produk: Bagian ini harus merangkum cara utama di mana perangkat lunak akan digunakan. Rincian harus diberikan dalam Bagian 3 dokumen. Jadi, hanya ringkasan singkat yang harus disajikan di sini.

Kelas pengguna: Berbagai kelas pengguna yang diharapkan menggunakan perangkat lunak ini diidentifikasi dan dijelaskan di sini. Kelas pengguna yang berbeda diidentifikasi oleh jenis fungsi yang diharapkan untuk mereka gunakan, atau tingkat keahlian mereka dalam menggunakan komputer.

Lingkungan operasi: Bagian ini harus membahas secara rinci platform perangkat keras di mana perangkat lunak akan berjalan, sistem operasi, dan perangkat lunak aplikasi lain yang akan berinteraksi dengan perangkat lunak yang dikembangkan.

Kendala desain dan implementasi: Pada bagian ini, batasan yang berbeda pada desain dan implementasi dibahas. Ini mungkin termasuk—kebijakan perusahaan atau peraturan; keterbatasan perangkat keras (persyaratan waktu, persyaratan memori); antarmuka ke aplikasi lain; teknologi, alat, dan basis data khusus yang akan digunakan; bahasa pemrograman tertentu yang akan digunakan; protokol komunikasi khusus yang akan digunakan; pertimbangan keamanan; konvensi desain atau standar pemrograman.

Dokumentasi pengguna: Bagian ini harus mencantumkan jenis dokumentasi pengguna, seperti manual pengguna, bantuan online, dan manual pemecahan masalah yang akan dikirimkan ke pelanggan bersama dengan perangkat lunak.

Persyaratan fungsional untuk organisasi dokumen SRS

Bagian ini dapat mengklasifikasikan fungsionalitas baik berdasarkan fungsionalitas spesifik yang dipanggil oleh pengguna yang berbeda, atau fungsionalitas yang tersedia dalam mode yang berbeda, dll., tergantung apa yang mungkin sesuai.

1. Kelas pengguna 1
   (a) Persyaratan fungsional 1.1
   (b) Persyaratan fungsional 1.2
2. Kelas pengguna 2
   (a) Persyaratan fungsional 2.1
   (b) Persyaratan fungsional 2.2

Persyaratan antarmuka eksternal

Antarmuka pengguna: Bagian ini harus menjelaskan deskripsi tingkat tinggi dari berbagai antarmuka dan berbagai prinsip yang harus diikuti. Deskripsi antarmuka pengguna dapat mencakup gambar layar sampel, standar GUI atau panduan gaya apa pun yang harus diikuti, batasan tata letak layar, tombol tekan standar (misalnya, bantuan) yang akan muncul di setiap layar, pintasan keyboard, standar tampilan pesan kesalahan, dll. Rincian desain antarmuka pengguna harus didokumentasikan dalam dokumen spesifikasi antarmuka pengguna yang terpisah.

Antarmuka perangkat keras: Bagian ini harus menjelaskan antarmuka antara perangkat lunak dan komponen perangkat keras dari sistem. Bagian ini dapat mencakup deskripsi jenis perangkat yang didukung, sifat data dan interaksi kontrol antara perangkat lunak dan perangkat keras, dan protokol komunikasi yang akan digunakan.

Antarmuka perangkat lunak: Bagian ini harus menjelaskan hubungan antara perangkat lunak ini dan komponen perangkat lunak spesifik lainnya, termasuk database, sistem operasi, alat, perpustakaan, dan komponen komersial terintegrasi, dll. Identifikasi item data yang akan dimasukkan ke perangkat lunak dan data yang yang akan dihasilkan harus diidentifikasi dan tujuan masing-masing harus dijelaskan.

Antarmuka komunikasi: Bagian ini harus menjelaskan persyaratan yang terkait dengan semua jenis komunikasi yang diperlukan oleh perangkat lunak, seperti email, akses

web, protokol komunikasi server jaringan, dll. Bagian ini harus menentukan format pesan terkait yang akan digunakan. Itu juga harus mengidentifikasi standar komunikasi apa pun yang akan digunakan, seperti soket TCP, FTP, HTTP, atau SHTTP. Tentukan masalah keamanan komunikasi atau enkripsi apa pun yang mungkin relevan, dan juga kecepatan transfer data, dan mekanisme sinkronisasi.

Persyaratan non-fungsional lainnya untuk organisasi dokumen SRS

Selain kendala desain, implementasi dan persyaratan antarmuka eksternal, bagian ini akanmenjelaskan persyaratan non-fungsional.

Persyaratan kinerja: Aspek seperti jumlah transaksi yang harus diselesaikan per detik harus ditentukan di sini. Beberapa persyaratan kinerja mungkin khusus untuk persyaratan atau fitur fungsional individu. Ini juga harus ditentukan di sini.

Persyaratan keamanan: Persyaratan yang berkaitan dengan kemungkinan kehilangan atau kerusakan yang diakibatkan oleh penggunaan perangkat lunak ditentukan di sini. Misalnya, pemulihan setelah kegagalan daya, penanganan kegagalan perangkat lunak dan perangkat keras, dll. dapat didokumentasikan di sini.

Persyaratan keamanan: Bagian ini harus menetapkan persyaratan apa pun terkait persyaratan keamanan atau privasi pada data yang digunakan atau dibuat oleh perangkat lunak. Persyaratan otentikasi identitas pengguna harus dijelaskan di sini. Ini juga harus mengacu pada kebijakan atau peraturan eksternal tentang masalah keamanan. Tentukan sertifikasi keamanan atau privasi yang harus dipenuhi.

Untuk perangkat lunak yang memiliki mode operasi yang berbeda, di bagian persyaratan fungsional, mode operasi yang berbeda dapat dicantumkan dan di setiap mode fungsi spesifik yang tersedia untuk pemanggilan dapat diatur sebagai berikut.

Persyaratan fungsional

1.1. Modus operasi 1
  - (a) Persyaratan fungsional 1.1
  - (b) Persyaratan fungsional 1.2

2. Modus operasi 2
  - (a) Persyaratan fungsional 2.1
  - (b) Persyaratan fungsional 2.2

Spesifikasi perilaku mungkin tidak diperlukan untuk semua sistem. Biasanya diperlukan untuk sistem-sistem di mana perilaku sistem tergantung pada keadaan di mana sistem itu berada, dan sistem transit di antara sekumpulan keadaan tergantung pada beberapa kondisi dan kejadian yang telah ditentukan sebelumnya. Perilaku suatu sistem dapat ditentukan menggunakan *formalisme finite state machine* (FSM) dan formalisme alternatif lainnya. FSM dapat digunakan untuk menentukan kemungkinan status (mode) dari sistem dan transisi di antara status ini karena terjadinya peristiwa.

Contoh 4.9 (Perangkat lunak perpustakaan pribadi): Diusulkan untuk mengembangkan perangkat lunak yang akan digunakan oleh individu untuk mengelola koleksi buku pribadi mereka. Berikut ini adalah deskripsi informal tentang persyaratan perangkat lunak ini seperti yang dikerjakan oleh departemen pemasaran. Mengembangkan persyaratan fungsional dan non-fungsional untuk perangkat lunak.

Seseorang dapat memiliki hingga beberapa ratus buku. Rincian semua buku seperti nama buku, tahun penerbitan, tanggal pembelian, harga, dan penerbit akan dimasukkan oleh pemiliknya. Sebuah buku harus diberi nomor seri unik oleh komputer. Nomor ini akan ditulis oleh pemilik menggunakan pena di halaman dalam buku. Hanya teman terdaftar yang dapat meminjamkan buku. Saat mendaftarkan teman, data berikut harus disediakan—nama teman, alamatnya, nomor telepon rumah, dan nomor ponsel. Setiap kali permintaan penerbitan buku

diberikan, nama teman yang akan diterbitkan bukunya dan id unik bukunya dimasukkan. Pada saat ini, berbagai buku yang beredar terhadap peminjam bersama dengan tanggal pinjaman ditampilkan untuk informasi pemilik. Jika pemilik ingin melanjutkan penerbitan buku, maka tanggal penerbitan, judul buku, dan nomor identifikasi unik buku disimpan. Ketika seorang teman mengembalikan sebuah buku, tanggal pengembalian disimpan dan buku tersebut dihapus dari daftar peminjamannya. Berdasarkan permintaan, perangkat lunak harus menampilkan nama, alamat, dan nomor telepon dari setiap teman yang bukunya beredar bersama dengan judul buku yang beredar dan tanggal penerbitannya. Perangkat lunak harus memungkinkan pemilik untuk memperbarui rincian teman seperti alamat, telepon, nomor telepon, dll. Pemilik harus dapat menghapus semua data yang berkaitan dengan teman yang tidak lagi aktif menggunakan perpustakaan. Catatan harus disimpan menggunakan sistem manajemen basis data gratis (domain publik). Perangkat lunak harus berjalan pada mesin Windows dan Unix.

Setiap kali pemilik perangkat lunak perpustakaan meminjam buku dari teman-temannya, akan memasukkan rincian mengenai judul buku, tanggal pinjaman dan teman dari siapa dia meminjamnya. Demikian pula, rincian pengembalian buku akan dimasukkan. Perangkat lunak harus dapat menampilkan semua buku yang dipinjam dari berbagai teman atas permintaan pemiliknya. Seharusnya dimungkinkan bagi siapa pun untuk menanyakan tentang ketersediaan buku tertentu melalui browser web dari lokasi mana pun. Pemilik harus dapat menanyakan jumlah total buku di perpustakaan pribadi, dan jumlah total yang dia investasikan di perpustakaannya. Ia juga harus dapat melihat jumlah buku yang dipinjam dan dikembalikan oleh (atau semua) teman selama waktu tertentu.

Persyaratan fungsional

Perangkat lunak perlu mendukung tiga kategori fungsi seperti yang dijelaskan di bawah ini:

1. ***Kelola buku sendiri***

1.1. Daftar buku

Deskripsi: Untuk mendaftarkan buku di perpustakaan pribadi, rincian buku, seperti nama, tahun penerbitan, tanggal pembelian, harga dan penerbit dimasukkan. Ini disimpan dalam database dan nomor seri unik dihasilkan.

Masukan: Detail buku

Output: Nomor seri unik

R.1.2: Edisi buku

Deskripsi: Seseorang hanya dapat diberikan buku jika dia terdaftar. Berbagai buku yang beredar melawannya bersama dengan tanggal peminjaman pertama kali ditampilkan.

R.1.2.1: Menampilkan buku-buku yang luar biasa

Deskripsi: Dimasukkan terlebih dahulu nama teman dan nomor seri buku yang akan diterbitkan. Maka buku-buku yang beredar melawan teman harus ditampilkan.

Masukan: Nama teman

Keluaran: Daftar buku yang beredar beserta tanggal peminjamannya.

R.1.2.2: Konfirmasi buku terbitan

Jika pemiliknya mengkonfirmasi, maka buku itu harus dikeluarkan untuknya dan catatan yang relevan harus diperbarui.

Input: Konfirmasi pemilik untuk penerbitan buku. Output: Konfirmasi penerbitan buku.

R.1.3: Kueri buku-buku yang luar biasa

Deskripsi: Rincian teman-teman yang memiliki buku luar biasa dengan nama mereka ditampilkan.

Masukan: Pilihan pengguna
Keluaran: Tampilan termasuk nama, alamat dan nomor telepon dari setiap teman yang bukunya beredar beserta judul bukunya yang beredar dan tanggal penerbitannya.
R.1.4: Buku pertanyaan
Deskripsi: Setiap pengguna harus dapat menanyakan buku tertentu dari mana saja menggunakan browser web.
Masukan: Nama buku.
Output: Ketersediaan buku dan apakah buku tersebut diterbitkan.
R.1.5: Kembalikan buku
Deskripsi: Pada saat pengembalian buku oleh teman, tanggal pengembalian disimpan dan buku dihapus dari daftar peminjaman teman yang bersangkutan.
Masukan: Nama buku.
Keluaran: Pesan konfirmasi.

2. ***Kelola detail teman***

R.2.1: Daftarkan teman
Deskripsi: Seorang teman harus terdaftar sebelum dia dapat menerbitkan buku. Setelah data pendaftaran dimasukkan dengan benar, data tersebut harus disimpan dan pesan konfirmasi akan ditampilkan.
Masukan: Rincian teman termasuk nama teman, alamat, nomor telepon rumah dan nomor ponsel.
Output: Konfirmasi status pendaftaran.
R.2.2: Perbarui detail teman
Deskripsi: Ketika informasi pendaftaran teman berubah, hal yang sama harus diperbarui di komputer.
R.2.2.1: Menampilkan detail terkini
Masukan: Nama teman.
Output: Detail yang disimpan saat ini.
R2.2.2: Perbarui detail teman
Masukan: Perubahan diperlukan.
Output: Detail yang diperbarui dengan konfirmasi perubahan.
R.3.3: Menghapus catatan teman
Deskripsi: Menghapus catatan anggota yang tidak aktif.
Masukan: Nama teman.
Keluaran: Pesan konfirmasi.

3. ***Kelola buku yang dipinjam***

R.3.1: Mendaftarkan buku yang dipinjam
Deskripsi: Buku-buku yang dipinjam oleh pengguna perpustakaan pribadi terdaftar.
Input: Judul buku dan tanggal peminjaman.
Output: Konfirmasi status pendaftaran.
R.3.2: Deregister buku yang dipinjam
Deskripsi: Buku yang dipinjam dibatalkan pendaftarannya ketika dikembalikan.
Masukan: Nama buku.
Keluaran: Konfirmasi pembatalan pendaftaran.
R.3.3: Menampilkan buku yang dipinjam
Deskripsi: Data tentang buku yang dipinjam oleh pemilik ditampilkan.
Masukan: Pilihan pengguna.
Output: Daftar buku yang dipinjam dari teman lain.

4. ***Kelola statistik***

R.4.1: Menampilkan jumlah buku
Deskripsi: Jumlah total buku di perpustakaan pribadi harus ditampilkan.
Masukan: Pilihan pengguna.
Output: Hitungan buku.
R4.2: Tampilkan jumlah yang diinvestasikan
Deskripsi: Jumlah total yang diinvestasikan di perpustakaan pribadi ditampilkan.
Masukan: Pilihan pengguna.
Output: Jumlah total yang diinvestasikan.
R.4.2: Menampilkan jumlah transaksi
Deskripsi: Jumlah total buku yang diterbitkan dan dikembalikan selama periode tertentu oleh satu (atau semua) teman ditampilkan.
Input: Awal periode dan akhir periode.
Output: Jumlah buku yang diterbitkan dan jumlah buku yang dikembalikan.

Persyaratan non-fungsional

N.1: Basis Data: Sistem manajemen basis data yang tersedia gratis di domain publik harus digunakan.

N.2: Platform: Perangkat lunak versi Windows dan Unix perlu dikembangkan. N.3: Dukungan web: Seharusnya dimungkinkan untuk memanggil fungsionalitas buku kueri dari tempat mana pun dengan menggunakan browser web.

Pengamatan: Karena ada banyak persyaratan fungsional, persyaratan telah diatur menjadi empat bagian: Mengelola buku sendiri, mengelola teman, mengelola buku yang dipinjam, dan mengelola statistik. Sekarang setiap bagian memiliki kurang dari 7 persyaratan fungsional. Ini tidak hanya akan meningkatkan keterbacaan dokumen, tetapi juga akan membantu dalam desain.

Teknik untuk Mewakili Logika Kompleks

Dokumen SRS yang baik harus secara tepat mencirikan kondisi di mana skenario interaksi yang berbeda terjadi. Artinya, fungsi tingkat tinggi mungkin melibatkan langkah-langkah berbeda yang harus dilakukan sebagai konsekuensi dari beberapa keputusan yang dibuat setelah setiap langkah. Kadang-kadang kondisinya bisa kompleks dan banyak dan beberapa interaksi alternatif dan urutan pemrosesan mungkin ada tergantung pada hasil pemeriksaan kondisi yang sesuai. Deskripsi teks sederhana dalam kasus seperti itu bisa sulit untuk dipahami dan dianalisis. Dalam situasi seperti itu, pohon keputusan atau tabel keputusan dapat digunakan untuk mewakili logika dan pemrosesan yang terlibat. Juga, ketika pengambilan keputusan dalam kebutuhan fungsional telah direpresentasikan sebagai tabel keputusan, menjadi mudah untuk secara otomatis atau setidaknya secara manual merancang kasus uji untuk itu. Namun, penggunaan pohon keputusan atau tabel akan berlebihan dalam kasus di mana jumlah alternatifnya sedikit, atau logika keputusannya sederhana. Dalam kasus seperti itu, deskripsi teks sederhana sudah cukup.

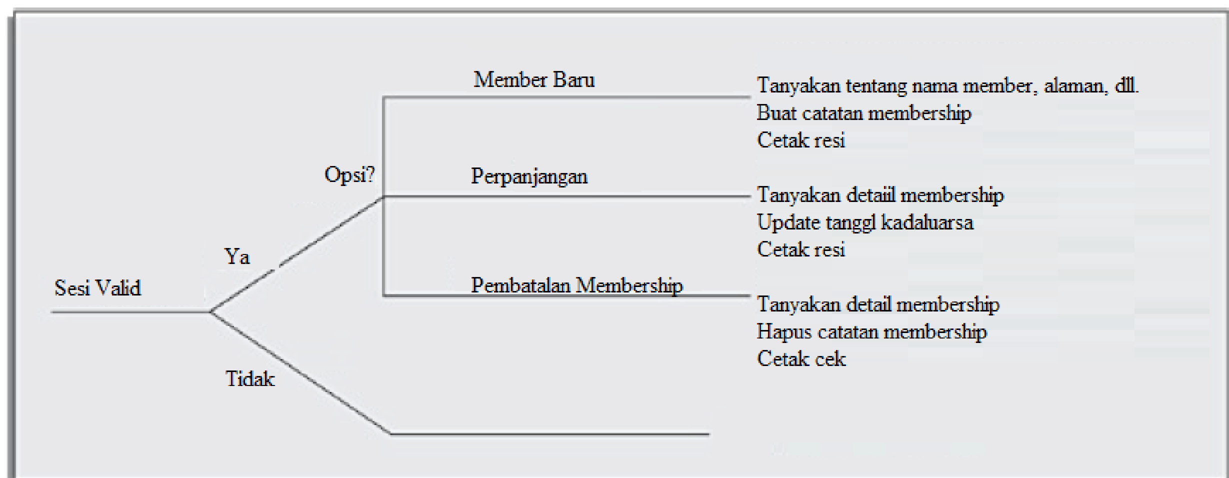
Ada dua teknik utama yang tersedia untuk menganalisis dan mewakili logika pemrosesan yang kompleks—pohon keputusan dan tabel keputusan. Setelah logika pengambilan keputusan ditangkap dalam bentuk pohon atau tabel, kasus uji untuk memvalidasi logika ini dapat diperoleh secara otomatis. Akan tetapi, harus dicatat bahwa pohon keputusan dan tabel keputusan memiliki penerapan yang jauh lebih luas daripada hanya menentukan logika pemrosesan yang kompleks dalam dokumen SRS. Misalnya, pohon keputusan dan tabel keputusan menemukan aplikasi dalam teori informasi dan teori switching.

Pohon keputusan

Pohon keputusan memberikan tampilan grafis dari logika pemrosesan yang terlibat dalam pengambilan keputusan dan tindakan terkait yang diambil. Tabel keputusan menentukan variabel mana yang akan diuji, dan berdasarkan ini tindakan apa yang harus diambil tergantung pada hasil logika pengambilan keputusan, dan urutan pengambilan keputusan dilakukan.

Tepi pohon keputusan mewakili kondisi dan simpul daun mewakili tindakan yang akan dilakukan tergantung pada hasil pengujian kondisi. Daripada membahas cara menggambar pohon keputusan untuk logika pemrosesan yang diberikan, saya akan menjelaskan melalui contoh sederhana bagaimana merepresentasikan logika pemrosesan dalam bentuk pohon keputusan.

Contoh 4.10 Perangkat lunak manajemen keanggotaan perpustakaan (LMS) harus mendukung tiga opsi berikut—anggota baru, pembaruan, dan pembatalan keanggotaan. Ketika opsi anggota baru dipilih, perangkat lunak harus menanyakan nama, alamat, dan nomor telepon anggota. Jika informasi yang benar dimasukkan, perangkat lunak harus membuat catatan keanggotaan untuk anggota baru dan mencetak tagihan untuk biaya keanggotaan tahunan dan uang jaminan yang harus dibayarkan. Jika opsi perpanjangan dipilih, LMS harus menanyakan nama anggota dan nomor keanggotaannya dan memeriksa apakah dia anggota yang valid. Jika rincian anggota yang dimasukkan valid, maka tanggal kedaluwarsa keanggotaan dalam catatan keanggotaan harus diperbarui dan biaya keanggotaan tahunan yang harus dibayar oleh anggota harus dicetak. Jika rincian keanggotaan yang dimasukkan tidak valid, pesan kesalahan akan ditampilkan. Jika opsi batalkan keanggotaan dipilih dan nama anggota yang valid dimasukkan, maka keanggotaan dibatalkan, cek untuk jumlah saldo karena anggota dicetak dan catatan keanggotaannya dihapus. Representasi pohon keputusan untuk masalah ini ditunjukkan pada Gambar 4.3.



Gambar 4.3 Pohon Keputusan untuk LMS.

Amati dari Gambar 4.3 bahwa simpul internal mewakili kondisi, tepi pohon sesuai dengan hasil dari kondisi yang sesuai. Node daun mewakili tindakan yang akan dilakukan oleh sistem. Pada pohon keputusan Gambar 4.3, pertama-tama pemilihan pengguna diperiksa. Berdasarkan apakah pilihan tersebut valid, pemeriksaan kondisi lebih lanjut dilakukan atau pesan kesalahan ditampilkan. Perhatikan bahwa urutan pemeriksaan kondisi secara eksplisit diwakili.

Tabel keputusan

Tabel keputusan menunjukkan logika pengambilan keputusan dan tindakan terkait yang diambil dalam bentuk tabel atau matriks. Baris atas tabel menentukan variabel atau

kondisi yang akan dievaluasi dan baris bawah menentukan tindakan yang harus diambil ketika tes evaluasi terpenuhi. Kolom dalam tabel disebut aturan. Aturan menyiratkan bahwa jika kombinasi kondisi tertentu benar, maka tindakan yang sesuai dijalankan. Tabel keputusan untuk masalah LMS dari Contoh 4.10 adalah seperti yang ditunjukkan pada Tabel 4.1.

Tabel 4.1 Tabel Keputusan untuk Soal LMS

Kondisi				
Seleksi valid	TIDAK	YA	YA	YA
Member baru	-	YA	TIDAK	TIDAK
Pembaruan	-	TIDAK	YA	TIDAK
Pembatalan	-	TIDAK	TIDAK	YA
Tampilkan pesan kesalahan	x			
Menanyakan nama anggota, dll.		x		
Buat catatan pelanggan		x		
Hasilkan tagihan		x	x	
Tanyakan detail keanggotaan			x	x
Perbarui tanggal kedaluwarsa			x	
Cetak cek				x
Hapus catatan				x

Tabel keputusan versus pohon keputusan

Meskipun kedua tabel keputusan dan pohon keputusan dapat digunakan untuk mewakili logika program yang kompleks, mereka dapat dibedakan berdasarkan tiga pertimbangan berikut:

Keterbacaan: Pohon keputusan lebih mudah dibaca dan dipahami ketika jumlah kondisi sedikit. Di sisi lain, tabel keputusan menyebabkan analis melihat setiap kemungkinan kombinasi kondisi yang mungkin dia hilangkan.

Representasi eksplisit dari urutan pengambilan keputusan: Berbeda dengan pohon keputusan, urutan pengambilan keputusan diabstraksikan dalam tabel keputusan. Situasi di mana pohon keputusan lebih berguna adalah ketika pengambilan keputusan bertingkat diperlukan. Pohon keputusan dapat lebih intuitif mewakili pengambilan keputusan bertingkat secara hierarkis, sedangkan tabel keputusan hanya dapat mewakili satu keputusan untuk memilih tindakan yang tepat untuk dieksekusi.

Mewakili logika keputusan yang kompleks: Pohon keputusan menjadi sangat kompleks untuk dipahami ketika jumlah kondisi dan tindakan meningkat. Bahkan mungkin menggambar pohon pada satu halaman. Ketika sejumlah besar keputusan terlibat, representasi tabel keputusan mungkin lebih disukai.

4.3 SPESIFIKASI SISTEM FORMAL

Dalam beberapa tahun terakhir, teknik formal3 telah muncul sebagai isu sentral dalam rekayasa perangkat lunak. Ini bukan kebetulan; pentingnya spesifikasi yang tepat, pemodelan, dan verifikasi diakui penting di sebagian besar disiplin ilmu teknik. Metode formal memberi kita alat untuk secara tepat menggambarkan suatu sistem dan menunjukkan bahwa suatu sistem diimplementasikan dengan benar. Suatu sistem diimplementasikan dengan benar ketika memenuhi spesifikasi yang diberikan. Spesifikasi sistem dapat diberikan baik sebagai daftar properti yang diinginkan (pendekatan berorientasi properti) atau sebagai model abstrak dari sistem (pendekatan berorientasi model). Kedua pendekatan ini dibahas di sini.

Sebelum membahas contoh representatif dari kedua jenis teknik spesifikasi formal ini, pertama-tama kita diskusikan beberapa konsep dasar dalam spesifikasi formal. Pertama-tama, kita akan menyoroti beberapa konsep penting dalam metode formal, dan memeriksa kelebihan dan kekurangan penggunaan teknik formal.

Apa itu Teknik Formal?

Teknik formal adalah metode matematis untuk menentukan sistem perangkat keras dan/atau perangkat lunak, memverifikasi apakah spesifikasi dapat direalisasikan, memverifikasi bahwa implementasi memenuhi spesifikasinya, membuktikan properti sistem tanpa harus menjalankan sistem, dll metode formal disediakan oleh bahasa spesifikasinya. Lebih tepatnya, bahasa spesifikasi formal terdiri dari dua *set-syn* dan sem, dan hubungan duduk di antara mereka. Himpunan *syn* disebut domain sintaksis, himpunan sem disebut domain semantik, dan relasi sat disebut relasi kepuasan. Untuk spesifikasi *syn*, dan model sistem sem yang diberikan, jika *sat* (*syn*, *sem),* maka *syn* dikatakan spesifikasi *sem*, dan sem disebut spesifikasi *syn*.

Paradigma yang diterima secara umum untuk pengembangan sistem adalah melalui hierarki abstraksi. Setiap tahap dalam hierarki ini merupakan implementasi dari tahap sebelumnya dan spesifikasi tahap berikutnya. Tahapan yang berbeda dalam aktivitas pengembangan sistem ini adalah spesifikasi kebutuhan, desain fungsional, desain arsitektur, desain detail, pengkodean, implementasi, dll. Secara umum, teknik formal dapat digunakan pada setiap tahapan aktivitas pengembangan sistem untuk memverifikasi bahwa output dari satu tahap sesuai dengan output tahap sebelumnya.

Domain sintaksis

Domain sintaksis bahasa spesifikasi formal terdiri dari alfabet simbol dan seperangkat aturan pembentukan untuk membangun formula yang terbentuk dengan baik dari alfabet. Rumus yang terbentuk dengan baik digunakan untuk menentukan sistem.

Domain semantik

Teknik formal dapat memiliki domain semantik yang sangat berbeda. Bahasa spesifikasi tipe data abstrak digunakan untuk menentukan aljabar, teori, dan program. Bahasa pemrograman digunakan untuk menentukan fungsi dari nilai input hingga output. Bahasa spesifikasi sistem serentak dan terdistribusi digunakan untuk menentukan urutan keadaan, urutan peristiwa, urutan transisi keadaan, pohon sinkronisasi, pesanan parsial, mesin keadaan, dll.

Hubungan kepuasan

Mengingat model sistem, penting untuk menentukan apakah elemen domain semantik memenuhi spesifikasi. Kepuasan ini ditentukan dengan menggunakan homomorfisme yang dikenal sebagai fungsi abstrak semantik. Fungsi abstraksi semantik memetakan elemen-elemen dari domain semantik ke dalam kelas-kelas yang setara. Mungkin ada spesifikasi berbeda yang menjelaskan aspek berbeda dari model sistem, mungkin menggunakan bahasa spesifikasi yang berbeda. Beberapa spesifikasi ini menggambarkan perilaku sistem dan yang lainnya menggambarkan struktur sistem. Akibatnya, dua kelas luas dari fungsi abstraksi semantik didefinisikan — yang melestarikan perilaku sistem dan yang melestarikan struktur sistem.

Model versus metode berorientasi properti

Metode formal biasanya diklasifikasikan ke dalam dua kategori besar—yang disebut pendekatan berorientasi model dan pendekatan berorientasi properti. Dalam gaya berorientasi model, seseorang mendefinisikan perilaku sistem secara langsung dengan membangun model sistem dalam bentuk struktur matematika seperti tupel, relasi, fungsi, himpunan, urutan, dll. Dalam gaya berorientasi properti, perilaku sistem didefinisikan secara

tidak langsung. dengan menyatakan sifat-sifatnya, biasanya dalam bentuk sekumpulan aksioma yang harus dipenuhi oleh sistem. Mari kita perhatikan contoh produsen/konsumen sederhana. Dalam gaya berorientasi properti, kita mungkin akan mulai dengan membuat daftar properti sistem seperti — konsumen dapat mulai mengkonsumsi hanya setelah produsen memproduksi suatu barang, produsen mulai memproduksi barang hanya setelah konsumen mengkonsumsi barang terakhir, dll. Dua contoh gaya spesifikasi berorientasi properti adalah spesifikasi aksiomatik dan spesifikasi aljabar. Dalam gaya berorientasi model, kita akan mulai dengan mendefinisikan operasi dasar, p (menghasilkan) dan c (mengkonsumsi). Kemudian kita dapat menyatakan bahwa S 1 + p⇒S, S + c⇒S 1. Jadi pendekatan berorientasi model pada dasarnya menentukan program dengan menulis program lain yang mungkin lebih sederhana. Beberapa contoh penting dari teknik spesifikasi berorientasi model yang populer adalah Z, CSP, CCS, dll.

Diduga bahwa pendekatan berorientasi properti lebih cocok untuk spesifikasi kebutuhan, dan pendekatan berorientasi model lebih cocok untuk spesifikasi desain sistem. Alasan untuk perbedaan ini adalah kenyataan bahwa pendekatan berorientasi properti menentukan perilaku sistem bukan dengan apa yang mereka katakan tentang sistem tetapi dengan apa yang tidak mereka katakan tentang sistem. Dengan demikian, spesifikasi berorientasi properti memungkinkan sejumlah besar kemungkinan implementasi. Lebih jauh lagi, pendekatan berorientasi properti menspesifikasikan sebuah sistem dengan aksioma-aksioma, dengan demikian membuatnya lebih mudah untuk mengubah/menambah spesifikasi pada tahap selanjutnya. Di sisi lain, metode berorientasi model tidak mendukung konjungsi dan disjungsi logis, dan dengan demikian bahkan perubahan kecil pada spesifikasi dapat menyebabkan perombakan seluruh spesifikasi. Karena persyaratan pelanggan awal mengalami beberapa perubahan saat pengembangan berlangsung, gaya berorientasi properti umumnya lebih disukai untuk spesifikasi persyaratan. Kemudian dalam bab ini, kita telah membahas dua teknik spesifikasi berorientasi properti.

Semantik Operasional

Secara informal, semantik operasional dari metode formal adalah cara komputasi direpresentasikan. Ada berbagai jenis semantik operasional menurut apa yang dimaksud dengan menjalankan sistem tunggal dan bagaimana menjalankan dikelompokkan bersama untuk menggambarkan perilaku sistem.

Semantik linier: Dalam pendekatan ini, menjalankan sistem dijelaskan oleh urutan (mungkin tak terbatas) dari peristiwa atau keadaan. Aktivitas bersamaan dari sistem diwakili oleh interleaving non-deterministik dari aksi atom. Misalnya, aktivitas bersamaan a || b diwakili oleh himpunan aktivitas berurutan a; B dan b; sebuah. Ini adalah representasi konkurensi yang sederhana namun agak tidak wajar. Perilaku sistem dalam model ini terdiri dari himpunan semua jalannya. Untuk membuat model ini lebih realistis, biasanya pembatasan keadilan dan kewajaran dikenakan pada perhitungan untuk mengecualikan interleaving yang tidak diinginkan.

Semantik bercabang: Dalam pendekatan ini, perilaku sistem diwakili oleh grafik berarah. Node grafik mewakili keadaan yang mungkin dalam evolusi suatu sistem. Keturunan dari setiap simpul grafik mewakili keadaan yang dapat dihasilkan oleh tindakan atom apa pun yang diaktifkan pada keadaan itu. Meskipun model semantik ini membedakan titik-titik percabangan dalam suatu komputasi, tetap saja model ini mewakili konkurensi dengan menyisipkan.

Semantik paralel maksimal: Dalam pendekatan ini, semua tindakan bersamaan yang diaktifkan pada keadaan apa pun diasumsikan diambil bersama. Ini sekali lagi bukan model

konkurensi alami karena secara implisit mengasumsikan ketersediaan semua sumber daya komputasi yang diperlukan.

Semantik urutan parsial: Di bawah pandangan ini, semantik dianggap berasal dari sistem adalah struktur negara memenuhi hubungan urutan parsial antara negara (peristiwa). Urutan parsial mewakili urutan prioritas di antara peristiwa, dan membatasi beberapa peristiwa terjadi hanya setelah beberapa peristiwa lain terjadi; sedangkan terjadinya peristiwa lain (disebut peristiwa bersamaan) dianggap tidak ada bandingannya. Fakta ini mengidentifikasi konkurensi sebagai fenomena yang tidak dapat diterjemahkan ke representasi interleaved apa pun.

Kelebihan dan keterbatasan metode formal

Selain memfasilitasi perumusan spesifikasi yang tepat, metode formal memiliki beberapa fitur positif, beberapa di antaranya dibahas sebagai berikut:

- Spesifikasi formal mendorong ketelitian. Sering terjadi bahwa proses konstruksi dari spesifikasi yang ketat lebih penting daripada spesifikasi formal itu sendiri. Konstruksi spesifikasi arogan menjelaskan beberapa aspek perilaku sistem yang tidak jelas dalam spesifikasi informal. Diakui secara luas bahwa menghabiskan lebih banyak upaya pada tahap spesifikasi adalah hemat biaya, jika tidak, banyak kekurangan yang tidak diperhatikan hanya untuk dideteksi pada tahap pengembangan perangkat lunak selanjutnya yang akan menyebabkan perubahan berulang terjadi dalam siklus hidup pengembangan. Menurut perkiraan, untuk sistem yang besar dan kompleks seperti sistem waktu nyata terdistribusi 80 persen dari biaya proyek dan sebagian besar kelebihan biaya dihasilkan dari perubahan berulang yang diperlukan dalam proses pengembangan sistem karena perumusan spesifikasi persyaratan yang tidak tepat. Dengan demikian, upaya tambahan yang diperlukan untuk membangun spesifikasi yang ketat sepadan dengan masalahnya.
- Metode formal biasanya memiliki dasar matematika yang kuat. Dengan demikian, spesifikasi formal tidak hanya lebih tepat, tetapi juga terdengar secara matematis dan dapat digunakan untuk menalar tentang sifat-sifat spesifikasi dan untuk membuktikan secara ketat bahwa suatu implementasi memenuhi spesifikasinya. Spesifikasi informal mungkin berguna dalam memahami sistem dan dokumentasinya, tetapi spesifikasi tersebut tidak dapat berfungsi sebagai dasar verifikasi. Bahkan spesifikasi yang ditulis dengan hati-hati rentan terhadap kesalahan, dan pengalaman telah menunjukkan bahwa spesifikasi yang belum diverifikasi sebanding dalam keandalannya dengan program yang tidak diverifikasi. secara otomatis dihindari ketika seseorang secara formal menentukan suatu sistem.
- Dasar matematika dari metode formal memungkinkan untuk mengotomatisasi analisis spesifikasi. Misalnya, teknik berbasis tablo telah digunakan untuk memeriksa konsistensi spesifikasi secara otomatis. Juga, teknik pembuktian teorema otomatis dapat digunakan untuk memverifikasi bahwa implementasi memenuhi spesifikasinya. Kemungkinan verifikasi otomatis adalah salah satu keuntungan terpenting dari metode formal.
- Spesifikasi formal dapat dijalankan untuk mendapatkan umpan balik langsung pada fitur dari sistem yang ditentukan. Konsep spesifikasi yang dapat dieksekusi ini terkait dengan pembuatan prototipe cepat. Secara informal, prototipe adalah model kerja "mainan" dari suatu sistem yang dapat memberikan umpan balik pada perilaku sistem yang ditentukan, dan sangat berguna dalam memeriksa kelengkapan spesifikasi.

Jelas bahwa metode formal menyediakan kerangka kerja yang kuat secara matematis di mana sistem yang besar dan kompleks dapat ditentukan, dikembangkan, dan diverifikasi secara sistematis daripada secara ad hoc. Namun, metode formal memiliki beberapa kekurangan, beberapa di antaranya adalah sebagai berikut:

- Metode formal sulit dipelajari dan digunakan.
- Hasil ketidaklengkapan dasar logika orde pertama menunjukkan bahwa tidak mungkin untuk memeriksa kebenaran mutlak sistem menggunakan teknik pembuktian teorema.
- Teknik formal tidak mampu menangani masalah yang kompleks. Kekurangan ini dihasilkan dari fakta bahwa, bahkan masalah yang cukup rumit pun meledakkan kompleksitas spesifikasi formal dan analisisnya. Juga, sekumpulan besar rumus matematika yang tidak terstruktur sulit untuk dipahami.

Telah ditunjukkan oleh beberapa peneliti bahwa spesifikasi formal tidak menggantikan atau membuat deskripsi informal menjadi usang tetapi melengkapinya. Faktanya, kelengkapan spesifikasi formal sangat ditingkatkan ketika spesifikasi disertai dengan deskripsi informal. Yang disarankan adalah penggunaan teknik formal sebagai pedoman luas untuk penggunaan teknik informal. Sebuah contoh menarik dari pendekatan tersebut dilaporkan oleh Jones pada [1980]. Dalam pendekatan ini, penggunaan metode formal mengidentifikasi langkah-langkah verifikasi yang diperlukan yang perlu dilakukan, tetapi sah untuk menerapkan penalaran informal dalam penyajian argumen kebenaran dan transformasi. Keraguan atau pertanyaan apa pun yang berkaitan dengan argumen informal harus diselesaikan dengan bukti formal. Dalam dua bagian berikut, kita membahas gaya spesifikasi aksiomatik dan aljabar. Kedua teknik ini dapat diklasifikasikan sebagai teknik spesifikasi berorientasi properti.

4.4 SPESIFIKASI AXIOMATIS

Dalam spesifikasi aksiomatik suatu sistem, logika orde pertama digunakan untuk menulis kondisi sebelum dan sesudah untuk menentukan operasi sistem dalam bentuk aksioma. Pra-kondisi pada dasarnya menangkap kondisi yang harus dipenuhi sebelum operasi dapat berhasil dipanggil. Intinya, pra-kondisi menangkap persyaratan pada parameter input suatu fungsi. Post-conditions adalah kondisi yang harus dipenuhi ketika suatu fungsi post-conditions pada dasarnya adalah kendala pada hasil yang dihasilkan agar eksekusi fungsi dianggap berhasil.

Bagaimana cara mengembangkan spesifikasi aksiomatik?

Berikut ini adalah urutan langkah-langkah yang dapat diikuti untuk mengembangkan spesifikasi aksiomatik suatu fungsi secara sistematis:

- Tetapkan rentang nilai input di mana fungsi harus berperilaku dengan benar. Tetapkan batasan pada parameter input sebagai predikat.
- Tentukan predikat yang mendefinisikan kondisi yang harus dipertahankan pada output fungsi jika berperilaku dengan benar.
- Tetapkan perubahan yang dibuat pada parameter input fungsi setelah eksekusi fungsi. Fungsi matematika murni tidak mengubah inputnya dan oleh karena itu pernyataan jenis ini tidak diperlukan untuk fungsi murni.
- Gabungkan semua hal di atas ke dalam kondisi pra dan pasca fungsi.

Sekarang mari mengilustrasikan bagaimana tipe data abstrak sederhana dapat ditentukan secara aljabar melalui dua contoh sederhana.

Contoh 4.11 Tentukan kondisi sebelum dan sesudah fungsi yang menggunakan bilangan real sebagai argumen dan mengembalikan setengah dari nilai input jika inputnya kurang dari atau sama dengan 100, atau mengembalikan nilai dua kali lipat.

f (x : nyata) : nyata
pra : $x \in R$
pos : $\{(x \leq 100) \wedge (f(x)=x/2)\} \vee \{(x>100) \wedge (f(x)=2*x)\}$.

4.5 SPESIFIKASI ALJABAR

Dalam teknik spesifikasi aljabar, kelas objek atau tipe ditentukan dalam hal hubungan yang ada antara operasi yang didefinisikan pada tipe itu. Ini pertama kali diperkenalkan oleh Guttag [1980,1985] dalam spesifikasi tipe data abstrak. Berbagai notasi spesifikasi aljabar telah berkembang, termasuk yang didasarkan pada bahasa OBJ dan Larch.

Pada dasarnya, spesifikasi aljabar mendefinisikan sistem sebagai aljabar heterogen. Aljabar heterogen adalah kumpulan himpunan yang berbeda di mana beberapa operasi didefinisikan. Aljabar tradisional adalah homogen. Sebuah aljabar homogen terdiri dari satu set dan beberapa operasi yang didefinisikan dalam set ini; misalnya { I, +, -, *, / }. Sebaliknya, string alfabet S bersama-sama dengan operasi penggabungan dan panjang {S, I , con, len}, bukanlah aljabar homogen, karena rentang operasi panjang adalah himpunan bilangan bulat. Setiap himpunan simbol dalam aljabar heterogen disebut semacam aljabar. Untuk mendefinisikan aljabar heterogen, selain mendefinisikan jenisnya, kita perlu menentukan operasi yang terlibat, tanda tangannya, dan domain serta rentangnya. Menggunakan spesifikasi aljabar, saya mendefinisikan arti dari satu set prosedur antarmuka dengan menggunakan persamaan. Spesifikasi aljabar biasanya disajikan dalam empat bagian.

Tipe bagian: Di bagian ini, jenis (atau tipe data) yang digunakan ditentukan.

Bagian pengecualian: Bagian ini memberikan nama kondisi luar biasa yang mungkin terjadi ketika operasi yang berbeda dilakukan. Kondisi pengecualian ini digunakan di bagian selanjutnya dari spesifikasi aljabar.

Bagian sintaks: Bagian ini mendefinisikan tanda tangan dari prosedur antarmuka. Kumpulan set yang membentuk domain input dari operator dan pengurutan di mana output dihasilkan disebut tanda tangan operator. Misalnya, PUSH mengambil tumpukan dan elemen sebagai inputnya dan mengembalikan tumpukan baru yang telah dibuat.

Bagian persamaan: Bagian ini memberikan seperangkat aturan penulisan ulang (atau persamaan) yang mendefinisikan arti dari prosedur antarmuka dalam hal satu sama lain. Secara umum, bagian ini diperbolehkan berisi ekspresi kondisional.

Dengan konvensi, setiap persamaan secara implisit dikuantifikasi secara universal atas semua kemungkinan nilai variabel. Ini berarti bahwa persamaan berlaku untuk semua kemungkinan nilai variabel. Nama yang tidak disebutkan di bagian sintaks seperti *r* atau *e* adalah variabel. Langkah pertama dalam mendefinisikan spesifikasi aljabar adalah mengidentifikasi himpunan operasi yang diperlukan. Setelah mengidentifikasi operator yang diperlukan, akan sangat membantu untuk mengklasifikasikan mereka sebagai operator konstruktor dasar, operator konstruktor tambahan, operator inspektur dasar, atau operator inspeksi tambahan. Definisi dari kategori operator tersebut adalah sebagai berikut:

Operator konstruksi dasar: Operator ini digunakan untuk membuat atau memodifikasi entitas dari suatu tipe. Operator konstruksi dasar sangat penting untuk menghasilkan semua elemen yang mungkin dari tipe yang ditentukan. Misalnya, 'buat' dan 'tambahkan' adalah operator konstruksi dasar dalam Contoh 4.13. Operator konstruksi tambahan: Ini adalah operator konstruksi selain operator konstruksi dasar. Misalnya, operator 'remove' dalam Contoh 4.13 adalah operator konstruksi tambahan, karena bahkan tanpa menggunakan 'remove' dimungkinkan untuk menghasilkan semua nilai dari tipe yang ditentukan.

Operator inspeksi dasar: Operator ini mengevaluasi atribut suatu tipe tanpa memodifikasinya, misalnya eval, get, dll. Misalkan S adalah himpunan operator yang

jangkauannya bukan tipe data yang ditentukan—ini adalah operator inspeksi. Himpunan operator dasar S1 adalah himpunan bagian dari S , sehingga setiap operator dari S -S 1 dapat dinyatakan dalam operator dari S 1.

Operator inspeksi tambahan: Ini adalah operator inspeksi yang bukan inspektur dasar. Cara sederhana untuk menentukan apakah operator adalah konstruktor (dasar atau ekstra) atau inspektur (dasar atau ekstra) adalah dengan memeriksa ekspresi sintaks untuk operator. Jika tipe yang ditentukan muncul di sisi kanan ekspresi maka itu adalah konstruktor, jika tidak, itu adalah operator inspeksi. Misalnya, dalam Contoh 4.13, create adalah konstruktor karena titik muncul di sisi kanan ekspresi dan titik adalah tipe data yang ditentukan. Namun, xcoord adalah operator inspeksi karena tidak mengubah tipe titik.

Aturan praktis yang baik saat menulis spesifikasi aljabar, adalah pertama-tama menetapkan mana yang merupakan konstruktor (dasar dan tambahan) dan operator inspeksi (dasar dan tambahan). Kemudian tuliskan aksioma untuk komposisi setiap operator konstruksi dasar di atas setiap operator inspeksi dasar dan operator konstruktor tambahan. Juga, tuliskan aksioma untuk masing-masing inspektur tambahan dalam kaitannya dengan inspektur dasar. Jadi, jika ada m1 konstruktor dasar, m2 konstruktor tambahan, n1 inspektur dasar, a n d n2 inspektur tambahan, kita harus memiliki aksioma m1 × (m2 + n1) + n2. Namun, harus dicatat dengan jelas bahwa aksioma m1 × (m2 + n1) + n2 ini adalah minimum yang diperlukan dan banyak lagi aksioma mungkin diperlukan untuk melengkapi spesifikasi. Menggunakan seperangkat aturan penulisan ulang yang lengkap, dimungkinkan untuk menyederhanakan urutan operasi sewenang-wenang pada prosedur antarmuka.

Saat mengembangkan aturan penulisan ulang, orang yang berbeda dapat menghasilkan set persamaan yang berbeda. Namun, saat mengembangkan persamaan, kita harus berhati-hati bahwa persamaan harus dapat menangani semua komposisi operator yang berarti, dan mereka harus memiliki sifat terminasi dan terminasi yang unik. Kedua properti aturan penulisan ulang ini dibahas nanti di bagian ini.

Contoh 4.13 Mari kita tentukan titik tipe data yang mendukung operasi create, xcoord, ycoord, isequal; di mana operasi memiliki arti yang biasa.

Jenis:

mendefinisikan titik

menggunakan boolean, integer

Sintaksis:

1. buat : integer × integer→point
2. xcoord : titik→bilangan bulat
3. ycoord : titik→bilangan bulat
4. sama : titik × titik→boolean

Persamaan:

1. xcoord(crreate(x, y)) = x
2. ycoord(create(x, y)) = y
3. same(create(x1, y1), buat(x2, y2)) = ((x1 = x2)dan(y1 = y2))

Dalam contoh ini, kita hanya memiliki satu konstruktor dasar (buat), dan tiga pemeriksa dasar (xcoord, ycoord, dan isequal). Oleh karena itu, kita hanya memiliki 3 persamaan. Aturan penulisan ulang memungkinkan Anda menentukan arti dari setiap urutan panggilan pada jenis titik. Perhatikan ekspresi berikut: isequal (create e (xcoord (create(2, 3)), 5),create (ycoord (create(2, 3)), 5)). Dengan menerapkan aturan penulisan ulang 1, Anda dapat menyederhanakan ekspresi yang diberikan sebagai isequal (create (2, 5), create (ycoord (create(2, 3)), 5)). Dengan menggunakan aturan penulisan ulang 2, Anda dapat

menyederhanakan ini lebih lanjut sebagai sama (buat (2, 5), buat (3, 5)). Ini salah dengan menulis ulang aturan 3.

Sifat spesifikasi aljabar

Tiga sifat penting yang harus dimiliki setiap spesifikasi aljabar adalah:

Kelengkapan: Properti ini memastikan bahwa dengan menggunakan persamaan, harus dimungkinkan untuk mengurangi urutan operasi sembarang pada prosedur antarmuka. Ketika persamaan tidak lengkap, pada beberapa langkah selama proses reduksi, kita mungkin tidak dapat mereduksi ekspresi yang sampai pada langkah tersebut dengan menggunakan salah satu persamaan. Tidak ada prosedur sederhana untuk memastikan bahwa spesifikasi aljabar sudah lengkap.

Properti terminasi terbatas: Properti ini pada dasarnya menjawab pertanyaan berikut: Apakah aplikasi aturan penulisan ulang ke ekspresi arbitrer yang melibatkan prosedur antarmuka selalu berakhir? Untuk persamaan aljabar arbitrer, konvergensi (terminasi hingga) tidak dapat diputuskan. Namun, jika sisi kanan dari setiap aturan penulisan ulang memiliki istilah yang lebih sedikit daripada sisi kiri, maka proses penulisan ulang harus dihentikan.

Properti terminasi unik: Properti ini menunjukkan apakah penerapan aturan penulisan ulang dalam urutan yang berbeda selalu menghasilkan jawaban yang sama. Pada dasarnya, untuk menentukan properti ini, jawaban atas pertanyaan berikut perlu diperiksa—Dapatkah semua urutan pilihan yang mungkin dalam penerapan aturan penulisan ulang ke ekspresi arbitrer yang melibatkan prosedur antarmuka selalu memberikan jawaban yang sama? Memeriksa properti terminasi unik adalah masalah yang sangat sulit.

Contoh 4.14 Mari kita tentukan antrian FIFO yang mendukung operasi create, append, remove, first, dan isempty; di mana operasi memiliki arti yang biasa.

Jenis:

mendefinisikan antrian

menggunakan boolean, elemen

Pengecualian:

arus bawah, tidak ada nilai

Sintaksis:

1. buat :φ→antrian
2. tambahkan : antrian × elemen → antrian
3. delete : antrian → antrian + {underf low}
4. fisrst : antrian → elemen + {novalue}
5. empty : antrian → boolean

Persamaan:

1. empty(create()) = benar
2. empty(add(q, e)) = f alse
3. first(create()) = tidak bernilai
4. f pertama(tambahkan(q, e)) = jika kosong(q) maka e lain pertama(q)
5. delete(create()) = underf rendah
6. delete(add(q, e)) = jika kosong(q) lalu buat( ) lain tambahkan(hapus(q), e)

Dalam contoh ini, kita memiliki dua operator konstruksi dasar (buat dan tambahkan). Kita memiliki satu operator konstruksi tambahan (hapus). *Remove* sebagai operator konstruksi tambahan karena semua nilai antrian dapat direalisasikan, bahkan tanpa operator hapus. Kita memiliki dua inspektur dasar (pertama dan kosong). Oleh karena itu kita memiliki 2 × 3 = 6 persamaan.

Fungsi Bantu

Kadang-kadang saat menentukan sistem, seseorang perlu memperkenalkan fungsi tambahan bukan bagian dari sistem untuk mendefinisikan arti dari beberapa prosedur antarmuka. Ini disebut fungsi bantu. Berikut ini, kita membahas contoh di mana menjadi perlu untuk menggunakan fungsi bantu untuk dapat menentukan sistem.

Contoh 4.15 Mari kita tentukan antrian FIFO terbatas yang memiliki ukuran maksimum MaxSize dan mendukung operasi buat, tambahkan, hapus, pertama, dan kosong; di mana operasi memiliki arti yang biasa.

Jenis:

mendefinisikan antrian

menggunakan boolean, elemen, integer

Pengecualian:

underflow, novalue, overflow

Sintaksis:

1. buat :φ→antrian
2. tambahkan : antrian × elemen → antrian + {overf low}
3. ukuran : antrian → bilangan bulat
4. hapus : antrian → antrian + {underf low}
5. pertama : antrian → elemen + {novalue}
6. kosong : antrian → boolean

Persamaan:

1. first(CREATE()) = tidak bernilai
2. first(append(q,e)) = if size(q)= MaxSize lalu overflow else i f isempty(q) maka e else first(q)
3. delete(CREATE()) = underf rendah
4. delete(add(q, e)) = jika kosong(q) lalu buat() else. if size(q) = M axS ize lalu overf low else append(remove(q), e)
5. size(create()) = 0
6. size(append(q, e)) = if size(q) = M axS ize maka overf low else size(q) + 1
7. isempty(q) = if (size(q) = 0) maka true elsef alse

Dalam contoh ini, kita menggunakan ukuran fungsi tambahan untuk memungkinkan kita dalam menentukan bahwa selama menambahkan elemen, overflow mungkin terjadi jika ukuran antrian melebihi MaxSize. Namun, setelah ada ukuran fungsi bantu, kita dapat menemukan bahwa operator kosong tidak lagi dapat dianggap sebagai pemeriksa dasar karena kosong dapat dinyatakan dalam ukuran. Sehingga, aksioma untuk operator *issmpty* yang digunakan dalam Contoh 4.15 dihapus, dan sebagai gantinya kita dapat menggunakan aksioma untuk menyatakan *isempty* dalam hal ukuran. Dua aksioma untuk menyatakan ukuran dalam hal operator konstruksi dasar juga ditambahkan (membuat dan menambahkan).

Spesifikasi Terstruktur

Mengembangkan spesifikasi aljabar memakan waktu. Oleh karena itu upaya telah dilakukan untuk menemukan cara untuk memudahkan tugas mengembangkan spesifikasi aljabar. Berikut ini adalah beberapa teknik yang telah berhasil digunakan untuk mengurangi upaya dalam menulis spesifikasi.

Spesifikasi inkremental: Ide di balik spesifikasi inkremental adalah pertama-tama mengembangkan spesifikasi tipe sederhana dan kemudian menentukan tipe yang lebih kompleks dengan menggunakan spesifikasi tipe sederhana.

Spesifikasi Instansiasi: Ini melibatkan mengambil spesifikasi yang ada yang telah dikembangkan menggunakan parameter generik dan instantiating dengan beberapa jenis lain.

Pro dan Kontra spesifikasi aljabar

Spesifikasi aljabar memiliki dasar matematika yang kuat dan dapat dipandang sebagai aljabar heterogen. Oleh karena itu, mereka tidak ambigu dan tepat. Menggunakan spesifikasi aljabar, efek dari sembarang urutan operasi yang melibatkan prosedur antarmuka dapat dipelajari secara otomatis. Kelemahan utama dari spesifikasi aljabar adalah bahwa mereka tidak dapat menangani efek samping. Oleh karena itu, spesifikasi aljabar sulit untuk diintegrasikan dengan bahasa pemrograman biasa. Juga, spesifikasi aljabar sulit dipahami.

4.6 SPESIFIKASI YANG DAPAT DILAKUKAN DAN 4GL

Ketika spesifikasi suatu sistem dinyatakan secara formal atau dijelaskan dengan menggunakan bahasa pemrograman, maka spesifikasi dapat langsung dieksekusi tanpa harus merancang dan menulis kode untuk implementasi. Namun, spesifikasi yang dapat dieksekusi biasanya lambat dan tidak efisien, 4GLs4 (Bahasa Generasi ke-4) adalah contoh bahasa spesifikasi yang dapat dieksekusi. 4GL berhasil karena ada banyak kesamaan granularity besar di seluruh aplikasi pemrosesan data yang telah diidentifikasi dan dipetakan ke kode program. 4GL mendapatkan kekuatannya dari penggunaan kembali perangkat lunak, di mana abstraksi umum telah diidentifikasi dan diparameterisasi. Eksperimen yang cermat telah menunjukkan bahwa menulis ulang program 4GL dalam 3GL menghasilkan penggunaan memori hingga 50 persen lebih rendah dan juga waktu eksekusi program dapat berkurang hingga sepuluh kali lipat.

4.7 RINGKASAN

- Banyak waktu dan usaha harus dihabiskan dalam mengembangkan dokumen SRS yang berkualitas sebelum memulai aktivitas desain. Setiap spesifikasi yang tidak tepat dari fase ini akan mempengaruhi semua fase pengembangan selanjutnya dan memiliki biaya yang sangat mahal.
- Analisis kebutuhan dan fase spesifikasi terdiri dari dua aktivitas penting
  - pengumpulan dan analisis kebutuhan
  - spesifikasi kebutuhan.
- Tujuan dari analisis persyaratan adalah untuk memahami dengan jelas persyaratan pengguna yang tepat dan untuk menghilangkan inkonsistensi, anomali, dan ketidaklengkapan dalam persyaratan ini.
- Selama aktivitas spesifikasi persyaratan, secara sistematis persyaratan diatur ke dalam dokumen SRS.
- Menetapkan persyaratan secara formal memiliki banyak keuntungan. Kelemahan utama dari teknik spesifikasi formal adalah sulit untuk digunakan. Namun, ada kemungkinan bahwa teknik formal akan menjadi lebih bermanfaat di masa depan dengan pengembangan frontend yang sesuai. gambaran tentang beberapa masalah yang terlibat dalam spesifikasi formal.

4.8 LATIHAN

1. Pilih opsi yang benar.
   a. (Siapa di antara berikut ini yang merupakan pemangku kepentingan dalam proyek pengembangan perangkat lunak?
      i. Pemegang saham organisasi yang mengembangkan perangkat lunak
      ii. Siapa pun yang tertarik dengan perangkat lunak
      iii. Siapa pun yang merupakan sumber persyaratan untuk perangkat lunak
      iv. Siapa pun yang mungkin terpengaruh oleh perangkat lunak

b. Dokumen spesifikasi persyaratan perangkat lunak (SRS) harus menghindari pembahasan yang mana dari berikut ini?
    i. Persyaratan fungsional
    ii. Persyaratan non-fungsional
    iii. Spesifikasi desain
    iv. Kendala pada implementasi
c. Manakah dari berikut ini yang bukan merupakan tujuan dari analisis kebutuhan?
    i. Singkirkan ambiguitas dalam persyaratan
    ii. Menyingkirkan inkonsistensi dalam persyaratan
    iii. Singkirkan persyaratan non-fungsional
    iv. Menyingkirkan ketidaklengkapan dalam persyaratan
d. Pertimbangkan persyaratan berikut untuk perangkat lunak pengolah kata: "Perangkat lunak harus menyediakan fasilitas untuk mengimpor gambar yang ada yang tersedia sebagai file jpeg ke dalam dokumen yang sedang dibuat." Manakah dari jenis persyaratan berikut ini?
    i. Persyaratan fungsional
    ii. Persyaratan non-fungsional
    iii. Kendala pada implementasi
    iv. Tujuan implementasi
e. Asumsikan bahwa Anda adalah manajer proyek dari proyek pengembangan untuk aplikasi pemrosesan data di mana persyaratan pengguna untuk bagian GUI tidak terlalu jelas. Model siklus hidup mana yang akan Anda gunakan untuk mengembangkan bagian GUI?
    i. Model air terjun klasik
    ii. Model air terjun berulang
    iii. Model prototipe
    iv. Model spiral

2. Apa perbedaan antara analisis persyaratan dan spesifikasi? Apa saja aktivitas penting yang dilakukan selama fase analisis kebutuhan dan spesifikasi? Apa hasil akhir dari fase analisis kebutuhan dan spesifikasi?
3. Apa tujuan dari fase analisis kebutuhan dan spesifikasi? Bagaimana kegiatan analisis kebutuhan dan spesifikasi dilakukan dan oleh siapa?
4. Diskusikan cara-cara penting di mana dokumen SRS yang dirumuskan dengan baik dapat bermanfaat bagi berbagai pemangku kepentingan.
5. Apa perbedaan antara persyaratan fungsional dan non-fungsional dari suatu sistem? Identifikasi setidaknya dua persyaratan fungsional dari sistem mesin anjungan tunai mandiri (ATM) bank. Juga mengidentifikasi satu kebutuhan non-fungsional untuk sistem ATM.
6. Apa empat jenis kebutuhan non-fungsional yang diusulkan oleh dokumen standar IEEE 830. Berikan satu contoh dari masing-masing kategori persyaratan ini.
7. Apa yang Anda pahami dengan pengumpulan persyaratan? Sebutkan dan jelaskan teknik pengumpulan persyaratan yang berbeda yang biasanya digunakan oleh seorang analis.
8. Apa saja jenis masalah persyaratan yang biasanya diantisipasi dan diperbaiki oleh analis dalam persyaratan yang dikumpulkan sebelum mulai menulis dokumen SRS? Berikan setidaknya satu contoh untuk masing-masing.
9. Jelaskan kemungkinan konsekuensi memulai upaya pengembangan proyek besar tanpa secara akurat memahami dan mendokumentasikan persyaratan pelanggan.

10. Misalkan Anda telah ditunjuk sebagai analis untuk proyek pengembangan perangkat lunak yang besar. Diskusikan aspek produk perangkat lunak yang akan Anda dokumentasikan dalam dokumen spesifikasi persyaratan perangkat lunak (SRS)? Apa yang akan menjadi organisasi dokumen SRS Anda? Bagaimana Anda akan memvalidasi dokumen SRS Anda?
11. Buat daftar periksa kesalahan yang mungkin ada dalam dokumen SRS. Daftar periksa ini dapat digunakan untuk meninjau dokumen SRS.
12. Tuliskan pengguna penting dari dokumen SRS untuk sebuah proyek, cara spesifik di mana mereka menggunakan dokumen tersebut, dan harapan spesifik mereka dari dokumen tersebut, jika ada.
13. Apa yang Anda pahami dengan masalah spesifikasi berlebih, referensi ke depan, dan noise dalam dokumen SRS? Jelaskan masing-masing dengan contoh yang sesuai.
14. Apa perbedaan antara persyaratan fungsional dan non-fungsional? Berikan satu contoh dari setiap jenis persyaratan untuk perangkat lunak otomatisasi perpustakaan.
15. Sebutkan lima karakteristik yang diinginkan dari dokumen spesifikasi persyaratan perangkat lunak (SRS) yang baik.
16. Misalkan Anda mencoba mengumpulkan persyaratan untuk perangkat lunak yang perlu dikembangkan untuk mengotomatisasi kegiatan pembukuan supermarket. Identifikasi tugas utama yang akan Anda lakukan sebagai analis untuk mengumpulkan persyaratan secara memuaskan.
17. Bagaimana prinsip abstraksi dan dekomposisi digunakan dalam pengembangan spesifikasi kebutuhan perangkat lunak yang baik?
18. Misalkan analis dari usaha pengembangan produk besar telah menyiapkan dokumen SRS dalam bentuk esai naratif dari sistem yang akan dikembangkan. Berdasarkan dokumen ini, aktivitas pengembangan produk dimulai. Jelaskan masalah yang mungkin dibuat oleh dokumen spesifikasi persyaratan saat mengembangkan perangkat lunak.
19. Diskusikan keuntungan relatif dari spesifikasi persyaratan formal dan informal.
20. Mengapa dokumen SRS disebut juga sebagai spesifikasi kotak hitam suatu sistem?
21. Siapa saja kategori pengguna dokumen SRS yang berbeda? Dalam hal apa dokumen SRS berguna bagi mereka?
22. Berikan contoh kebutuhan fungsional yang tidak konsisten. Jelaskan mengapa menurut Anda persyaratan tersebut tidak konsisten.
23. Apa yang Anda pahami dengan ketertelusuran dalam konteks spesifikasi persyaratan perangkat lunak. Bagaimana ketertelusuran dicapai? Identifikasi setidaknya dua manfaat penting memiliki ketertelusuran di antara artefak pengembangan.
24. Nyatakan apakah pernyataan berikut BENAR atau SALAH. Berikan alasan untuk jawaban Anda.
    a. Aplikasi yang dikembangkan menggunakan 4GL biasanya akan lebih efisien dan berjalan lebih cepat dibandingkan dengan aplikasi yang dikembangkan menggunakan 3GL.
    b. Spesifikasi formal tidak boleh ambigu.
    c. Spesifikasi formal tidak boleh tidak lengkap.
    d. Spesifikasi formal tidak boleh tidak konsisten.
    e. Rencana pengujian sistem dapat disiapkan segera setelah selesainya fase spesifikasi persyaratan.
    f. Dokumen SRS adalah spesifikasi formal dari suatu sistem.
    g. Masalah antarmuka pengguna suatu sistem biasanya merupakan persyaratan fungsionalnya.

h. Dokumen SRS ditulis menggunakan terminologi pelanggan dari berbagai data dan prosedur dalam masalah, daripada terminologi tim pengembangan.
i. Spesifikasi yang tepat tidak mungkin tidak lengkap.
j. Jika spesifikasi persyaratan tepat, maka secara otomatis akan menyiratkan bahwa itu adalah spesifikasi persyaratan yang tidak ambigu.

25. Gambarkan logika pemrosesan dari masalah berikut dalam bentuk tabel keputusan: Sistem Otomasi Keanggotaan Perpustakaan perlu mendukung tiga fungsi: add new member, memperbarui keanggotaan, membatalkan keanggotaan. Jika pengguna meminta fungsi apa pun selain dari ketiga fungsi ini, maka pesan kesalahan akan di-flash. Ketika permintaan tambah anggota baru dibuat, catatan anggota baru dibuat dan tagihan untuk biaya keanggotaan tahunan untuk anggota baru dibuat. Jika permintaan perpanjangan keanggotaan dibuat, maka tanggal kedaluwarsa dari catatan keanggotaan yang bersangkutan diperbarui dan tagihan untuk biaya keanggotaan tahunan dibuat. Jika permintaan pembatalan keanggotaan dibuat, maka catatan keanggotaan yang bersangkutan dihapus dan cek untuk jumlah saldo yang harus dibayar anggota dicetak.
26. Apa yang Anda pahami dengan kondisi pra dan pasca suatu fungsi? Tulis pra-dan pasca-kondisi untuk secara aksiomatis menentukan fungsi-fungsi berikut:
    a. Sebuah fungsi mengambil dua bilangan floating point yang mewakili sisi-sisi persegi panjang sebagai input dan mengembalikan luas dari yang sesuai persegi panjang sebagai output.
    b. Suatu fungsi menerima tiga bilangan bulat dalam kisaran -100 dan +100 dan menentukan yang terbesar dari ketiga bilangan bulat tersebut.
    c. Sebuah fungsi mengambil array bilangan bulat sebagai input dan menemukan nilai minimumnya.
    d. Sebuah fungsi bernama square-array membuat 10 elemen array dimana setiap elemen array, nilai setiap elemen array adalah kuadrat dari indeksnya.
    e. Pengurutan fungsi mengambil array integer sebagai argumennya dan mengurutkan array input dalam urutan menaik.
27. Dengan menggunakan metode spesifikasi aljabar, tentukan secara formal string yang mendukung operasi berikut:
    a. tambahkan: tambahkan string yang diberikan ke string lain
    b. tambahkan: menambahkan karakter ke string
    c. buat: buat string nol baru
    d. isequal: memeriksa apakah dua string sama atau tidak
    e. isempty: memeriksa apakah string tersebut null
28. Menggunakan metode spesifikasi aljabar, tentukan secara formal array elemen tipe generik. Asumsikan bahwa array mendukung operasi berikut:
    a. create: mengambil batas larik sebagai parameter dan menginisialisasi nilai larik ke undefined.
    b. eval: mengungkapkan nilai elemen tertentu.
    c. first: mengembalikan batas pertama larik.
    d. last: mengembalikan batas terakhir larik.
29. Apa yang Anda pahami dengan bahasa spesifikasi yang dapat dieksekusi? Apa bedanya dengan bahasa pemrograman prosedural tradisional? Berikan contoh bahasa spesifikasi yang dapat dieksekusi.
30. Apa yang dimaksud dengan teknik pemrograman generasi keempat? Apa kelebihan dan kekurangannya dibandingkan dengan teknik generasi ketiga?
31. Apa fungsi bantu dalam spesifikasi aljabar? Mengapa ini dibutuhkan?

32. Apa yang Anda pahami dengan perkembangan inkremental dari spesifikasi aljabar? Apa keuntungan dari pengembangan inkremental dari spesifikasi aljabar?
    a. Secara aljabar tentukan tipe data abstrak yang menyimpan sekumpulan elemen dan mendukung operasi berikut. Asumsikan bahwa elemen ADT telah ditentukan dan Anda dapat menggunakannya:
        i. new: membuat set nol.
        ii. add : mengambil set dan elemen dan mengembalikan set dengan elemen tambahan yang disimpan.
        iii. size: mengambil satu set sebagai argumen dan mengembalikan jumlah elemen dalam set.
        iv. remove: mengambil set dan elemen sebagai argumennya dan mengembalikan set dengan elemen yang dihapus.
        v. fill: satu set dan elemen sebagai argumennya dan mengembalikan nilai boolean true jika elemen termasuk dalam himpunan dan mengembalikan false jika elemen bukan milik himpunan.
        vi. equals: mengambil dua set sebagai argumen dan mengembalikan true jika mengandung elemen identik dan mengembalikan false sebaliknya.
    b. Menggunakan spesifikasi yang telah Anda kembangkan untuk himpunan ADT, kurangi ekspresi berikut dengan menerapkan aturan penulisan ulang: equals (add(5, (add(6, new()), add(6, (add(5, new ()))).Tampilkan detail setiap pengurangan.
33. Secara aljabar tentukan tipe data Point, yang mendukung operasi berikut: create, xcoord, ycoord, move, movex, movey. Arti informal dari operasi ini adalah sebagai berikut—create mengambil dua bilangan bulat sebagai argumennya dan membuat turunan tipe titik yang memiliki dua bilangan bulat sebagai nilai koordinat x dan y masing-masing, xcoord dan ycoord mengembalikan koordinat x dan y dari a titik yang diberikan, move mengambil sebuah titik dan dua nilai integer sebagai argumennya dan menetapkan koordinat x dan y dari titik tersebut ke nilai yang ditentukan, movex mengambil sebuah titik dan sebuah nilai integer sebagai argumennya dan menetapkan koordinat x dari titik tersebut ke nilai integer yang diberikan. Demikian pula, movey mengambil titik dan nilai bilangan bulat dan menetapkan koordinat y dari titik ke nilai bilangan bulat yang diberikan. Kurangi ekspresi berikut, tunjukkan dengan jelas setiap langkah dan sebutkan aturan pengurangan yang digunakan. xcoord(movex(create(20.100), ycoord(create(10,50)))
34. Tulis spesifikasi aljabar formal dari tabel simbol pengurutan yang operasinya didefinisikan secara informal sebagai berikut:
    a. create: membuat tabel simbol menjadi ada.
    b. enter: memasukkan tabel simbol dan tipenya ke dalam tabel.
    c. lookup: mengembalikan tipe yang terkait dengan nama dalam tabel.
    d. delete: hapus nama, ketik pasangan dari tabel, beri nama sebagai parameter.
    e. replace: mengganti tipe yang terkait dengan nama yang diberikan dengan tipe yang ditentukan sebagai parameternya. Operasi enter gagal jika nama sudah ada di tabel. Operasi pencarian, penghapusan, dan penggantian gagal jika nama tidak tersedia dalam tabel.
35. Tentukan secara aljabar antrian tipe data yang mendukung operasi berikut:
    a. create: membuat antrian kosong.
    b. append: mengambil antrian dan item sebagai argumennya dan mengembalikan antrian dengan item ditambahkan di akhir antrian.

c. delete: mengambil antrian sebagai argumennya dan mengembalikan antrian dengan elemen pertama dari antrian asli dihapus.
d. inspect: mengambil antrian sebagai argumennya dan mengembalikan nilai item pertama dalam antrian.
e. isempty: mengambil antrian sebagai argumennya dan mengembalikan true jika antrian tidak mengandung elemen, dan mengembalikan false jika berisi satu atau lebih elemen.
f. Anda dapat mengasumsikan bahwa item tipe data telah ditentukan sebelumnya dan Anda dapat menggunakan kembali spesifikasi ini.

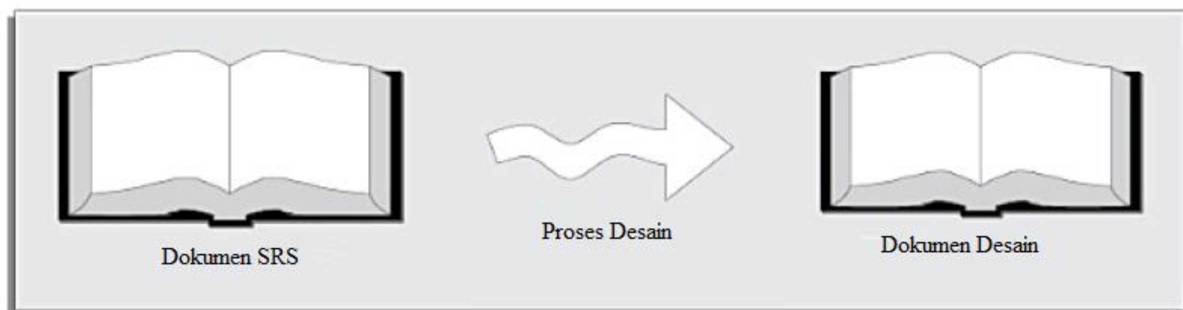
36. Tentukan terminasi hingga dan properti terminasi unik dari spesifikasi aljabar? Mengapa diperlukan spesifikasi aljabar untuk memenuhi sifat-sifat ini?
37. Jika model prototyping digunakan dalam upaya pengembangan, apakah perlu untuk mengembangkan dokumen spesifikasi persyaratan?
38. Nyatakan pengambilan keputusan yang terlibat dalam fungsi penarikan tunai berikut dari ATM bank menggunakan tabel keputusan. Untuk menarik uang tunai, pertama-tama diperlukan identifikasi pelanggan yang valid. Untuk ini, pelanggan diminta untuk memasukkan kartu ATM-nya ke mesin pengecekan kartu. Jika kartunya ditemukan tidak valid, kartu dikeluarkan bersama dengan pesan yang sesuai ditampilkan. Jika kartu diverifikasi sebagai kartu yang valid, pelanggan diminta untuk mengetikkan kata sandinya. Jika kata sandi tidak valid, pesan kesalahan ditampilkan dan pelanggan diminta untuk memasukkan kata sandinya lagi. Jika nasabah salah memasukkan password sebanyak tiga kali berturut-turut, maka kartunya disita dan diminta untuk menghubungi pengelola bank. Di sisi lain, jika pelanggan memasukkan kata sandinya dengan benar, maka dia dianggap telah mengidentifikasi dirinya secara sah dan diminta untuk memasukkan jumlah yang perlu dia tarik. Jika ia memasukkan jumlah yang bukan kelipatan Rp. 100 ribu, dia diminta untuk memasukkan jumlah lagi. Setelah ia memasukkan jumlah yang merupakan kelipatan Rp. 100 ribu, uang tunai dikeluarkan jika jumlah yang cukup tersedia di rekeningnya dan kartunya dikeluarkan; jika tidak, kartunya dikeluarkan tanpa ada uang tunai yang dikeluarkan bersama dengan tampilan pesan tentang posisi dana yang tidak mencukupi di rekeningnya.
39. Identifikasi kebutuhan fungsional dan non-fungsional dalam uraian masalah berikut dan dokumentasikan. Perangkat lunak jam kosmopolitan akan dikembangkan yang menampilkan hingga 6 jam dengan nama kota dan waktu setempat. Jam harus dirancang secara estetis. Perangkat lunak harus memungkinkan pengguna untuk mengubah nama kota mana pun dan mengubah pembacaan waktu setiap jam dengan mengetikkan c (untuk konfigurasi) pada jam mana pun. Pengguna juga harus dapat beralih antara jam digital dan tampilan jam analog dengan mengetikkan d (untuk digital) atau a (untuk analog) pada tampilan jam. Setelah implementasi yang berdiri sendiri bekerja, versi web harus dikembangkan yang dapat diunduh pada browser sebagai applet dan dijalankan. Jam seharusnya hanya menggunakan siklus idle di komputer yang dijalankannya.
40. Apa yang Anda pahami dengan inkonsistensi, anomali, dan ketidaklengkapan dalam dokumen SRS. Identifikasi inkonsistensi, anomali, dan ketidaklengkapan dalam persyaratan berikut dari perangkat lunak otomatisasi aktivitas akademik dari lembaga pendidikan: "Kinerja semester setiap mahasiswa dihitung sebagai kinerja akademik rata-rata untuk semester tersebut. Wali dari semua mahasiswa yang memiliki catatan kinerja buruk di semester dikirimi surat yang menginformasikan tentang kinerja lingkungan yang buruk dan mengisyaratkan bahwa pengulangan kinerja yang buruk di semester berikutnya dapat menyebabkan pengusiran. Kegiatan ekstrakurikuler seorang

mahasiswa juga dinilai dan dijadikan pertimbangan untuk penentuan prestasi semester."

41. Identifikasi inkonsistensi, anomali, dan ketidaklengkapan yang ada dalam persyaratan berikut yang dikumpulkan dengan mewawancarai pegawai departemen CSE untuk mengembangkan perangkat lunak otomatisasi akademik (AAS): "CGPA setiap mahasiswa dihitung sebagai kinerja rata-rata untuk semester. Orang tua dari semua mahasiswa yang memiliki kinerja buruk dikirimi surat yang menginformasikan tentang kinerja yang buruk di lingkungan mereka dan dengan permintaan untuk menyampaikan peringatan kepada mahasiswa bahwa kinerja yang buruk tidak boleh diulangi.
42. Mewakili pengambilan keputusan yang terlibat dalam persyaratan fungsional sistem otomasi perpustakaan berikut: Item Masalah: Item ketika diserahkan di konter bersama dengan kartu identitas perpustakaan, pertama-tama ditentukan apakah anggota telah melebihi kuotanya. Jika dia telah melebihi kuotanya, maka tidak ada barang yang bisa dikeluarkan untuknya. Jika item yang diminta adalah jurnal, maka diterbitkan hanya untuk dua hari. Jika itu buku, maka diperiksa apakah itu buku referensi. Buku referensi tidak dapat dikeluarkan. Jika itu bukan buku referensi, ditentukan apakah ada yang memesannya. Buku yang dipesan tidak dapat diterbitkan. Jika permintaan penerbitan buku anggota memenuhi semua kriteria yang disebutkan, maka buku tersebut dikeluarkan untuk anggota selama satu bulan, entri yang sesuai dibuat di rekening anggota dan slip penerbitan dicetak.
43. Misalkan Anda ingin mengembangkan perangkat lunak pengolah kata yang memiliki fitur yang mirip dengan Microsoft Word. Kembangkan dokumen SRS untuk perangkat lunak pengolah kata ini.

BAB 5
DESAIN SOFTRWARE

Selama fase desain sebubah software, dokumen desain diproduksi hrus didasarkan pada kebutuhan pelanggan sebagaimana didokumentasikan dalam dokumen SRS. Kita dapat menyatakan tujuan utama dari fase desain. Kegiatan yang dilakukan selama tahap desain (disebut sebagai proses desain) mengubah dokumen SRS menjadi dokumen desain. Pandangan proses desain ini telah ditunjukkan secara skematis pada Gambar 5.1. Seperti terlihat pada Gambar 5.1, proses desain dimulai dengan menggunakan dokumen SRS dan diakhiri dengan pembuatan dokumen desain. Dokumen desain yang dihasilkan pada akhir fase desain harus dapat diimplementasikan menggunakan bahasa pemrograman pada fase (pengkodean) berikutnya.



Gambar 5.1 Proses desain.

5.1 TINJAUAN PROSES DESAIN

Proses desain pada dasarnya mengubah dokumen SRS menjadi dokumen desain. Pada bagian ini kita akan membahas beberapa masalah penting yang terkait dengan proses desain.

Hasil dari Proses Desain

Item berikut dirancang dan didokumentasikan selama fase desain.

Modul yang berbeda diperlukan: Modul yang berbeda dalam solusi harus diidentifikasi dengan jelas. Setiap modul adalah kumpulan fungsi dan data yang dibagikan oleh fungsi modul. Setiap modul harus menyelesaikan beberapa tugas yang terdefinisi dengan baik di luar tanggung jawab keseluruhan perangkat lunak. Setiap modul harus diberi nama sesuai dengan tugas yang dilakukannya. Misalnya, dalam perangkat lunak otomasi akademik, modul yang terdiri dari fungsi dan data yang diperlukan untuk menyelesaikan tugas pendaftaran mahasiswa harus diberi nama menangani pendaftaran siswa.

Hubungan kontrol antar modul: Hubungan kontrol antara dua modul pada dasarnya muncul karena pemanggilan fungsi di kedua modul. Hubungan kontrol yang ada di antara berbagai modul harus diidentifikasi dalam dokumen desain.

Antarmuka antara modul yang berbeda: Antarmuka antara dua modul mengidentifikasi item data yang tepat yang dipertukarkan antara dua modul ketika satu modul memanggil fungsi dari modul lainnya.

Struktur data masing-masing modul: Setiap modul biasanya menyimpan beberapa data yang perlu dibagikan oleh fungsi modul untuk menyelesaikan tanggung jawab keseluruhan modul. Struktur data yang sesuai untuk menyimpan dan mengelola data modul perlu dirancang dan didokumentasikan dengan baik.

Algoritma yang diperlukan untuk mengimplementasikan modul individu: Setiap fungsi dalam modul biasanya melakukan beberapa aktivitas pemrosesan. Algoritme yang

diperlukan untuk menyelesaikan aktivitas pemrosesan berbagai modul perlu dirancang dan didokumentasikan dengan hati-hati dengan pertimbangan yang diberikan pada keakuratan hasil, kompleksitas ruang dan waktu.

Dimulai dengan dokumen SRS (seperti yang ditunjukkan pada Gambar 5.1), dokumen desain dihasilkan melalui iterasi melalui serangkaian langkah yang akan kita bahas dalam bab ini dan tiga bab berikutnya. Dokumen desain ditinjau oleh anggota tim pengembangan untuk memastikan bahwa solusi desain sesuai dengan spesifikasi persyaratan.

Klasifikasi Kegiatan Desain

Sebuah desain perangkat lunak yang baik jarang diwujudkan dengan menggunakan prosedur satu langkah, melainkan memerlukan iterasi atas serangkaian langkah yang disebut kegiatan desain. Mari kita mengklasifikasikan kegiatan desain terlebih dahulu sebelum membahasnya secara rinci. Tergantung pada urutan di mana berbagai kegiatan desain dilakukan, kita dapat secara luas mengklasifikasikannya menjadi dua tahap penting.

- Desain awal (atau tingkat tinggi), dan
- Desain yang rinci.

Arti dan ruang lingkup dari dua tahap ini dapat sangat bervariasi dari satu metodologi desain ke metodologi desain lainnya. Namun, untuk pendekatan desain berorientasi fungsi tradisional, adalah mungkin untuk mendefinisikan tujuan dari desain tingkat tinggi sebagai berikut: Melalui desain tingkat tinggi, masalah didekomposisi menjadi satu set modul. Hubungan kontrol antar modul diidentifikasi, dan juga antarmuka di antara berbagai modul diidentifikasi.

Hasil dari desain tingkat tinggi disebut struktur program atau rancangan roftware. Desain tingkat tinggi adalah langkah penting dalam desain keseluruhan perangkat lunak. Ketika desain tingkat tinggi selesai, masalahnya seharusnya didekomposisi menjadi banyak modul kecil yang berfungsi independen yang kohesif, memiliki sambungan rendah di antara mereka sendiri, dan diatur dalam hierarki. Banyak jenis notasi yang berbeda telah digunakan untuk mewakili desain tingkat tinggi. Sebuah notasi yang banyak digunakan untuk pengembangan prosedural adalah diagram seperti pohon yang disebut diagram struktur. Teknik representasi desain populer lainnya yang disebut UML yang digunakan untuk mendokumentasikan desain berorientasi objek, melibatkan pengembangan beberapa jenis diagram untuk mendokumentasikan desain berorientasi objek dari suatu sistem. Meskipun notasi lain seperti diagram Jackson [1975] atau diagram Warnier-Orr [1977, 1981] tersedia untuk mendokumentasikan desain perangkat lunak. Setelah desain tingkat tinggi selesai, desain rinci dilakukan. Selama desain rinci, setiap modul diperiksa dengan cermat untuk merancang struktur data dan algoritmenya.

Hasil dari tahap desain detail biasanya didokumentasikan dalam bentuk dokumen spesifikasi modul (MSPEC). Setelah desain tingkat tinggi selesai, masalahnya akan didekomposisi menjadi modul kecil, dan struktur data dan algoritma yang akan digunakan dijelaskan menggunakan MSPEC dan dapat dengan mudah dipahami oleh programmer untuk memulai pengkodean. Dalam teks ini, kita tidak membahas MSPEC dan membatasi perhatian kita pada desain tingkat tinggi saja.

Klasifikasi Metodologi Desain

Kegiatan desain sangat bervariasi berdasarkan metodologi desain khusus yang digunakan. Sejumlah besar metodologi desain perangkat lunak tersedia. Kita dapat secara kasar mengklasifikasikan metodologi ini ke dalam pendekatan prosedural dan berorientasi objek. Kedua pendekatan ini adalah dua paradigma desain yang berbeda secara fundamental. Dalam bab ini, kita akan membahas karakteristik penting dari dua pendekatan desain

mendasar ini. Selama tiga bab berikutnya, kita akan mempelajari kedua pendekatan ini secara rinci.

Apakah teknik desain menghasilkan solusi yang unik?

Bahkan saat menggunakan metodologi desain yang sama, desainer yang berbeda biasanya sampai pada solusi desain yang sangat berbeda. Alasannya adalah bahwa teknik desain sering mengharuskan desainer untuk membuat banyak keputusan subjektif dan bekerja di luar kompromi untuk tujuan yang kontradiktif. Akibatnya, mungkin saja perancang yang sama dapat mengerjakan banyak solusi berbeda untuk masalah yang sama. Oleh karena itu, memperoleh desain yang baik akan melibatkan mencoba beberapa alternatif (atau solusi kandidat) dan memilih yang terbaik. Namun, pertanyaan mendasar yang muncul pada saat ini adalah—bagaimana membedakan solusi desain yang unggul dari yang lebih rendah? Kecuali kita tahu apa itu desain perangkat lunak yang baik dan bagaimana membedakan solusi desain yang unggul dari yang lebih rendah, kita tidak mungkin mendesainnya.

Analisis versus desain

Kegiatan analisis dan desain berbeda dalam tujuan dan ruang lingkup. Tujuan dari setiap teknik analisis adalah untuk menguraikan kebutuhan pelanggan melalui pemikiran yang cermat dan pada saat yang sama secara sadar menghindari pengambilan keputusan apa pun mengenai cara yang tepat untuk mengimplementasikan sistem. Hasil analisis bersifat umum dan tidak mempertimbangkan implementasi atau masalah yang terkait dengan platform tertentu. Model analisis biasanya didokumentasikan menggunakan beberapa formalisme grafis. Dalam hal pendekatan berorientasi fungsi yang akan kita bahas, model analisis akan didokumentasikan menggunakan diagram aliran data (DFD), sedangkan desain akan didokumentasikan menggunakan bagan struktur. Di sisi lain, untuk pendekatan berorientasi objek, baik model desain maupun model analisis akan didokumentasikan menggunakan bahasa pemodelan terpadu (UML). Model analisis biasanya akan sangat sulit diimplementasikan menggunakan bahasa pemrograman.

Model desain diperoleh dari model analisis melalui transformasi melalui serangkaian langkah. Berbeda dengan model analisis, model desain mencerminkan beberapa keputusan yang diambil mengenai cara yang tepat untuk mengimplementasikan sistem. Model desain harus cukup detail agar mudah diimplementasikan menggunakan bahasa pemrograman.

5.2 BAGAIMANA KARAKTERISASI DESAIN PERANGKAT LUNAK YANG BAIK?

Menghasilkan karakterisasi yang akurat dari desain perangkat lunak yang baik yang akan bertahan di berbagai domain masalah tentu tidak mudah. Faktanya, definisi desain perangkat lunak yang "baik" dapat bervariasi tergantung pada aplikasi persis yang sedang dirancang. Misalnya, "ukuran memori yang digunakan oleh suatu program" mungkin menjadi masalah penting untuk Mengkarakterisasi solusi yang baik untuk pengembangan perangkat lunak yang disematkan—karena aplikasi yang disematkan sering kali diperlukan untuk bekerja di bawah ukuran memori yang sangat terbatas karena pertimbangan biaya, ruang, atau konsumsi daya. Untuk aplikasi yang disematkan, faktor-faktor seperti pemahaman desain mungkin tidak diperhatikan saat menilai kebaikan desain. Jadi untuk aplikasi yang disematkan, seseorang dapat mengorbankan pemahaman desain untuk mencapai kekompakan kode. Demikian pula, biasanya tidak benar bahwa kriteria yang sangat penting untuk beberapa aplikasi, hampir sepenuhnya diabaikan untuk aplikasi lain. Oleh karena itu jelas bahwa kriteria yang digunakan untuk menilai solusi desain dapat sangat bervariasi di berbagai jenis aplikasi. Tidak hanya kriteria yang digunakan untuk menilai solusi desain tergantung pada aplikasi yang tepat yang sedang dirancang, tetapi untuk membuat masalah menjadi lebih buruk, tidak ada kesepakatan umum antara insinyur perangkat lunak dan peneliti tentang kriteria yang tepat

untuk digunakan untuk menilai desain bahkan untuk tujuan tertentu. kategori aplikasi. Namun, sebagian besar peneliti dan insinyur perangkat lunak menyetujui beberapa karakteristik yang diinginkan yang harus dimiliki oleh setiap desain perangkat lunak yang baik untuk aplikasi umum. Karakteristik ini tercantum di bawah ini:

- **Kebenaran**: Desain yang baik pertama-tama harus benar. Artinya, ia harus mengimplementasikan semua fungsi sistem dengan benar.
- **Dapat dimengerti**: Desain yang baik harus mudah dimengerti. Kecuali solusi desain mudah dimengerti, akan sulit untuk menerapkan dan memeliharanya.
- **Efisiensi:** Sebuah solusi desain yang baik harus secara memadai mengatasi masalah sumber daya, waktu, dan optimasi biaya.
- **Pemeliharaan**: Desain yang baik harus mudah diubah. Ini adalah persyaratan penting, karena permintaan perubahan biasanya terus datang dari pelanggan bahkan setelah rilis produk.

Pemahaman Desain: Perhatian Utama

Saat melakukan desain untuk masalah tertentu, asumsikan bahwa kita telah sampai pada sejumlah besar solusi desain dan perlu memilih yang terbaik. Jelas semua desain yang salah harus dibuang terlebih dahulu. Dari solusi desain yang benar, bagaimana kita bisa mengidentifikasi yang terbaik? Mengingat bahwa kita hanya memilih dari solusi desain yang benar, pemahaman solusi desain mungkin merupakan masalah yang paling penting untuk dipertimbangkan saat menilai kebaikan desain.

Ingat kembali dari diskusi kita di Bab 1 bahwa desain yang baik seharusnya membantu mengatasi keterbatasan kognitif manusia yang muncul karena keterbatasan memori jangka pendek. Masalah besar menguasai pikiran manusia, dan desain yang buruk akan memperburuk masalah. Kecuali solusi desain mudah dimengerti, itu bisa menyebabkan implementasi memiliki sejumlah besar cacat dan pada saat yang sama sangat mendorong biaya pengembangan. Oleh karena itu, solusi desain yang baik harus sederhana dan mudah dimengerti. Sebuah desain yang mudah dipahami juga mudah untuk dikembangkan dan dipelihara. Sebuah desain yang kompleks akan menyebabkan sangat meningkatkan biaya siklus hidup. Kecuali sebuah desain mudah dimengerti, itu akan membutuhkan upaya yang luar biasa untuk mengimplementasikan, menguji, men-debug, dan memeliharanya. Sekitar 60 persen dari total upaya dalam siklus hidup produk tipikal dihabiskan untuk pemeliharaan. Jika perangkat lunak tidak mudah dipahami, tidak hanya akan menyebabkan peningkatan biaya pengembangan, upaya yang diperlukan untuk memelihara produk juga akan meningkat berlipat ganda. Selain itu, solusi desain yang sulit dipahami akan menghasilkan program yang penuh dengan bug dan tidak dapat diandalkan. Ingat kembali bahwa kita telah membahas di Bab 1 bahwa pemahaman solusi desain dapat ditingkatkan melalui aplikasi cerdas dari prinsip-prinsip abstraksi dan dekomposisi.

Desain yang dapat dimengerti adalah modular dan berlapis

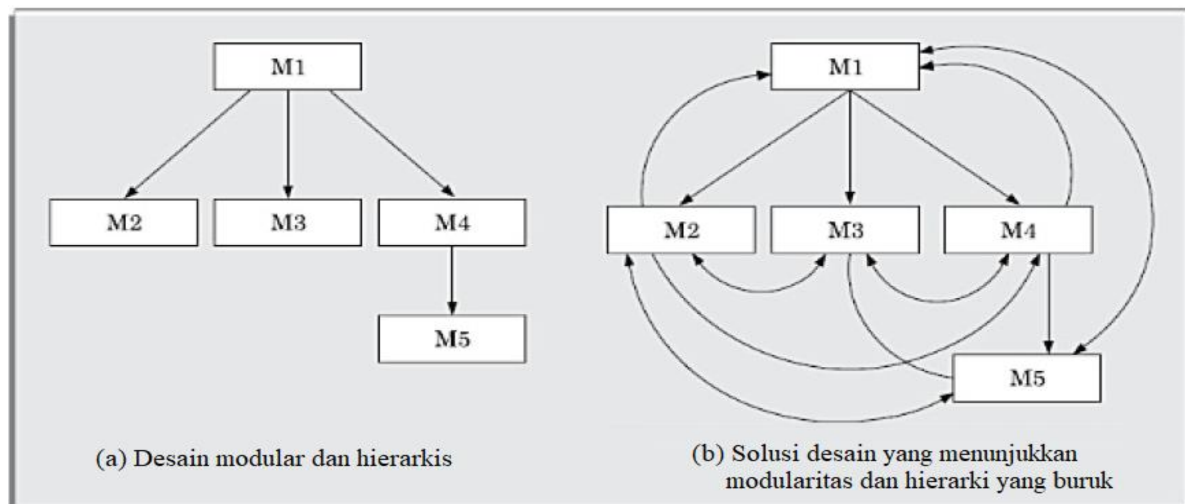
Bagaimana pemahaman dari dua desain yang berbeda dapat dibandingkan, sehingga kita dapat memilih yang lebih baik? Untuk dapat membandingkan pemahaman dua solusi desain, setidaknya kita harus memiliki pemahaman tentang fitur umum yang harus dimiliki oleh desain yang mudah dipahami. Solusi desain harus memiliki karakteristik berikut agar mudah dipahami:

- Ini harus menetapkan nama yang konsisten dan bermakna untuk berbagai komponen desain.
- Itu harus menggunakan prinsip-prinsip dekomposisi dan abstraksi dalam ukuran yang baik untuk menyederhanakan desain.

Konsep penting di balik prinsip abstraksi dan prinsip dekomposisi di Bab 1. Namun, bagaimana prinsip abstraksi dan dekomposisi digunakan dalam mencapai solusi desain? Kedua prinsip ini dimanfaatkan oleh metodologi desain untuk membuat desain modular dan berlapis. (Meskipun ada juga beberapa bentuk lain di mana prinsip abstraksi dan dekomposisi dapat digunakan dalam solusi desain, kita akan membahasnya nanti). Sekarang kita dapat mendefinisikan karakteristik desain yang mudah dipahami sebagai berikut: Solusi desain dapat dimengerti, jika bersifat modular dan modul disusun dalam lapisan yang berbeda. Solusi desain harus modular dan berlapis agar dapat dimengerti. Konsep modularitas dan pelapisan modul:

Modularitas

Desain modular adalah dekomposisi masalah yang efektif. Ini adalah karakteristik dasar dari setiap solusi desain yang baik. Desain modular, dengan kata sederhana, menyiratkan bahwa masalah telah didekomposisi menjadi satu set modul yang hanya memiliki interaksi terbatas satu sama lain. Dekomposisi masalah ke dalam modul memfasilitasi pengambilan keuntungan dari prinsip membagi dan menaklukkan. Jika modul yang berbeda tidak memiliki interaksi atau sedikit interaksi satu sama lain, maka setiap modul dapat dipahami secara terpisah. Ini sangat mengurangi kompleksitas yang dirasakan dari solusi desain. Untuk memahami mengapa demikian, ingatlah bahwa mungkin sangat sulit untuk mematahkan seikat tongkat yang telah diikat menjadi satu, tetapi sangat mudah untuk mematahkan tongkat satu per satu.



Gambar 5.2 Dua solusi desain untuk masalah yang sama

Tidak sulit untuk menyatakan bahwa modularitas merupakan karakteristik penting dari solusi desain yang baik. Tetapi, bahkan dengan ini, bagaimana kita dapat membandingkan modularitas dari dua solusi desain alternatif? Dari pemeriksaan struktur modul, paling tidak mungkin untuk secara intuitif membentuk ide desain mana yang lebih modular Misalnya, pertimbangkan dua solusi desain alternatif untuk masalah yang direpresentasikan pada Gambar 5.2, di mana modul M1 , M2 dll telah digambar sebagai persegi panjang. Pemanggilan modul oleh modul lain telah ditunjukkan sebagai panah. Dapat dengan mudah dilihat bahwa solusi desain Gambar 5.2(a) akan lebih mudah dipahami karena interaksi antar modul yang berbeda rendah. Tapi, bisakah kita mengukur modularitas solusi desain secara kuantitatif? Kecuali kita dapat mengukur secara kuantitatif modularitas solusi desain, akan sulit untuk mengatakan solusi desain mana yang lebih modular daripada yang lain. Sayangnya, tidak ada metrik kuantitatif yang tersedia untuk secara langsung mengukur modularitas desain. Namun,

kita dapat secara kuantitatif mengkarakterisasi modularitas solusi desain berdasarkan kohesi dan kopling yang ada dalam desain.

Solusi desain dikatakan sangat modular, jika modul yang berbeda dalam solusi memiliki kohesi tinggi dan sambungan antar-modulnya rendah. Sebuah desain perangkat lunak dengan kohesi tinggi dan kopling rendah antara modul adalah dekomposisi masalah efektif yang kita bahas di Bab 1. Desain seperti itu akan mengarah pada peningkatan produktivitas selama pengembangan program dengan menurunkan kompleksitas masalah yang dirasakan.

Berdasarkan klasifikasi ini, kita akan dapat dengan mudah menilai kohesi dan kopling yang ada dalam solusi desain. Dari pengetahuan tentang kohesi dan kopling dalam desain, kita dapat membentuk pendapat kita sendiri tentang modularitas solusi desain. Kita dapat mendefinisikan konsep kohesi dan kopling dan berbagai kelas kohesi. Sekarang mari kita bahas karakteristik penting lainnya dari solusi desain yang baik—desain berlapis.

Desain berlapis

Sebuah desain berlapis adalah salah satu di mana ketika hubungan panggilan antara modul yang berbeda direpresentasikan secara grafis, itu akan menghasilkan diagram seperti pohon dengan layering yang jelas. Dalam solusi desain berlapis, modul disusun dalam hierarki lapisan. Sebuah modul hanya dapat menjalankan fungsi dari modul-modul di lapisan tepat di bawahnya. Modul lapisan yang lebih tinggi dapat dianggap serupa dengan manajer yang memanggil (memesan) modul lapisan bawah untuk menyelesaikan tugas tertentu. Desain berlapis dapat dianggap menerapkan abstraksi kontrol, karena modul di lapisan bawah tidak mengetahui (tentang cara memanggil) modul lapisan yang lebih tinggi.

Sebuah desain berlapis dapat membuat solusi desain mudah dimengerti, karena untuk memahami cara kerja modul, seseorang harus memahami bagaimana modul lapisan bawah langsung bekerja tanpa harus khawatir tentang fungsi modul lapisan atas. Ketika kegagalan terdeteksi saat menjalankan modul, jelas bahwa modul di bawahnya mungkin bisa menjadi sumber kesalahan. Ini sangat menyederhanakan debugging karena seseorang hanya perlu berkonsentrasi pada beberapa modul untuk mendeteksi kesalahan.

5.3 KOHESI DAN COUPLING

Sejauh ini kita telah membahas bahwa dekomposisi masalah yang efektif merupakan karakteristik penting dari desain yang baik. Dekomposisi modul yang baik ditunjukkan melalui kohesi yang tinggi dari masing-masing modul dan kopling modul yang rendah satu sama lain. Sekarang mari kita definisikan apa yang dimaksud dengan kohesi dan kopling. Kohesi adalah ukuran kekuatan fungsional sebuah modul, sedangkan kopling antara dua modul adalah ukuran tingkat interaksi (atau saling ketergantungan) antara dua modul. Pada bagian ini, pertama-tama kita menguraikan konsep kohesi dan kopling. Selanjutnya, kita membahas klasifikasi kohesi dan kopling. Kopling: Secara intuitif, kita dapat memikirkan kopling sebagai berikut. Dua modul dikatakan sangat berpasangan, jika salah satu dari dua situasi berikut muncul:

- Jika pemanggilan fungsi antara dua modul melibatkan melewatkan sebagian besar data bersama, modul-modul tersebut digabungkan secara erat.
- Jika interaksi terjadi melalui beberapa data bersama, maka mereka sangat berpasangan.

Jika dua modul tidak berinteraksi satu sama lain sama sekali atau paling baik berinteraksi dengan tidak melewatkan data atau hanya beberapa item data primitif, mereka dikatakan memiliki kopling rendah.

Kohesi: Untuk memahami kohesi, mari kita pahami analogi terlebih dahulu. Misalkan Anda mendengarkan ceramah oleh beberapa pembicara. Anda akan menyebut pidato menjadi kohesif, jika semua kalimat pidato memainkan peran dalam memberikan pidato satu tema dan fokus. Sekarang, kita dapat memperluas ini ke modul dalam solusi desain. Ketika fungsi modul bekerja sama satu sama lain untuk melakukan satu tujuan, maka modul memiliki kohesi yang baik. Jika fungsi modul melakukan hal yang sangat berbeda dan tidak bekerja sama satu sama lain untuk melakukan satu pekerjaan, maka modul memiliki kohesi yang sangat buruk.

Kemandirian fungsional

Dengan istilah independensi fungsional, modul melakukan satu tugas dan membutuhkan sedikit interaksi dengan modul lain. Sebuah modul yang sangat kohesif dan juga memiliki kopling rendah dengan modul lain dikatakan secara fungsional independen dari modul lainnya. Kemandirian fungsional adalah kunci untuk setiap desain yang baik terutama karena keuntungan-keuntungan berikut yang ditawarkannya:

Isolasi kesalahan: Setiap kali ada kesalahan dalam modul, independensi fungsional mengurangi kemungkinan kesalahan menyebar ke modul lain. Alasan di balik ini adalah bahwa jika sebuah modul secara fungsional independen, interaksinya dengan modul lain rendah. Oleh karena itu, kesalahan yang ada pada modul sangat kecil kemungkinannya mempengaruhi fungsi modul lainnya.

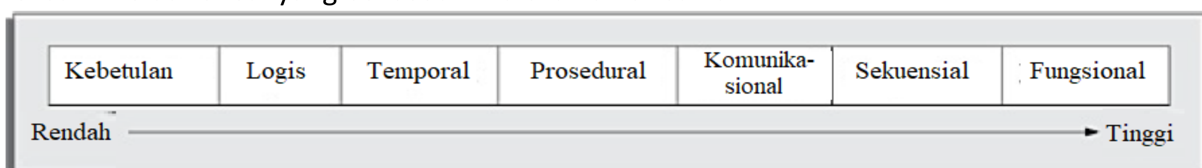
Selanjutnya, setelah kegagalan terdeteksi, isolasi kesalahan membuatnya sangat mudah untuk menemukan kesalahan. Di sisi lain, ketika modul tidak independen secara fungsional, setelah kegagalan terdeteksi dalam fungsi yang disediakan oleh modul, kesalahan dapat berpotensi terjadi di salah satu dari sejumlah besar modul dan disebarkan ke fungsi modul.

Lingkup penggunaan kembali: Penggunaan kembali modul untuk pengembangan aplikasi lain menjadi lebih mudah. Alasan untuk ini adalah sebagai berikut. Sebuah modul fungsional independen melakukan beberapa tugas yang terdefinisi dengan baik dan tepat dan antarmuka modul dengan modul lain sangat sedikit dan sederhana. Oleh karena itu, modul yang secara fungsional independen dapat dengan mudah dikeluarkan dan digunakan kembali dalam program yang berbeda. Di sisi lain, jika modul berinteraksi dengan beberapa modul lain atau fungsi modul melakukan tugas yang sangat berbeda, maka akan sulit untuk menggunakannya kembali. Hal ini terutama terjadi, jika modul mengakses data (atau kode) internal ke modul lain.

Dapat dimengerti: Ketika modul secara fungsional independen, kompleksitas desain sangat berkurang. Ini karena fakta bahwa modul yang berbeda dapat dipahami secara terpisah, karena modul tidak tergantung satu sama lain.

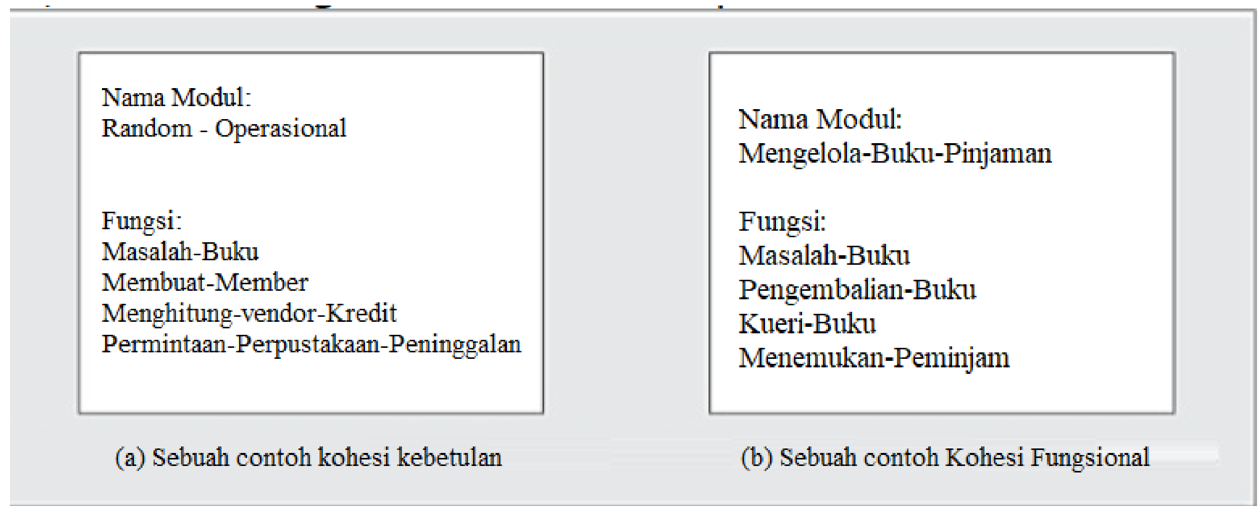
Klasifikasi Keterpaduan

Kekompakan modul adalah sejauh mana fungsi yang berbeda dari modul bekerja sama untuk bekerja menuju satu tujuan. Modul desain yang berbeda dapat memiliki derajat kebebasan yang berbeda. Namun, kelas kohesi yang berbeda yang dapat dimiliki modul digambarkan pada Gambar 5.3. Kekompakan meningkat dari kohesi kebetulan menjadi kohesi fungsional. Artinya, kebetulan adalah jenis kohesi terburuk dan fungsional adalah kohesi terbaik. Kelas kohesi yang berbeda ini diuraikan di bawah ini.



Gambar 5.3 Klasifikasi kohesi.

Kohesi kebetulan: Sebuah modul dikatakan memiliki kohesi kebetulan, jika melakukan serangkaian tugas yang berhubungan satu sama lain sangat longgar, jika sama sekali. Dalam hal ini, kita dapat mengatakan bahwa modul berisi kumpulan fungsi acak. Sangat mungkin bahwa fungsi-fungsi tersebut telah ditempatkan dalam modul karena kebetulan murni daripada melalui beberapa pemikiran atau desain. Desain yang dibuat oleh programmer pemula sering memiliki kategori kohesi ini, karena mereka sering menggabungkan fungsi ke modul dengan agak sewenang-wenang. Contoh modul dengan kohesi kebetulan ditunjukkan pada Gambar 5.4(a). Perhatikan bahwa fungsi modul yang berbeda melakukan aktivitas yang sangat berbeda dan tidak terkait mulai dari penerbitan buku perpustakaan hingga pembuatan catatan anggota perpustakaan di satu sisi, dan penanganan permintaan cuti pustakawan di sisi lain.



Gambar 5.4 Contoh kohesi

Kohesi logis: Sebuah modul dikatakan kohesif secara logis, jika semua elemen modul melakukan operasi serupa, seperti penanganan kesalahan, input data, output data, dll. Sebagai contoh kohesi logis, pertimbangkan modul yang berisi satu set print berfungsi untuk menghasilkan berbagai jenis laporan keluaran seperti grade sheet, slip gaji, laporan tahunan, dll.

Kohesi temporal: Ketika sebuah modul berisi fungsi-fungsi yang terkait dengan fakta bahwa fungsi-fungsi ini dijalankan dalam rentang waktu yang sama, maka modul tersebut dikatakan memiliki kohesi temporal. Sebagai contoh, perhatikan situasi berikut. Saat komputer di-boot, beberapa fungsi perlu dilakukan. Ini termasuk inisialisasi memori dan perangkat, memuat sistem operasi, dll. Ketika satu modul melakukan semua tugas ini, maka modul tersebut dapat dikatakan menunjukkan kohesi temporal. Contoh lain dari modul yang memiliki kohesi temporal adalah sebagai berikut. Demikian pula, modul akan menunjukkan kohesi temporal, jika terdiri dari fungsi untuk melakukan inisialisasi, atau start-up, atau shutdown dari beberapa proses.

Kohesi prosedural: Sebuah modul dikatakan memiliki kohesi prosedural, jika kumpulan fungsi modul dieksekusi satu demi satu, meskipun fungsi-fungsi ini dapat bekerja untuk tujuan yang sama sekali berbeda dan beroperasi pada data yang sangat berbeda. Pertimbangkan aktivitas yang terkait dengan pemrosesan pesanan di rumah perdagangan. Fungsi login(), place-order(), check-order(), printbill(), place-order-on-vendor(), update-inventory(), dan logout() semuanya melakukan hal yang berbeda dan beroperasi pada data yang berbeda. Namun, mereka biasanya dieksekusi satu demi satu selama pemrosesan pesanan biasa oleh petugas penjualan.

Kohesi komunikasi: Sebuah modul dikatakan memiliki kohesi komunikasi, jika semua fungsi modul mengacu atau memperbarui struktur data yang sama. Sebagai contoh kohesi prosedural, pertimbangkan sebuah modul bernama student di mana fungsi yang berbeda dalam modul seperti acceptStudent, enterMarks, printGradeSheet, dll. mengakses dan memanipulasi data yang disimpan dalam array bernama studentRecords yang didefinisikan di dalam modul.

Kohesi berurutan: Sebuah modul dikatakan memiliki kohesi berurutan, jika fungsi modul yang berbeda dijalankan secara berurutan, dan output dari satu fungsi dimasukkan ke fungsi berikutnya dalam urutan tersebut. Sebagai contoh perhatikan situasi berikut. Di toko online pertimbangkan bahwa setelah pelanggan meminta beberapa item, pertama-tama ditentukan apakah item tersebut tersedia. Dalam hal ini, jika fungsi create-order(), check-item-availability(), placeorder-on-vendor() ditempatkan dalam satu modul, maka modul akan menunjukkan kohesi berurutan. Perhatikan bahwa fungsi create-order() membuat pesanan yang diproses oleh fungsi check-item-availability() (apakah item tersedia dalam jumlah yang diperlukan dalam inventaris) dimasukkan ke place-order-on-vendor( ).

Kohesi fungsional: Sebuah modul dikatakan memiliki kohesi fungsional, jika fungsi yang berbeda dari modul bekerja sama untuk menyelesaikan satu tugas. Misalnya, modul yang berisi semua fungsi yang diperlukan untuk mengelola daftar gaji karyawan menampilkan kohesi fungsional. Dalam hal ini, semua fungsi modul (misalnya, computeOvertime(), computeWorkHours(), computeDeductions(), dll.) bekerja sama untuk menghasilkan slip gaji karyawan. Contoh lain dari modul yang memiliki kohesi fungsional telah ditunjukkan pada Gambar 5.4(b). Dalam contoh ini, fungsi issue-book(), return-book(), query-book(), dan find-borrower(), bersama-sama mengelola semua aktivitas yang berkaitan dengan peminjaman buku. Ketika sebuah modul memiliki kohesi fungsional, maka kita harus dapat menggambarkan apa yang dilakukan modul hanya dengan menggunakan satu kalimat sederhana. Sebagai contoh, untuk modul pada Gambar 5.4(a), kita dapat menggambarkan keseluruhan tanggung jawab modul dengan mengatakan “Ini mengelola prosedur peminjaman buku di perpustakaan.”

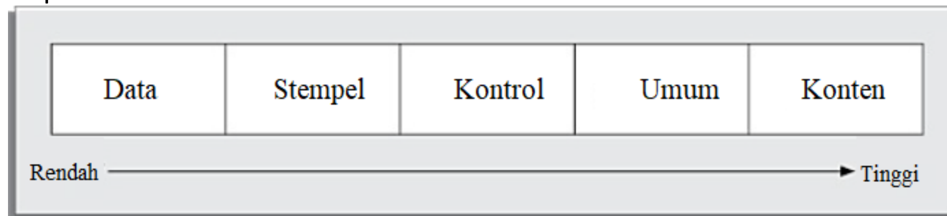
Cara sederhana untuk menentukan kekompakan dari setiap modul yang diberikan adalah sebagai berikut. Pertama periksa apa yang dilakukan fungsi modul. Kemudian, coba tuliskan sebuah kalimat untuk menggambarkan keseluruhan pekerjaan yang dilakukan oleh modul. Jika Anda memerlukan kalimat majemuk untuk menggambarkan fungsionalitas modul, maka modul tersebut memiliki kohesi sekuensial atau komunikasi. Jika Anda membutuhkan kata-kata seperti "pertama", "berikutnya", "setelah", "kemudian", dll., maka itu memiliki kohesi berurutan atau temporal. Jika perlu kata-kata seperti "inisialisasi", "setup", "shut down”, dll., untuk mendefinisikan fungsinya, maka ia memiliki kohesi temporal.

Kita sekarang dapat melakukan pengamatan berikut. Modul kohesif adalah modul di mana fungsi-fungsi berinteraksi di antara mereka sendiri secara intens untuk mencapai satu tujuan. Akibatnya, jika salah satu fungsi ini dihapus ke modul yang berbeda, kopling akan meningkat karena fungsi sekarang akan berinteraksi di dua modul yang berbeda.

Klasifikasi Kopling

Kopling antara dua modul menunjukkan tingkat saling ketergantungan di antara mereka. Secara intuitif, jika dua modul bertukar data dalam jumlah besar, maka mereka sangat saling bergantung atau digabungkan. Kita dapat menyatakan konsep ini secara bergantian sebagai berikut. Tingkat kopling antara dua modul tergantung pada kompleksitas antarmuka mereka. Kompleksitas antarmuka ditentukan berdasarkan jumlah parameter dan kompleksitas parameter yang dipertukarkan saat satu modul memanggil fungsi modul lainnya. Mari kita sekarang mengklasifikasikan berbagai jenis kopling yang ada di antara dua modul. Di

antara dua modul yang berinteraksi, salah satu dari lima jenis kopling berikut dapat ada. Jenis kopling yang berbeda ini, dalam urutan tingkat keparahannya yang meningkat juga telah ditunjukkan pada Gambar 5.5.



Gambar 5.5 Klasifikasi kopling.

Kopling data: Dua modul adalah data yang digabungkan, jika mereka berkomunikasi menggunakan item data dasar yang dilewatkan sebagai parameter di antara keduanya, mis. integer, float, karakter, dll. Item data ini harus terkait dengan masalah dan tidak digunakan untuk tujuan kontrol.

- **Kopling stempel**: Dua modul digabungkan stempel, jika mereka berkomunikasi menggunakan item data komposit seperti catatan dalam PASCAL atau struktur dalam C.
- **Kopling kontrol**: Kopling kontrol ada di antara dua modul, jika data dari satu modul digunakan untuk mengarahkan urutan eksekusi instruksi di modul lain. Contoh kopling kontrol adalah flag yang diatur dalam satu modul dan diuji di modul lain.
- **Kopling umum**: Dua modul digabungkan secara umum, jika mereka berbagi beberapa item data global.
- **Penggabungan konten**: Penggabungan konten ada di antara dua modul, jika mereka berbagi kode. Artinya, lompatan dari satu modul ke kode modul lain dapat terjadi. Bahasa pemrograman tingkat tinggi modern seperti C tidak mendukung lompatan lintas modul seperti itu.

Berbagai jenis kopling ditunjukkan secara skematis pada Gambar 5.5. Tingkat penggandengan meningkat dari penyambungan data ke penyambungan konten. Kopling yang tinggi di antara modul tidak hanya membuat solusi desain sulit untuk dipahami dan dipelihara, tetapi juga meningkatkan upaya pengembangan dan juga membuat modul ini sangat sulit untuk dikembangkan secara mandiri oleh anggota tim yang berbeda.

5.4 PENYUSUNAN MODUL BERLAPIS

Hirarki kontrol mewakili organisasi komponen program dalam hal hubungan panggilan mereka. Dengan demikian kita dapat mengatakan bahwa hierarki kontrol suatu desain ditentukan oleh urutan di mana modul yang berbeda memanggil satu sama lain. Banyak jenis notasi yang berbeda telah digunakan untuk mewakili hierarki kontrol. Notasi yang paling umum adalah diagram seperti pohon yang dikenal sebagai diagram struktur yang akan kita pelajari secara rinci di Bab 6. Namun, notasi lain seperti Warnier-Orr [1977, 1981] atau diagram Jackson [1975] juga dapat digunakan. Karena notasi Warnier-Orr dan Jackson tidak banyak digunakan saat ini.

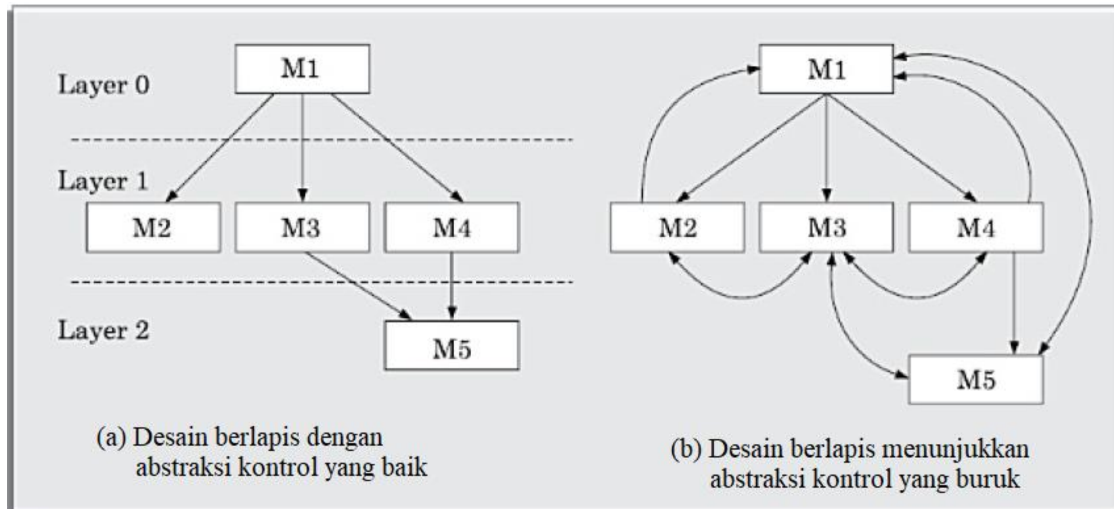
Dalam solusi desain berlapis, modul disusun menjadi beberapa lapisan berdasarkan hubungan panggilannya. Sebuah modul diperbolehkan untuk memanggil hanya modul yang berada di lapisan bawah. Artinya, modul tidak boleh memanggil modul yang berada di lapisan yang lebih tinggi atau bahkan di lapisan yang sama. Gambar 5.6(a) menunjukkan desain yang berlapis, sedangkan Gambar 5.6(b) menunjukkan desain yang tidak berlapis. Perhatikan bahwa solusi desain yang ditunjukkan pada Gambar 5.6(b), sebenarnya tidak berlapis karena

semua modul dapat dianggap berada dalam lapisan yang sama. Berikut ini, Saya akan menyatakan pentingnya desain berlapis dan selanjutnya menjelaskannya. Fitur karakteristik penting dari solusi desain yang baik adalah pelapisan modul. Desain berlapis mencapai abstraksi kontrol dan lebih mudah dipahami dan di-debug.

Dalam desain berlapis, modul paling atas dalam hierarki dapat dianggap sebagai manajer yang hanya memanggil layanan modul tingkat bawah untuk memenuhi tanggung jawabnya. Modul pada lapisan menengah menawarkan layanan ke lapisan yang lebih tinggi dengan menggunakan layanan dari modul lapisan bawah dan juga dengan melakukan beberapa pekerjaan sendiri sampai batas tertentu. Modul di lapisan paling bawah adalah modul pekerja. Ini tidak meminta layanan dari modul apa pun dan sepenuhnya menjalankan tanggung jawab mereka sendiri.

Memahami desain berlapis lebih mudah karena untuk memahami satu modul, paling-paling kita harus mempertimbangkan modul di lapisan bawah (yaitu, modul yang layanannya dipanggil). Selain itu, dalam desain berlapis kesalahan terisolasi, karena kesalahan dalam satu modul hanya dapat mempengaruhi modul lapisan yang lebih tinggi. Akibatnya, jika terjadi kegagalan modul, hanya modul di tingkat yang lebih rendah yang perlu diselidiki untuk kemungkinan kesalahan. Dengan demikian, waktu debugging berkurang secara signifikan dalam desain berlapis. Di sisi lain, jika modul yang berbeda saling memanggil secara sewenang-wenang, maka situasi ini akan sesuai dengan modul yang disusun dalam satu lapisan. Menemukan kesalahan akan sulit dan memakan waktu. Ini karena, setelah kegagalan diamati, penyebab kegagalan (yaitu kesalahan) berpotensi ada di modul apa pun, dan semua modul harus diselidiki untuk kesalahan tersebut. Berikut ini, beberapa konsep dan terminologi penting yang terkait dengan desain berlapis:

- **Modul superordinat dan subordinat**: Dalam hierarki kontrol, modul yang mengontrol modul lain dikatakan sebagai superordinat. Sebaliknya, modul yang dikendalikan oleh modul lain dikatakan berada di bawah pengontrol.
- **Visibilitas**: Sebuah modul B dikatakan terlihat oleh modul A lainnya, jika A secara langsung memanggil B. Dengan demikian, hanya modul-modul lapisan bawah yang dikatakan terlihat oleh sebuah modul.
- **Kontrol abstraksi:** Dalam desain berlapis, modul hanya boleh memanggil fungsi modul yang ada di lapisan tepat di bawahnya. Dengan kata lain, modul di lapisan yang lebih tinggi, tidak boleh terlihat (yaitu, disarikan) ke modul di lapisan bawah. Ini disebut sebagai abstraksi kontrol.
- **Kedalaman dan lebar:** Kedalaman dan lebar hierarki kontrol memberikan indikasi jumlah lapisan dan rentang kendali keseluruhan masing-masing. Untuk desain Gambar 5.6(a), kedalamannya adalah 3 dan lebarnya juga 3.
- **Fan-out:** Fan-out adalah ukuran jumlah modul yang dikontrol langsung oleh modul yang diberikan. Pada Gambar 5.6(a), fan-out modul M1 adalah 3. Sebuah desain di mana modul memiliki jumlah fan-out yang sangat tinggi bukanlah desain yang baik. Alasan untuk ini adalah bahwa fan-out yang sangat tinggi merupakan indikasi bahwa modul tidak memiliki kohesi. Modul yang memiliki fan-out besar (lebih besar dari 7) kemungkinan akan mengimplementasikan beberapa fungsi berbeda dan bukan hanya satu fungsi kohesif.
- **Fan-in:** Fan-in menunjukkan jumlah modul yang secara langsung memanggil modul yang diberikan. Fan-in tinggi mewakili penggunaan kembali kode dan secara umum, diinginkan dalam desain yang baik. Pada Gambar 5.6(a), fan-in modul M1 adalah 0, yang dari M2 adalah 1, dan dari M5 adalah 2.



Gambar 5.6 Contoh abstraksi kontrol yang baik dan buruk.

5.5 PENDEKATAN UNTUK DESAIN PERANGKAT LUNAK

Ada dua pendekatan yang berbeda secara mendasar untuk desain perangkat lunak yang digunakan saat ini — desain berorientasi fungsi, dan desain berorientasi objek. Meskipun kedua pendekatan desain ini sangat berbeda, keduanya saling melengkapi daripada teknik bersaing. Pendekatan berorientasi objek adalah teknologi yang relatif baru dan masih terus berkembang. Untuk pengembangan program besar, pendekatan berorientasi objek menjadi semakin populer karena keunggulan tertentu yang ditawarkannya. Di sisi lain, perancangan berorientasi fungsi adalah teknologi yang matang dan memiliki banyak pengikut. Fitur yang menonjol dari kedua pendekatan ini dibahas dalam sub-bagian 5.5.1 dan 5.5.2 masing-masing.

Desain berorientasi fungsi

Berikut ini adalah fitur yang menonjol dari pendekatan desain berorientasi fungsi:

Dekomposisi top-down: Sebuah sistem, untuk memulai, dipandang sebagai kotak hitam yang menyediakan layanan tertentu (juga dikenal sebagai fungsi tingkat tinggi) kepada pengguna sistem. Dalam dekomposisi top-down, mulai dari tampilan sistem tingkat tinggi, setiap fungsi tingkat tinggi secara berturut-turut disempurnakan menjadi fungsi yang lebih rinci.

Misalnya, pertimbangkan fungsi buat-anggota perpustakaan baru yang pada dasarnya membuat catatan untuk anggota baru, memberikan nomor keanggotaan unik kepadanya, dan mencetak tagihan untuk biaya keanggotaannya. Fungsi tingkat tinggi ini dapat disempurnakan menjadi subfungsi berikut:

- assign-membership-number
- buat-catatan-anggota
- tagihan cetak

Masing-masing subfungsi ini dapat dipecah menjadi subfungsi yang lebih rinci dan seterusnya.

Status sistem terpusat: Status sistem dapat didefinisikan sebagai nilai item data tertentu yang menentukan respons sistem terhadap tindakan pengguna atau peristiwa eksternal. Misalnya, kumpulan buku (yaitu apakah dipinjam oleh pengguna yang berbeda atau tersedia untuk diterbitkan) menentukan status sistem otomasi perpustakaan. Data tersebut dalam program prosedural biasanya memiliki cakupan global dan dibagi oleh banyak modul. Status sistem terpusat dan dibagi di antara fungsi-fungsi yang berbeda. Misalnya, dalam sistem manajemen perpustakaan, beberapa fungsi seperti berbagi data berikut seperti catatan anggota untuk referensi dan pembaruan:

- buat-anggota baru
- hapus-anggota
- perbarui-catatan-anggota

Sejumlah besar pendekatan desain berorientasi fungsi telah diusulkan di masa lalu. Beberapa pendekatan desain berorientasi fungsi yang mapan adalah sebagai berikut:

- Desain terstruktur oleh Constantine and Yourdon, [1979]
- Desain terstruktur Jackson oleh Jackson [1975]
- Metodologi Warnier-Orr [1977, 1981]
- Penyempurnaan bertahap oleh Wirth [1971]
- Metodologi Hatley dan Pirbhai [1987]

Desain berorientasi objek

Dalam pendekatan desain berorientasi objek (OOD), sebuah sistem dipandang sebagai terdiri dari kumpulan objek (yaitu entitas). Setiap objek diasosiasikan dengan sekumpulan fungsi yang disebut metodenya. Setiap objek berisi datanya sendiri dan bertanggung jawab untuk mengelolanya. Data internal ke suatu objek tidak dapat diakses secara langsung oleh objek lain dan hanya melalui pemanggilan metode objek. Status sistem terdesentralisasi karena tidak ada data yang dibagikan secara global dalam sistem dan data disimpan di setiap objek. Misalnya, dalam perangkat lunak otomatisasi perpustakaan, setiap anggota perpustakaan dapat menjadi objek terpisah dengan datanya sendiri dan berfungsi untuk beroperasi pada data yang disimpan. Metode yang didefinisikan untuk satu objek tidak dapat secara langsung merujuk atau mengubah data objek lain.

Paradigma desain berorientasi objek memanfaatkan secara ekstensif prinsip-prinsip abstraksi dan dekomposisi seperti yang dijelaskan di bawah ini. Objek menguraikan sistem menjadi modul fungsional independen. Objek juga dapat dianggap sebagai instance dari tipe data abstrak (ADT). Konsep ADT tidak berasal dari pendekatan berorientasi objek. Bahkan, konsep ADT banyak digunakan dalam bahasa pemrograman ADA yang diperkenalkan pada tahun 1970-an. ADT merupakan konsep penting yang membentuk pilar penting dari orientasi objek. Sekarang mari kita bahas konsep penting di balik ADT. Faktanya, ada tiga konsep penting yang terkait dengan ADT—abstraksi data, struktur data, tipe data:

Abstraksi data: Prinsip abstraksi data menyiratkan bahwa bagaimana data disimpan secara tepat diabstraksikan. Ini berarti bahwa setiap entitas di luar objek (yaitu, turunan dari ADT) tidak akan memiliki pengetahuan tentang bagaimana data secara tepat disimpan, diatur, dan dimanipulasi di dalam objek. Entitas eksternal ke objek dapat mengakses data internal ke objek hanya dengan memanggil metode tertentu yang terdefinisi dengan baik yang didukung oleh objek. Pertimbangkan ADT seperti tumpukan. Data objek tumpukan dapat disimpan secara internal dalam larik, daftar tertaut linier, atau daftar tertaut dua arah. Entitas eksternal tidak memiliki pengetahuan tentang ini dan dapat mengakses data objek tumpukan hanya melalui operasi yang didukung seperti push dan pop.

Struktur data: Struktur data dibangun dari kumpulan item data primitif. Sama seperti seorang insinyur sipil membangun struktur teknik sipil yang besar dengan menggunakan bahan bangunan primitif seperti batu bata, batang besi, dan semen; seorang programmer dapat membangun struktur data sebagai kumpulan terorganisir dari item data primitif seperti integer, angka floating point, karakter, dll.

Tipe data: Tipe adalah terminologi bahasa pemrograman yang mengacu pada apa pun yang dapat dipakai. Misalnya, int, float, char, dll., adalah tipe data dasar yang didukung oleh bahasa pemrograman C. Dengan demikian, kita dapat mengatakan bahwa ADT adalah tipe data yang ditentukan pengguna.

Dalam orientasi objek, kelas adalah ADT. Tapi, apa keuntungan mengembangkan aplikasi menggunakan ADT? Mari kita periksa tiga keuntungan utama menggunakan ADT dalam program:

- Data objek dienkapsulasi dalam metode. Prinsip enkapsulasi juga dikenal sebagai penyembunyian data. Prinsip enkapsulasi mensyaratkan bahwa data dapat diakses dan dimanipulasi hanya melalui metode yang didukung oleh objek dan tidak secara langsung. Ini melokalisasi kesalahan. Alasannya adalah sebagai berikut. Tidak ada elemen program yang diizinkan untuk mengubah data, kecuali melalui pemanggilan salah satu metode. Jadi, kesalahan apa pun dapat dengan mudah dilacak ke segmen kode yang mengubah nilainya. Artinya, metode yang mengubah item data, membuatnya salah dapat dengan mudah diidentifikasi.
- Desain berbasis ADT menampilkan kohesi tinggi dan kopling rendah. Oleh karena itu, desain berorientasi objek sangat modular.
- Karena prinsip abstraksi digunakan, itu membuat solusi desain mudah dimengerti dan membantu mengelola kompleksitas.

Objek serupa membentuk kelas. Dengan kata lain, setiap objek adalah anggota dari beberapa kelas. Kelas dapat mewarisi fitur dari kelas super. Secara konseptual, objek berkomunikasi melalui pesan yang lewat. Objek memiliki data internalnya sendiri. Jadi suatu objek mungkin ada di negara bagian yang berbeda tergantung nilai data internal. Di negara bagian yang berbeda, suatu objek dapat berperilaku berbeda. Bagian ini akan diuraikan konsep-konsep ini di Bab 7 dan selanjutnya asdalah metodologi desain berorientasi objek di Bab 8.

Pendekatan desain berorientasi fungsi berorientasi objek

Berikut ini adalah beberapa perbedaan penting antara desain berorientasi fungsi dan berorientasi objek:

- Tidak seperti metode desain berorientasi fungsi dalam OOD, abstraksi dasar bukanlah layanan yang tersedia bagi pengguna sistem seperti buku terbitan, detail buku tampilan, buku yang diterbitkan, dll., tetapi entitas dunia nyata seperti anggota , buku, register buku, dll. Misalnya dalam OOD, perangkat lunak penggajian karyawan tidak dikembangkan dengan merancang fungsi seperti update-employee-record, get-employee-address, dll., tetapi dengan mendesain objek seperti karyawan , departemen, dll.
- Dalam OOD, informasi keadaan ada dalam bentuk data yang didistribusikan di antara beberapa objek sistem. Sebaliknya, dalam desain prosedural, informasi status tersedia di penyimpanan data bersama yang terpusat. Misalnya, saat mengembangkan sistem daftar gaji karyawan, data karyawan seperti nama karyawan, nomor kode, gaji pokok, dll., biasanya diimplementasikan sebagai data global dalam sistem pemrograman tradisional; sedangkan dalam desain berorientasi objek, data ini didistribusikan di antara objek karyawan yang berbeda dari sistem. Obyek berkomunikasi melalui message passing. Oleh karena itu, satu objek dapat menemukan informasi status objek lain dengan mengirimkan pesan ke objek tersebut. Tentu saja, di suatu tempat atau lainnya fungsi dunia nyata harus diimplementasikan.
- Teknik berorientasi fungsi mengelompokkan fungsi bersama-sama jika, sebagai sebuah kelompok, mereka membentuk fungsi tingkat yang lebih tinggi. Di sisi lain, teknik berorientasi objek mengelompokkan fungsi bersama berdasarkan data yang mereka operasikan.

Untuk mengilustrasikan perbedaan antara pendekatan desain berorientasi objek dan berorientasi fungsi, mari kita perhatikan sebuah contoh—sistem alarm kebakaran otomatis untuk bangunan besar.

Sistem alarm kebakaran otomatis—persyaratan pelanggan

Pemilik gedung bertingkat yang besar ingin memiliki sistem alarm kebakaran terkomputerisasi yang dirancang, dikembangkan, dan dipasang di gedungnya. Detektor asap dan alarm kebakaran akan ditempatkan di setiap ruangan gedung. Sistem alarm kebakaran akan memantau status detektor asap ini. Setiap kali kondisi kebakaran dilaporkan oleh salah satu detektor asap, sistem alarm kebakaran harus menentukan lokasi di mana api telah dirasakan dan kemudian membunyikan alarm hanya di lokasi tetangga. Sistem alarm kebakaran juga harus mem-flash pesan alarm di konsol komputer. Personel pemadam kebakaran akan menjaga konsol sepanjang waktu. Setelah kondisi kebakaran berhasil ditangani, sistem alarm kebakaran harus mendukung pengaturan ulang alarm oleh personel pemadam kebakaran.

Pendekatan berorientasi fungsi: Dalam pendekatan ini, fungsi tingkat tinggi yang berbeda pertama kali diidentifikasi, dan kemudian struktur data dirancang.

```
/* Global data (system state) accessible by various functions */
        BOOL  detector_status[MAX_ROOMS];
        int   detector_locs[MAX_ROOMS];
        BOOL  alarm-status[MAX_ROOMS]; /* alarm activated when status is set */
        int   alarm_locs[MAX_ROOMS]; /* room number where alarm is located */
        int   neighbour-alarms[MAX-ROOMS][10]; /* each detector has at most */
                                             /* 10 neighbouring alarm locations */
        int   sprinkler[MAX_ROOMS];
```

Fungsi-fungsi yang beroperasi pada status sistem adalah:

interrogate_detectors();
get_detector_location();
determine_neighbour_alarm();
determine_neighbour_sprinkler();
ring_alarm();
activate_sprinkler();
reset_alarm();
reset_sprinkler();
report_fire_location();

Pendekatan berorientasi objek: Dalam pendekatan berorientasi objek, kelas objek yang berbeda diidentifikasi. Selanjutnya, metode dan data untuk setiap objek diidentifikasi. Akhirnya, jumlah instance yang sesuai dari setiap kelas dibuat.

class detector
attributes: status, location, neighbours
operations: create, sense-status, get-location,
find-neighbours
class alarm
attributes: location, status
operations: create, ring-alarm, get_location, resetalarm
class sprinkler
attributes: location, status
operations: create, activate-sprinkler, get_location,
reset-sprinkler

Kita sekarang dapat membandingkan pendekatan berorientasi fungsi dan berorientasi objek berdasarkan dua contoh yang dibahas di atas, dan dengan mudah mengamati perbedaan utama berikut:

- Dalam program berorientasi fungsi, status sistem (data) terpusat dan beberapa fungsi mengakses dan memodifikasi data pusat ini. Dalam kasus program berorientasi objek, informasi status (data) didistribusikan di antara berbagai objek.
- Dalam desain berorientasi objek, data bersifat pribadi di objek yang berbeda dan ini tidak tersedia untuk objek lain untuk akses langsung dan modifikasi.
- Unit dasar merancang program berorientasi objek adalah objek, sedangkan fungsi dan modul dalam perancangan prosedural. Objek muncul sebagai kata benda dalam deskripsi masalah; sedangkan fungsi muncul sebagai kata kerja.

Pada titik ini, kita harus menekankan bahwa desain berorientasi objek tidak perlu diimplementasikan dengan menggunakan bahasa berorientasi objek saja. Namun, bahasa berorientasi objek seperti C++ dan Java mendukung definisi semua mekanisme dasar kelas, pewarisan, objek, metode, dll. dan juga mendukung semua konsep kunci berorientasi objek yang baru saja kita bahas. Dengan demikian, bahasa berorientasi objek memfasilitasi implementasi OOD. Namun, OOD juga dapat diimplementasikan menggunakan bahasa prosedural konvensional—meskipun mungkin memerlukan lebih banyak upaya untuk mengimplementasikan OOD menggunakan bahasa prosedural dibandingkan dengan upaya yang diperlukan untuk mengimplementasikan desain yang sama menggunakan bahasa berorientasi objek. Faktanya, kompiler C++ yang lebih lama pada dasarnya adalah pra-prosesor yang menerjemahkan kode C++ ke dalam kode C.

Meskipun teknik berorientasi objek dan berorientasi fungsi adalah pendekatan yang sangat berbeda untuk desain perangkat lunak, namun yang satu tidak menggantikan yang lain; tetapi mereka saling melengkapi dalam beberapa hal. Misalnya, biasanya seseorang menerapkan teknik berorientasi fungsi top-down untuk merancang metode internal kelas, setelah kelas diidentifikasi. Dalam hal ini, meskipun secara lahiriah sistem tampaknya telah dikembangkan dalam orientasi objek mode, tetapi di dalam setiap kelas mungkin ada hierarki kecil fungsi yang dirancang secara top-down.

5.6 RINGKASAN

- Desain perangkat lunak biasanya dilakukan melalui dua tahap—desain tingkat tinggi, dan desain detail. Selama desain tingkat tinggi, komponen (modul) penting diidentifikasi dari sistem dan interaksinya. Selama desain rinci, algoritma dan struktur data diidentifikasi.
- Tidak ada solusi desain yang unik untuk masalah apa pun dan orang perlu memilih solusi terbaik di antara serangkaian solusi kandidat. Untuk dapat mencapai hal ini, kita harus mengidentifikasi faktor-faktor yang menjadi dasar desain superior dapat dibedakan dari desain inferior.
- Pemahaman desain adalah kriteria utama yang menentukan kebaikan desain. Ciri dari pemahaman desain dalam hal penggunaan dekomposisi dan prinsip abstraksi yang memuaskan, mengkarakterisasi kedalam hal kohesi, kopling, pelapisan, abstraksi kontrol, fanin, fan-out, dll.
- Mengidentifikasi dua pendekatan yang berbeda secara fundamental terhadap desain perangkat lunak—desain berorientasi fungsi dan desain berorientasi objek. Filosofi penting dalam mengatur kedua pendekatan ini. Kedua pendekatan untuk

desain perangkat lunak ini bukanlah pendekatan yang benar-benar bersaing tetapi pendekatan yang saling melengkapi.

5.7 LATIHAN

1. Pilih opsi yang benar
    a. Luasnya pertukaran data antara dua modul disebut:
        i. Kopling
        ii. Kohesi
        iii. Struktur
        iv. Serikat
    b. Manakah dari jenis kohesi berikut yang dapat dianggap sebagai kohesi terkuat:
        i. Logis
        ii. Kebetulan
        iii. Sementara
        iv. Fungsional
    c. Modul dalam desain perangkat lunak yang baik harus memiliki karakteristik berikut:
        i. Kohesi tinggi, kopling rendah
        ii. Kohesi rendah, kopling tinggi
        iii. Kohesi rendah, kopling rendah
        iv. Kohesi tinggi, kopling tinggi
2. Apakah Anda setuju dengan pernyataan berikut? Solusi desain yang sulit dipahami akan menyebabkan peningkatan biaya pengembangan dan pemeliharaan. Berikan alasan untuk jawaban Anda.
3. Apa yang Anda maksud dengan istilah kohesi dan kopling dalam konteks desain perangkat lunak? Bagaimana konsep-konsep ini berguna dalam mencapai desain sistem yang baik?
4. Apakah yang Anda maksud: desain modular Bagaimana Anda bisa menentukan apakah desain yang diberikan bersifat modular atau tidak?
5. Sebutkan berbagai jenis kohesi yang mungkin ditunjukkan oleh modul dalam desain. Berikan contoh masing-masing.
6. Hitung berbagai jenis kopling yang mungkin ada di antara dua modul. Berikan contoh masing-masing.
7. Benarkah setiap kali Anda meningkatkan kohesi desain Anda, kopling dalam desain akan berkurang secara otomatis? Buktikan jawaban Anda dengan menggunakan contoh-contoh yang sesuai.
8. Menurut Anda apa ciri-ciri desain perangkat lunak yang baik?
9. Apa yang Anda pahami dengan istilah independensi fungsional dalam konteks desain perangkat lunak? Apa keuntungan dari kemandirian fungsional? Bagaimana kemandirian fungsional dalam desain perangkat lunak dapat dicapai?
10. Jelaskan bagaimana prinsip-prinsip abstraksi dan dekomposisi digunakan untuk sampai pada desain yang baik.
11. Apa yang Anda pahami dengan menyembunyikan informasi dalam konteks desain perangkat lunak? Jelaskan mengapa pendekatan desain berdasarkan prinsip penyembunyian informasi cenderung mengarah pada desain yang dapat digunakan kembali dan dapat dipelihara. Ilustrasikan jawaban Anda dengan contoh yang sesuai.
12. Dalam konteks pengembangan perangkat lunak, bedakan antara analisis dan desain sehubungan dengan niat, metodologi, dan teknik dokumentasi yang digunakan.

13. Nyatakan apakah pernyataan berikut BENAR atau SALAH. Berikan alasan untuk jawaban Anda.
    a. Inti dari setiap teknik desain berorientasi fungsi yang baik adalah memetakan fungsi yang melakukan aktivitas serupa ke dalam sebuah modul.
    b. Desain prosedural tradisional dilakukan dari atas ke bawah sedangkan desain berorientasi objek biasanya dilakukan dari bawah ke atas.
    c. Kopling umum adalah jenis kopling terburuk antara dua modul.
    d. Kohesi temporal adalah jenis kohesi terburuk yang dapat dimiliki modul.
    e. Sejauh mana dua modul saling bergantung satu sama lain menentukan kohesi kedua modul.
14. Bandingkan keuntungan relatif dari pendekatan berorientasi objek dan berorientasi fungsi untuk desain perangkat lunak.
15. Sebutkan beberapa teknik desain perangkat lunak berorientasi fungsi yang mapan.
16. Jelaskan penyebab penting dan solusi untuk kopling tinggi antara dua modul perangkat lunak.
17. Masalah apa yang mungkin muncul jika dua modul memiliki kopling tinggi?
18. Masalah apa yang mungkin terjadi jika modul memiliki kohesi rendah?
19. Bedakan antara desain tingkat tinggi dan detail. Dokumen apa yang harus dihasilkan pada penyelesaian desain tingkat tinggi dan detail masing-masing?
20. Apa yang dimaksud dengan istilah kohesi dalam konteks desain perangkat lunak? Benarkah dalam desain yang baik, modul harus memiliki kohesi yang rendah? Mengapa?
21. Apa yang dimaksud dengan istilah kopling dalam konteks desain perangkat lunak? Benarkah dalam desain yang baik, modul harus memiliki kopling yang rendah? Mengapa?
22. Apakah yang Anda maksud: desain modular Apa faktor berbeda yang mempengaruhi modularitas desain? Bagaimana Anda bisa menilai modularitas desain? Apa keuntungan dari desain modular?
23. Bagaimana Anda meningkatkan desain perangkat lunak yang menampilkan kohesi sangat rendah dan kopling tinggi?
24. Jelaskan bagaimana keseluruhan kohesi dan penggabungan desain akan terpengaruh jika semua modul desain digabungkan menjadi satu modul.
25. Jelaskan apa yang Anda pahami dengan istilah dekomposisi dan abstraksi dalam konteks desain perangkat lunak. Bagaimana kedua prinsip ini digunakan untuk mendapatkan desain prosedural yang baik?
26. Apa itu ADT? Keuntungan apa yang diperoleh ketika teknik desain perangkat lunak didasarkan pada ADT? Jelaskan mengapa paradigma objek dikatakan berdasarkan ADT.
27. Dengan menggunakan contoh yang sesuai, jelaskan istilah berikut yang terkait dengan tipe data abstrak (ADT)—abstraksi data, struktur data, tipe data.
28. Apa yang Anda pahami dengan istilah dekomposisi top-down dalam konteks desain berorientasi fungsi? Jelaskan jawaban Anda dengan menggunakan contoh yang sesuai.
29. Apa yang Anda pahami dengan desain perangkat lunak berlapis? Apa keuntungan dari desain berlapis? Jelaskan jawaban Anda dengan menggunakan contoh yang sesuai.
30. Apa perbedaan utama antara metodologi desain perangkat lunak berdasarkan abstraksi fungsional dan yang didasarkan pada abstraksi data? Sebutkan setidaknya satu teknik desain populer berdasarkan masing-masing dari dua paradigma desain perangkat lunak ini.
31. Apa keuntungan utama menggunakan pendekatan berorientasi objek untuk desain perangkat lunak dibandingkan pendekatan berorientasi fungsi?

32. Tunjukkan tiga perbedaan penting antara pendekatan berorientasi fungsi dan berorientasi objek pada desain perangkat lunak. Buktikan jawaban Anda melalui contoh-contoh yang sesuai.
33. Identifikasi kriteria yang akan Anda gunakan untuk memutuskan mana dari dua solusi desain berorientasi fungsi alternatif untuk suatu masalah yang lebih unggul.
34. Jelaskan perbedaan utama antara desain arsitektur, desain tingkat tinggi, dan desain rinci dari sistem perangkat lunak.

BAB 6
DESAIN PERANGKAT LUNAK BERORIENTASI FUNGSI

Teknik desain berorientasi fungsi diusulkan hampir empat dekade lalu. Teknik-teknik ini pada saat ini masih sangat populer dan sedang digunakan di banyak organisasi pengembangan perangkat lunak. Teknik-teknik ini, untuk memulai, memandang sistem sebagai kotak hitam yang menyediakan serangkaian layanan kepada pengguna perangkat lunak. Layanan ini disediakan oleh perangkat lunak (misalnya, buku terbitan, buku penelusuran, dll., untuk Perangkat Lunak Otomasi Perpustakaan kepada penggunanya juga dikenal sebagai fungsi tingkat tinggi yang didukung oleh perangkat lunak. Selama proses desain, fungsi tingkat tinggi ini berturut-turut didekomposisi menjadi fungsi yang lebih rinci Istilah dekomposisi top-down sering digunakan untuk menunjukkan dekomposisi berturut-turut dari satu set fungsi tingkat tinggi menjadi fungsi yang lebih rinci. Setelah dekomposisi top-down dilakukan, berbagai fungsi yang diidentifikasi dipetakan ke modul dan struktur modul dibuat. Struktur modul ini akan memiliki semua karakteristik desain yang baik yang diidentifikasi dalam bab terakhir.

Dalam teks ini, kita tidak akan fokus pada metodologi desain tertentu. Sebagai gantinya, kita akan membahas metodologi yang memiliki fitur penting dari beberapa metodologi desain berorientasi fungsi yang penting. Pendekatan seperti itu akan memungkinkan kita untuk dengan mudah mengasimilasi setiap metodologi desain khusus di masa depan kapan pun diperlukan. Mempelajari metodologi tertentu mungkin diperlukan untuk Anda nanti, karena rumah pengembangan perangkat lunak yang berbeda mengikuti metodologi yang berbeda. Bagaimanapun, teknik desain prosedural yang berbeda dapat dianggap sebagai teknik saudara perempuan yang hanya memiliki sedikit perbedaan dalam hal metodologi dan notasi. Teknik desain yang dibahas dalam teks ini sebagai metodologi analisis terstruktur/desain terstruktur (SA/SD). Teknik ini sangat menarik dari metodologi desain yang diusulkan oleh penulis berikut:

- DeMarco dan Yourdon [1978]
- Constantine dan Yourdon [1979]
- Gane dan Sarson [1979]
- Hatley dan Pirbhai [1987]

Teknik SA/SD dapat digunakan untuk melakukan desain perangkat lunak tingkat tinggi. Rincian teknik SA/SD dibahas lebih lanjut.

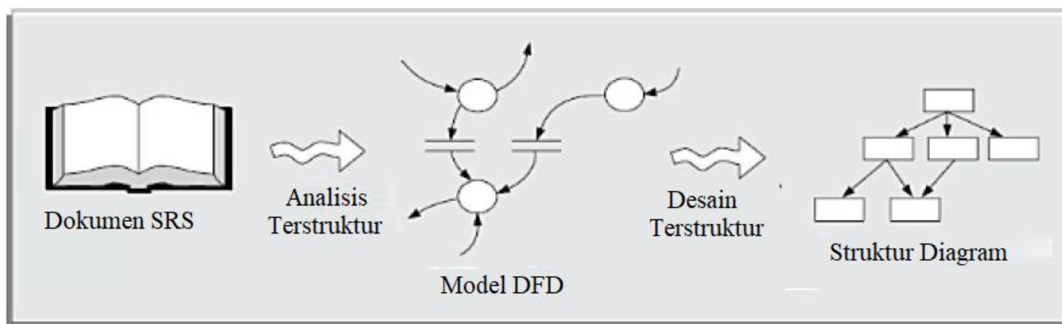
6.1 TINJAUAN METODOLOGI SA/SD

Sesuai dengan namanya, metodologi SA/SD melibatkan pelaksanaan dua aktivitas yang berbeda:

- Analisis terstruktur (SA)
- Desain terstruktur (SD)

Peran analisis terstruktur (SA) dan desain terstruktur (SD) ditunjukkan secara skematis pada Gambar 6.1. Perhatikan hal-hal berikut dari gambar:

- Selama analisis terstruktur, dokumen SRS diubah menjadi model diagram aliran data (DFD).
- Selama desain terstruktur, model DFD diubah menjadi bagan struktur.



Gambar 6.1 Analisis terstruktur dan metodologi desain terstruktur.

Seperti yang ditunjukkan pada Gambar 6.1, aktivitas analisis terstruktur mengubah dokumen SRS menjadi model grafis yang disebut model DFD. Selama analisis terstruktur, dekomposisi fungsional sistem tercapai. Artinya, setiap fungsi yang perlu dilakukan sistem dianalisis dan didekomposisi secara hierarkis menjadi fungsi yang lebih detail. Di sisi lain, selama desain terstruktur, semua fungsi yang diidentifikasi selama analisis terstruktur dipetakan ke struktur modul. Struktur modul ini juga disebut desain tingkat tinggi atau rancangan roftware untuk masalah yang diberikan. Ini direpresentasikan menggunakan diagram struktur. Tahap desain tingkat tinggi biasanya diikuti oleh tahap desain detail. Selama tahap desain rinci, algoritma dan struktur data untuk masing-masing modul dirancang. Detail design dapat langsung diimplementasikan sebagai sistem kerja dengan menggunakan bahasa pemrograman konvensional.

Penting untuk dipahami bahwa tujuan dari analisis terstruktur adalah untuk menangkap struktur rinci dari sistem seperti yang dirasakan oleh pengguna, sedangkan tujuan dari desain terstruktur adalah untuk menentukan struktur solusi yang cocok untuk implementasi dalam beberapa bahasa pemrograman. Oleh karena itu, hasil analisis terstruktur dapat dengan mudah dipahami oleh pengguna. Faktanya, fungsi dan data yang berbeda dalam analisis terstruktur diberi nama menggunakan terminologi pengguna. Oleh karena itu, pengguna bahkan dapat meninjau hasil analisis terstruktur untuk memastikan bahwa ia menangkap semua persyaratannya.

6.2 ANALISIS TERSTRUKTUR

Selama analisis terstruktur, tugas pemrosesan utama (fungsi tingkat tinggi) dari sistem dianalisis, dan aliran data di antara tugas pemrosesan ini direpresentasikan secara grafis. Kontribusi yang signifikan untuk pengembangan teknik analisis terstruktur telah dibuat oleh Gane dan Sarson [1979], dan DeMarco dan Yourdon [1978]. Teknik analisis terstruktur didasarkan pada prinsip-prinsip yang mendasari berikut:

- Pendekatan dekomposisi top-down.
- Penerapan prinsip membagi dan menaklukkan. Melalui ini setiap fungsi tingkat tinggi secara independen didekomposisi menjadi fungsi-fungsi terperinci.
- Representasi grafis dari hasil analisis menggunakan data flow diagram (DFD).

Representasi DFD dari suatu masalah sangat mudah untuk dibangun. Meskipun sangat sederhana, ini adalah alat yang sangat ampuh untuk mengatasi kompleksitas masalah standar industri. DFD adalah model grafis hierarkis dari suatu sistem yang menunjukkan aktivitas pemrosesan atau fungsi yang berbeda yang dilakukan sistem dan pertukaran data di antara fungsi-fungsi tersebut. Harap dicatat bahwa model DFD hanya mewakili aspek aliran data dan tidak menunjukkan urutan eksekusi fungsi yang berbeda dan kondisi berdasarkan fungsi yang mungkin atau mungkin tidak dijalankan. Bahkan, itu benar-benar mengabaikan aspek seperti

aliran kontrol, algoritma khusus yang digunakan oleh fungsi, dll. Dalam terminologi DFD, setiap fungsi disebut proses atau gelembung. Hal ini berguna untuk mempertimbangkan setiap fungsi sebagai stasiun pemrosesan (atau proses) yang mengkonsumsi beberapa data input dan menghasilkan beberapa data output.

DFD adalah teknik pemodelan elegan yang dapat digunakan tidak hanya untuk mewakili hasil analisis terstruktur dari masalah perangkat lunak, tetapi juga berguna untuk beberapa aplikasi lain seperti menunjukkan aliran dokumen atau item dalam suatu organisasi. Ingatlah bahwa di Bab 1 kita telah memberikan contoh (lihat Gambar 1.10) untuk mengilustrasikan bagaimana DFD dapat digunakan untuk mewakili aktivitas pemrosesan dan aliran material di pabrik perakitan mobil otomatis.

***Data Flow Diagrams* (DFD)**

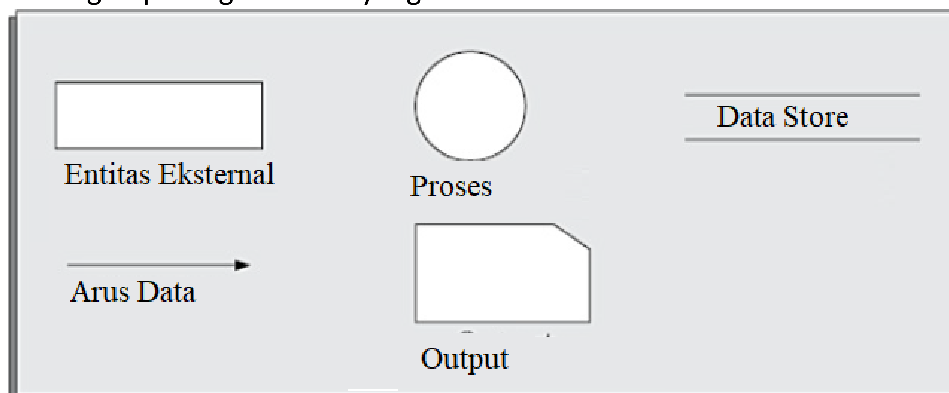
DFD (juga dikenal sebagai bagan gelembung) adalah formalisme grafis sederhana yang dapat digunakan untuk mewakili sistem dalam hal data input ke sistem, berbagai pemrosesan yang dilakukan pada data tersebut, dan data output yang dihasilkan oleh sistem. Alasan utama mengapa teknik DFD begitu populer mungkin karena fakta bahwa DFD adalah formalisme yang sangat sederhana— mudah dipahami dan digunakan. Model DFD menggunakan simbol primitif dalam jumlah yang sangat terbatas (ditunjukkan pada Gambar 6.2) untuk mewakili fungsi yang dilakukan oleh sistem dan aliran data di antara fungsi-fungsi ini.

Dimulai dengan serangkaian fungsi tingkat tinggi yang dilakukan sistem, model DFD mewakili subfungsi yang dilakukan oleh fungsi menggunakan hierarki diagram. Setiap representasi hierarkis adalah sarana yang efektif untuk mengatasi kompleksitas. Pikiran manusia sedemikian rupa sehingga dapat dengan mudah memahami model hierarki sistem apa pun—karena dalam model hierarkis, dimulai dengan model sistem yang sangat abstrak, berbagai detail sistem secara perlahan diperkenalkan melalui berbagai tingkat hierarki. Teknik DFD juga didasarkan pada seperangkat konsep dan aturan intuitif yang sangat sederhana.

Simbol primitif yang digunakan untuk membangun DFD

Pada dasarnya ada lima jenis simbol yang digunakan untuk membangun DFD. Simbol-simbol primitif ini digambarkan pada Gambar 6.2. Arti dari simbol-simbol tersebut dijelaskan sebagai berikut:

Simbol entitas eksternal: Entitas eksternal seperti pustakawan, anggota perpustakaan, dll. diwakili oleh persegi panjang. Entitas eksternal pada dasarnya adalah entitas fisik di luar sistem perangkat lunak yang berinteraksi dengan sistem dengan memasukkan data ke sistem atau dengan mengonsumsi data yang dihasilkan oleh sistem. Selain pengguna manusia, simbol entitas eksternal dapat digunakan untuk mewakili perangkat keras dan perangkat lunak eksternal seperti perangkat lunak aplikasi lain yang akan berinteraksi dengan perangkat lunak yang dimodelkan.



Gambar 6.2 Simbol yang digunakan untuk mendesain DFD.

Simbol aliran data: Busur berarah (atau panah) digunakan sebagai simbol aliran data. Simbol aliran data mewakili aliran data yang terjadi antara dua proses atau antara entitas eksternal dan proses dalam arah panah aliran data. Simbol aliran data biasanya dianotasi dengan nama data yang sesuai. Misalnya DFD pada Gambar 6.3(a) menunjukkan tiga aliran data—nomor item data mengalir dari proses read-number ke validasi-number, dataitem mengalir ke read-number, dan valid-number mengalir keluar darivalid-number.

Simbol penyimpanan data: Penyimpanan data direpresentasikan menggunakan dua garis paralel. Ini mewakili file logis. Artinya, simbol penyimpanan data dapat mewakili struktur data atau file fisik pada disk. Setiap penyimpanan data terhubung ke proses melalui simbol aliran data. Arah panah aliran data menunjukkan apakah data sedang dibaca atau ditulis ke dalam penyimpanan data. Panah yang mengalir masuk atau keluar dari penyimpanan data secara implisit mewakili seluruh data dari penyimpanan data dan karenanya panah yang menghubungkan ke penyimpanan data tidak perlu dianotasi dengan nama item data yang sesuai. Sebagai contoh penyimpanan data, nomor adalah penyimpanan data pada Gambar 6.3(b).

Simbol keluaran: Simbol keluaran adalah seperti yang ditunjukkan pada Gambar 6.2. Simbol output digunakan ketika hard copy diproduksi.

Notasi yang kita ikuti dalam teks ini lebih dekat dengan notasi Yourdon daripada notasi lainnya. Terkadang Anda mungkin menemukan notasi dalam buku lain yang sedikit berbeda dari yang dibahas di sini. Misalnya, penyimpanan data mungkin terlihat seperti kotak dengan salah satu ujungnya terbuka. Itu karena, mereka mungkin mengikuti notasi seperti yang dimiliki Gane dan Sarson [1979].

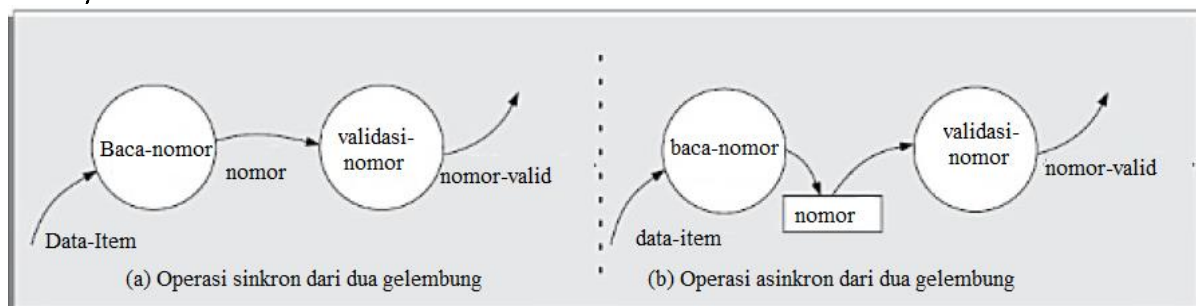
Simbol fungsi: Fungsi direpresentasikan menggunakan lingkaran. Simbol ini disebut proses atau gelembung. Gelembung dianotasi dengan nama fungsi yang sesuai (lihat Gambar 6.3).

Konsep penting yang terkait dengan pembuatan model DFD

Sebelum kita membahas bagaimana membangun model DFD suatu sistem, mari kita bahas beberapa konsep penting yang terkait dengan DFD:

Operasi sinkron dan asinkron

Jika dua gelembung terhubung langsung oleh panah aliran data, maka keduanya sinkron. Ini berarti bahwa mereka beroperasi pada kecepatan yang sama. Contoh pengaturan seperti itu ditunjukkan pada Gambar 6.3(a). Di sini, gelembung validate-number dapat mulai memproses hanya setelah gelembung nomor baca telah menyediakan data ke dalamnya; dan gelembung nomor baca harus menunggu hingga gelembung validate-number menghabiskan datanya.



Gambar 6.3 Aliran data sinkron dan asinkron.

Namun, jika dua gelembung dihubungkan melalui penyimpanan data, seperti pada Gambar 6.3(b) maka kecepatan operasi gelembung tidak tergantung. Pernyataan tersebut dapat dijelaskan dengan menggunakan penalaran berikut. Data yang dihasilkan oleh

gelembung produsen disimpan di penyimpanan data. Oleh karena itu, ada kemungkinan bahwa gelembung produsen menyimpan beberapa item data, bahkan sebelum gelembung konsumen mengkonsumsi salah satu dari mereka.

Kamus data

Setiap model DFD dari suatu sistem harus disertai dengan kamus data. Kamus data mencantumkan semua item data yang muncul dalam model DFD. Item data yang terdaftar mencakup semua aliran data dan isi dari semua penyimpanan data yang muncul di semua DFD dalam model DFD. Harap diingat bahwa model DFD dari suatu sistem biasanya terdiri dari beberapa DFD, yaitu DFD level 0, DFD level 1, DFD level 2, dll., seperti yang ditunjukkan pada Gambar 6.4. Namun, kamus data tunggal harus menangkap semua data yang muncul di semua DFD yang merupakan model DFD dari suatu sistem. Kamus data mencantumkan tujuan dari semua item data dan definisi semua item data komposit dalam hal item data komponennya. Misalnya, entri kamus data dapat menunjukkan bahwa data grossPay terdiri dari komponen RegularPay dan OvertimePay.

GrossPay = RegularPay + OvertimePay

Untuk unit terkecil dari item data, kamus data hanya mencantumkan nama dan tipenya. Item data komposit dinyatakan dalam item data komponen menggunakan operator tertentu. Operator yang menggunakan item data komposit dapat diekspresikan dalam hal item data komponennya dibahas selanjutnya. Kamus memainkan peran yang sangat penting dalam setiap proses pengembangan perangkat lunak, terutama karena alasan berikut:

- Kamus data menyediakan terminologi standar untuk semua data yang relevan untuk digunakan oleh developer yang bekerja dalam suatu proyek. Kosakata yang konsisten untuk item data sangat penting, karena dalam proyek besar developer proyek yang berbeda memiliki kecenderungan untuk menggunakan istilah yang berbeda untuk merujuk pada data yang sama, yang secara tidak perlu menyebabkan kebingungan.
- Kamus data membantu developer untuk menentukan definisi struktur data yang berbeda dalam hal elemen komponen mereka saat mengimplementasikan desain.
- Kamus data membantu melakukan analisis dampak. Artinya, dimungkinkan untuk menentukan pengaruh beberapa data pada berbagai aktivitas pemrosesan dan sebaliknya. Analisis dampak semacam itu sangat berguna ketika seseorang ingin memeriksa dampak perubahan tipe nilai input, atau bug di beberapa fungsi, dll.

Untuk sistem yang besar, kamus data bisa menjadi sangat kompleks dan banyak. Bahkan proyek berukuran sedang dapat memiliki ribuan entri dalam kamus data. Menjadi sangat sulit untuk memelihara kamus yang banyak secara manual. Alat rekayasa perangkat lunak berbantuan komputer (CASE) berguna untuk mengatasi masalah ini. Sebagian besar alat CASE biasanya menangkap item data yang muncul dalam DFD saat DFD digambar, dan secara otomatis menghasilkan kamus data. Akibatnya, para desainer tidak perlu menghabiskan hampir semua upaya dalam membuat kamus data. Alat CASE ini juga mendukung beberapa fasilitas bahasa query untuk query tentang definisi dan penggunaan item data. Misalnya, kueri dapat dirumuskan untuk menentukan item data mana yang memengaruhi proses mana, atau proses mana yang memengaruhi item data mana, atau definisi dan penggunaan item data tertentu, dll. Penanganan kueri difasilitasi dengan menyimpan kamus data dalam manajemen basis data relasional sistem (RDBMS).

Definisi data

Item data komposit dapat didefinisikan dalam hal item data primitif menggunakan operator definisi data berikut.

- +: menunjukkan komposisi dua item data, mis. a+b mewakili data a dan b.

- [,,]: mewakili pemilihan, yaitu salah satu item data yang tercantum di dalam kurung siku dapat terjadi Misalnya, [a,b] mewakili baik a terjadi atau b terjadi.
- (): isi di dalam tanda kurung mewakili data opsional yang mungkin muncul atau tidak. a+(b) mewakili a atau a+b terjadi.
- {}: mewakili definisi data berulang, mis. {name}5 mewakili lima data nama
- {name}* mewakili nol atau lebih contoh data nama.
- =: mewakili kesetaraan, mis. a=b+c berarti a adalah item data gabungan yang terdiri dari b dan c.
- /* */: Apa pun yang muncul di dalam /* dan */ dianggap sebagai komentar.

6.3 MENGEMBANGKAN MODEL DFD DARI SISTEM

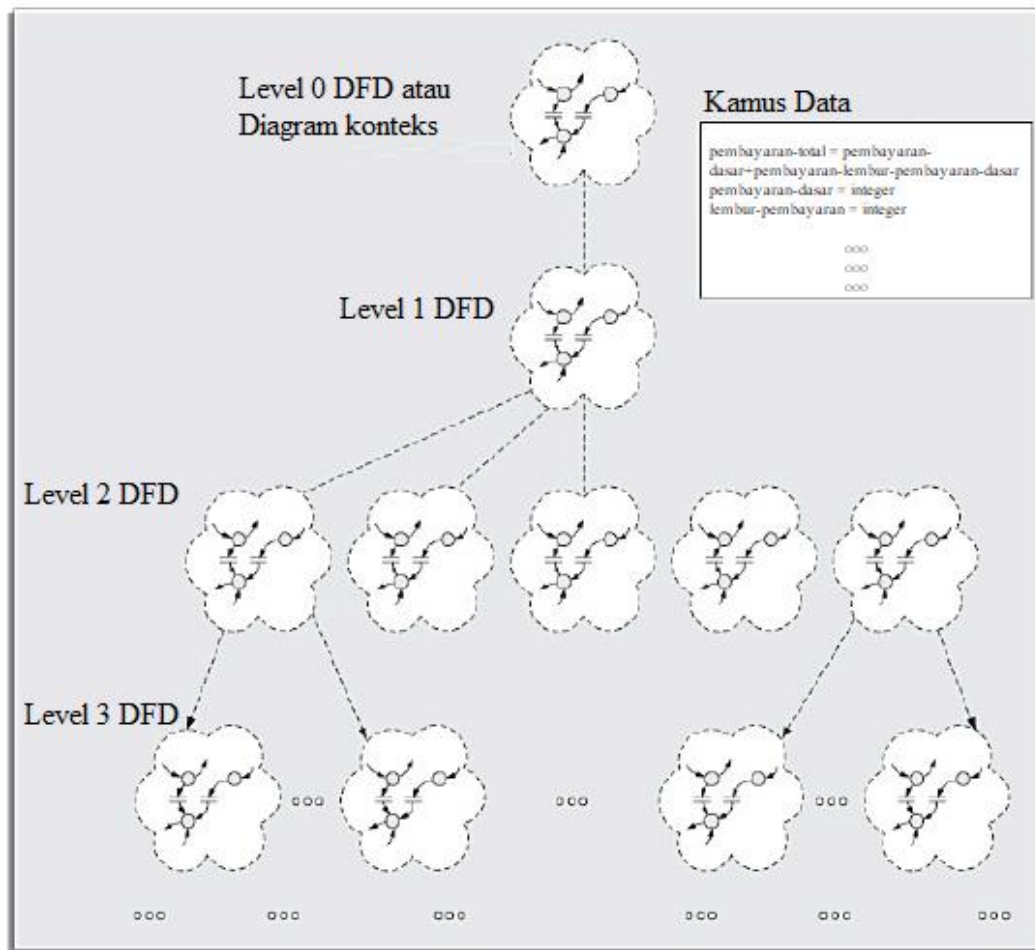
Model DFD dari suatu sistem secara grafis mewakili bagaimana setiap data input ditransformasikan ke data output yang sesuai melalui hierarki DFD. Model DFD dari suatu masalah terdiri dari banyak DFD dan kamus data tunggal. Model DFD dari suatu sistem dibangun dengan menggunakan hierarki DFD (lihat Gambar 6.4). DFD level atas disebut DFD level 0 atau diagram konteks. Ini adalah representasi sistem yang paling abstrak (paling sederhana) (tingkat tertinggi). Ini adalah yang paling mudah untuk menggambar dan memahami. Pada setiap FD d tingkat bawah berturut-turut, semakin banyak detail yang diperkenalkan secara bertahap. Untuk mengembangkan model DFD tingkat yang lebih tinggi, proses didekomposisi menjadi subprosesnya dan aliran data di antara subproses ini diidentifikasi.

Untuk mengembangkan model aliran data suatu sistem, pertama-tama representasi paling abstrak (tingkat tertinggi) dari masalah harus diselesaikan. Selanjutnya, DFD tingkat yang lebih rendah dikembangkan. Level 0 dan Level 1 masing-masing hanya terdiri dari satu DFD. Level 2 dapat berisi hingga 7 DFD terpisah, dan level 3 hingga 49 DFD, dan seterusnya. Namun, hanya ada satu kamus data untuk seluruh model DFD. Semua nama data yang muncul di semua DFD diisi dalam kamus data dan kamus data berisi definisi semua item data.

Diagram konteks

Diagram konteks adalah representasi aliran data yang paling abstrak (tingkat tertinggi) dari suatu sistem. Ini mewakili seluruh sistem sebagai gelembung tunggal. Gelembung dalam diagram konteks dianotasi dengan nama sistem perangkat lunak yang sedang dikembangkan (biasanya kata benda). Ini adalah satu-satunya gelembung dalam model DFD, di mana kata benda digunakan untuk menamai gelembung. Gelembung di semua level lainnya dianotasi dengan kata kerja sesuai dengan fungsi utama yang dilakukan oleh gelembung. Hal ini diharapkan karena tujuan diagram konteks adalah untuk menangkap konteks sistem daripada fungsinya. Sebagai contoh diagram konteks, pertimbangkan diagram konteks perangkat lunak yang dikembangkan untuk mengotomatisasi kegiatan pembukuan supermarket (lihat Gambar 6.10). Diagram konteks telah diberi label sebagai 'perangkat lunak Supermarket'.

Diagram konteks menetapkan konteks di mana sistem beroperasi; yaitu siapa pengguna, data apa yang dimasukkan ke sistem, dan data apa yang diterima sistem. Diagram konteks nama dari DFD level 0 dibenarkan karena mewakili konteks di mana sistem akan ada; yaitu, entitas eksternal yang akan berinteraksi dengan sistem dan item data spesifik yang akan mereka suplai ke sistem dan item data yang akan mereka terima dari sistem. Berbagai entitas eksternal dengan sistem yang berinteraksi dan aliran data yang terjadi antara sistem dan entitas eksternal diwakili. Data masukan ke sistem dan data keluaran dari sistem direpresentasikan sebagai panah masuk dan keluar. Panah aliran data ini harus diberi keterangan dengan nama data yang sesuai.



Gambar 6.4 Model DFD suatu sistem terdiri dari hierarki DFD dan kamus data tunggal.

Untuk mengembangkan diagram konteks sistem, kita harus menganalisis dokumen SRS untuk mengidentifikasi berbagai jenis pengguna yang akan menggunakan sistem dan jenis data yang akan mereka masukkan ke sistem dan data yang akan mereka terima dari sistem. sistem. Di sini, istilah pengguna sistem juga mencakup sistem eksternal apa pun yang memasok data ke atau menerima data dari sistem.

Tingkat 1 DFD

DFD level 1 biasanya berisi tiga hingga tujuh gelembung. Artinya, sistem direpresentasikan sebagai melakukan tiga sampai tujuh fungsi penting. Untuk mengembangkan DFD tingkat 1, periksa persyaratan fungsional tingkat tinggi dalam dokumen SRS. Jika ada tiga hingga tujuh persyaratan fungsional tingkat tinggi, maka masing-masing dapat langsung direpresentasikan sebagai gelembung di DFD tingkat 1. Selanjutnya, periksa data input ke fungsi-fungsi ini dan output data dari fungsi-fungsi ini seperti yang didokumentasikan dalam dokumen SRS dan gambarkan dengan tepat dalam diagram.

Bagaimana jika suatu sistem memiliki lebih dari tujuh persyaratan tingkat tinggi yang diidentifikasi dalam dokumen SRS? Dalam hal ini, beberapa persyaratan terkait harus digabungkan dan direpresentasikan sebagai gelembung tunggal di DFD level 1. Ini dapat dibagi dengan tepat di tingkat DFD yang lebih rendah. Jika suatu sistem memiliki kurang dari tiga persyaratan fungsional tingkat tinggi, maka beberapa persyaratan tingkat tinggi perlu dipecah menjadi subfungsinya sehingga kita memiliki kira-kira lima hingga tujuh gelembung yang direpresentasikan pada diagram. Ilustrasi dari konstruksi DFD level 1 ada pada Contoh 6.1 hingga 6.4.

Dekomposisi

Setiap gelembung di DFD mewakili fungsi yang dilakukan oleh sistem. Gelembung didekomposisi menjadi subfungsi pada tingkat model DFD yang berurutan. Penguraian gelembung juga dikenal sebagai pemfaktoran atau ledakan gelembung. Setiap gelembung pada setiap tingkat DFD biasanya didekomposisi menjadi tiga hingga tujuh gelembung. Beberapa gelembung di tingkat mana pun m a k e tingkat itu berlebihan. Misalnya, jika sebuah gelembung didekomposisi menjadi satu atau dua gelembung saja, maka dekomposisi ini menjadi sepele dan berlebihan. Di sisi lain, terlalu banyak gelembung (yaitu lebih dari tujuh gelembung) pada setiap tingkat DFD membuat model DFD sulit untuk dipahami. Dekomposisi gelembung harus dilakukan sampai tingkat tercapai di mana fungsi gelembung dapat dijelaskan dengan menggunakan algoritma sederhana. Sekarang kita dapat menjelaskan bagaimana mengembangkan model DFD dari sebuah sistem secara lebih sistematis.

Konstruksi diagram konteks: Periksa dokumen SRS untuk menentukan:

- Berbagai fungsi tingkat tinggi yang perlu dilakukan sistem.
- Input data ke setiap fungsi tingkat tinggi.
- Keluaran data dari setiap fungsi tingkat tinggi.
- Interaksi (aliran data) di antara fungsi tingkat tinggi yang teridentifikasi.

Gambarkan aspek-aspek fungsi tingkat tinggi ini dalam bentuk diagram. Ini akan membentuk diagram aliran data tingkat atas (DFD), biasanya disebut DFD 0.

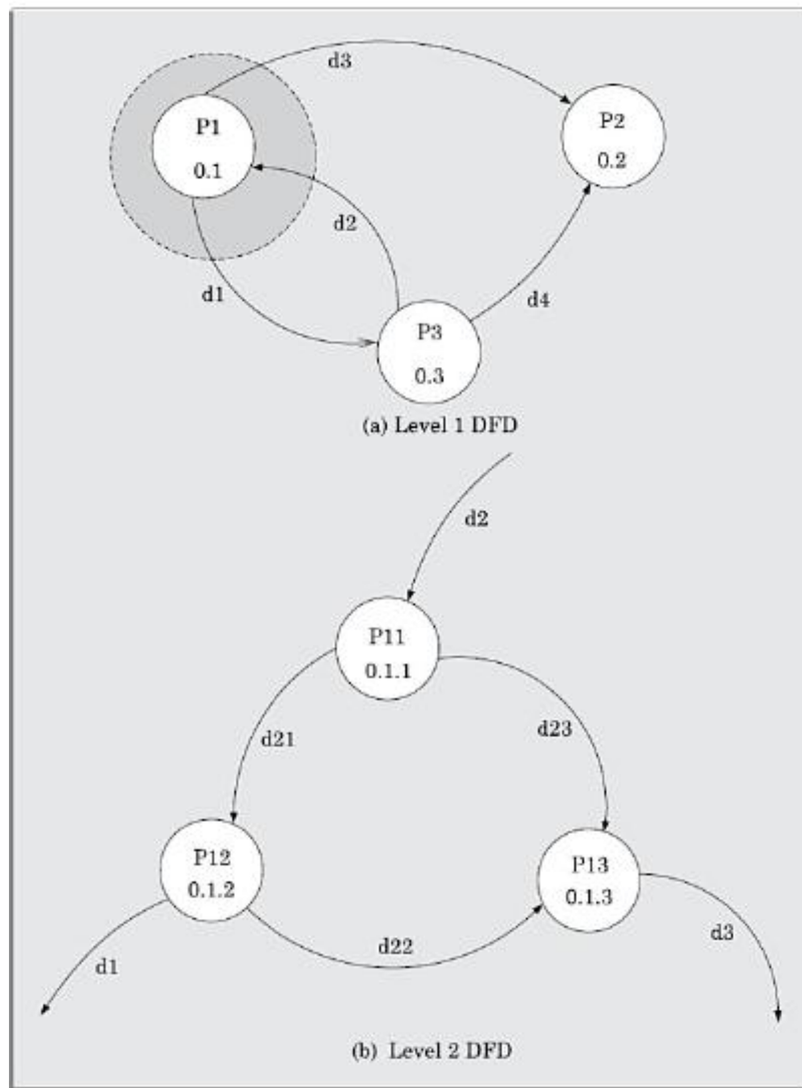
Konstruksi diagram level 1: Periksa fungsi tingkat tinggi yang dijelaskan dalam dokumen SRS. Jika ada tiga sampai tujuh persyaratan tingkat tinggi dalam dokumen SRS, maka mewakili masing-masing fungsi tingkat tinggi dalam bentuk gelembung. Jika ada lebih dari tujuh gelembung, maka beberapa di antaranya harus digabungkan. Jika ada kurang dari tiga gelembung, maka beberapa di antaranya harus dipecah. Konstruksi diagram tingkat bawah: Dekomposisi setiap fungsi tingkat tinggi menjadi subfungsi penyusunnya melalui rangkaian aktivitas berikut:

- Identifikasi subfungsi yang berbeda dari fungsi tingkat tinggi.
- Identifikasi input data ke masing-masing subfungsi ini.
- Identifikasi output data dari masing-masing subfungsi ini.
- Identifikasi interaksi (aliran data) di antara subfungsi ini.

Gambarkan aspek-aspek ini dalam bentuk diagram menggunakan DFD. Ulangi Langkah 3 secara rekursif untuk setiap subfungsi hingga subfungsi dapat direpresentasikan dengan menggunakan algoritma sederhana.

Penomoran gelembung

Hal ini diperlukan untuk nomor gelembung yang berbeda terjadi di DFD. Angka-angka ini membantu dalam mengidentifikasi gelembung apa pun di DFD secara unik dari nomor gelembungnya. Gelembung di tingkat konteks biasanya diberi nomor 0 untuk menunjukkan bahwa itu adalah DFD tingkat 0. Gelembung pada level 1 diberi nomor, 0,1, 0,2, 0,3, dst. Ketika gelembung bernomor x diurai, gelembung anak-anaknya diberi nomor x.1, x.2, x.3, dst. Dalam skema penomoran ini, dengan melihat jumlah gelembung kita dapat dengan jelas menentukan levelnya, leluhurnya, dan penerusnya.



Gambar 6.5 Contoh yang menunjukkan dekomposisi seimbang.

Menyeimbangkan DFD

Model DFD dari suatu sistem biasanya terdiri dari banyak DFD yang diatur dalam suatu hierarki. Dalam konteks ini, DFD diperlukan untuk diseimbangkan sehubungan dengan gelembung yang sesuai dari DFD induk. Data yang masuk atau keluar dari bubble harus sesuai dengan aliran data pada level DFD berikutnya. Ini dikenal sebagai penyeimbangan DFD. lustrasi konsep penyeimbangan DFD terdapat pada Gambar 6.5. Pada DFD level 1, item data d1 dan d3 mengalir keluar dari gelembung 0,1 dan item data d2 mengalir ke dalam gelembung 0,1 (ditunjukkan dengan lingkaran putus-putus). Di level berikutnya, gelembung 0.1 didekomposisi menjadi tiga DFD (0.1.1,0.1.2,0.1.3). Dekomposisi seimbang, karena d1 dan d3 mengalir keluar dari diagram level 2 dan d2 mengalir masuk. Harap dicatat bahwa panah menjuntai (d1,d2,d3) mewakili aliran data ke dalam atau keluar dari diagram.

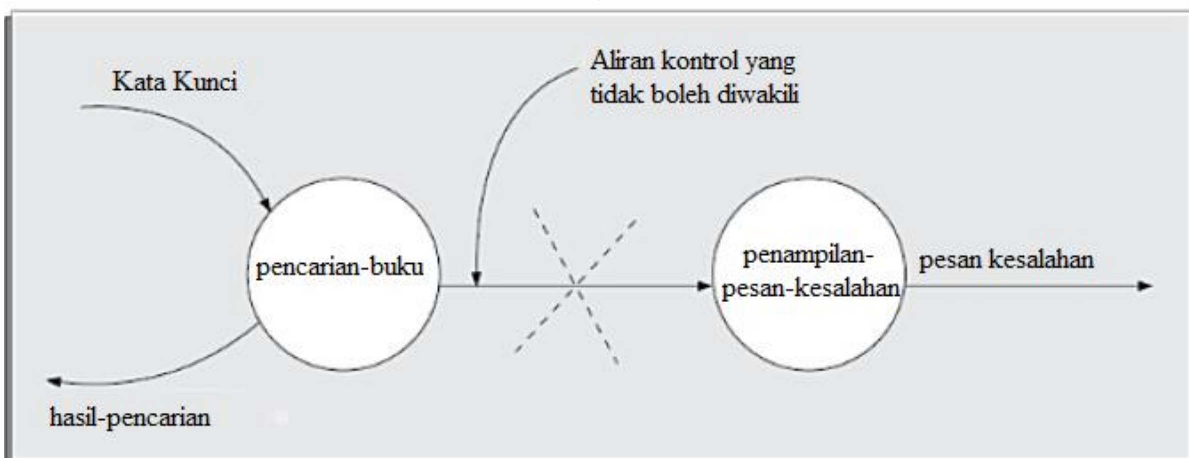
Seberapa jauh terurai?

Gelembung tidak boleh didekomposisi lebih lanjut setelah gelembung ditemukan untuk mewakili serangkaian instruksi sederhana. Untuk masalah sederhana, dekomposisi hingga level 1 sudah cukup. Namun, masalah standar industri besar mungkin memerlukan dekomposisi hingga level 3 atau level 4. Jarang, jika pernah, dekomposisi di luar level 4 diperlukan.

Kesalahan yang sering dibuat saat membuat model DFD

Meskipun DFD mudah dipahami dan digambar, mahasiswa dan praktisi sama-sama menghadapi jenis masalah yang sama saat memodelkan masalah perangkat lunak menggunakan DFD. Sementara belajar dari pengalaman adalah hal yang kuat, itu adalah teknik pedagogis yang mahal di dunia bisnis. Oleh karena itu, sangat berguna untuk memahami berbagai jenis kesalahan yang biasanya dilakukan pemula saat membangun model sistem DFD, sehingga Anda dapat secara sadar mencoba menghindarinya. Kesalahannya adalah sebagai berikut:

- Banyak pemula melakukan kesalahan dengan menggambar lebih dari satu gelembung dalam diagram konteks. Diagram konteks harus menggambarkan sistem sebagai gelembung tunggal.
- Banyak pemula membuat model DFD di mana entitas eksternal muncul di semua level DFD. Semua entitas eksternal yang berinteraksi dengan sistem harus diwakili hanya dalam diagram konteks. Entitas eksternal tidak boleh muncul di DFD di level lain mana pun.
- Ini adalah kesalahan umum untuk memiliki terlalu sedikit atau terlalu banyak gelembung dalam DFD. Hanya tiga hingga tujuh gelembung per diagram yang diizinkan. Ini juga berarti bahwa setiap gelembung dalam DFD harus diuraikan tiga hingga tujuh gelembung di tingkat berikutnya.
- Banyak pemula meninggalkan DFD pada tingkat yang berbeda dari model DFD tidak seimbang.
- Kesalahan umum yang dilakukan oleh banyak pemula saat mengembangkan model DFD adalah mencoba merepresentasikan informasi kontrol dalam DFD.

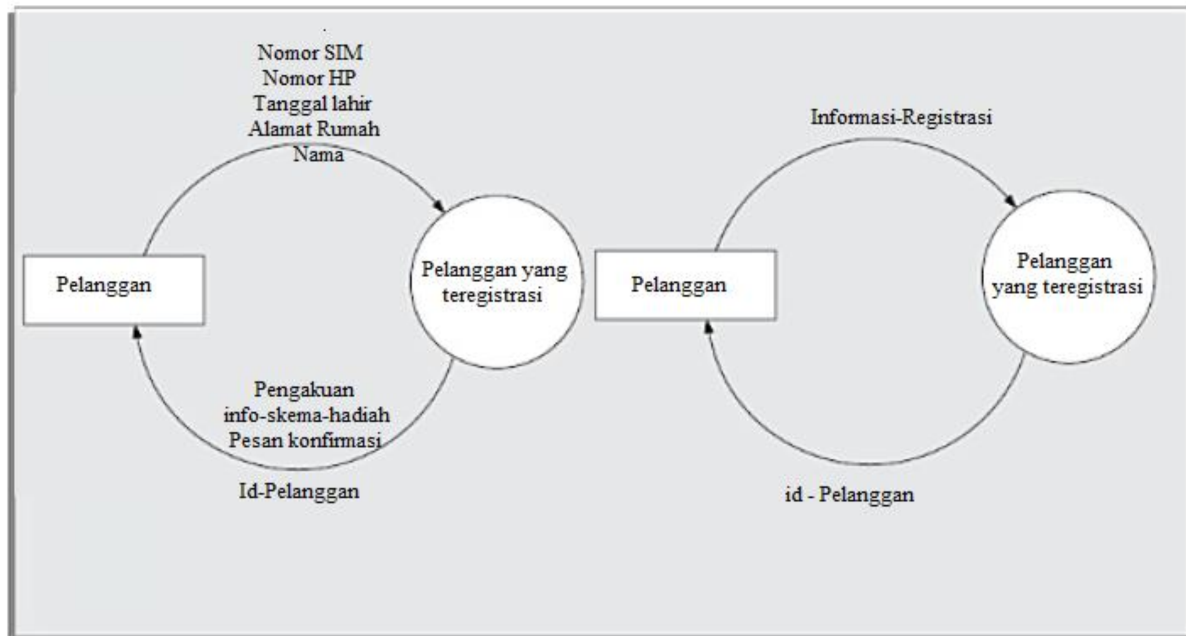


Gambar 6.6 Menampilkan informasi kontrol pada DFD tidak benar.

Penting untuk disadari bahwa DFD hanya mewakili aliran data, dan tidak mewakili informasi kontrol apa pun. Berikut ini adalah beberapa ilustratif kesalahan mencoba untuk mewakili aspek kontrol seperti:

Ilustrasi 1. Sebuah buku dapat dicari di katalog perpustakaan dengan memasukkan namanya. Jika buku tersedia di perpustakaan, maka detail buku akan ditampilkan. Jika buku tidak tercantum dalam katalog, maka akan muncul pesan kesalahan. Saat mengembangkan model DFD untuk masalah sederhana ini, banyak pemula melakukan kesalahan menggambar panah (seperti yang ditunjukkan pada Gambar 6.6) untuk menunjukkan bahwa fungsi kesalahan dipanggil setelah buku pencarian. Tapi, ini adalah informasi kontrol dan tidak boleh ditampilkan di DFD.

Ilustrasi 2. Jenis kesalahan lain terjadi ketika seseorang mencoba untuk mewakili kapan atau dalam urutan apa fungsi (proses) yang berbeda dipanggil. Sebuah DFD juga tidak boleh mewakili kondisi di mana fungsi yang berbeda dipanggil.



Gambar 6.7 Ilustrasi cara menghindari kekacauan data.

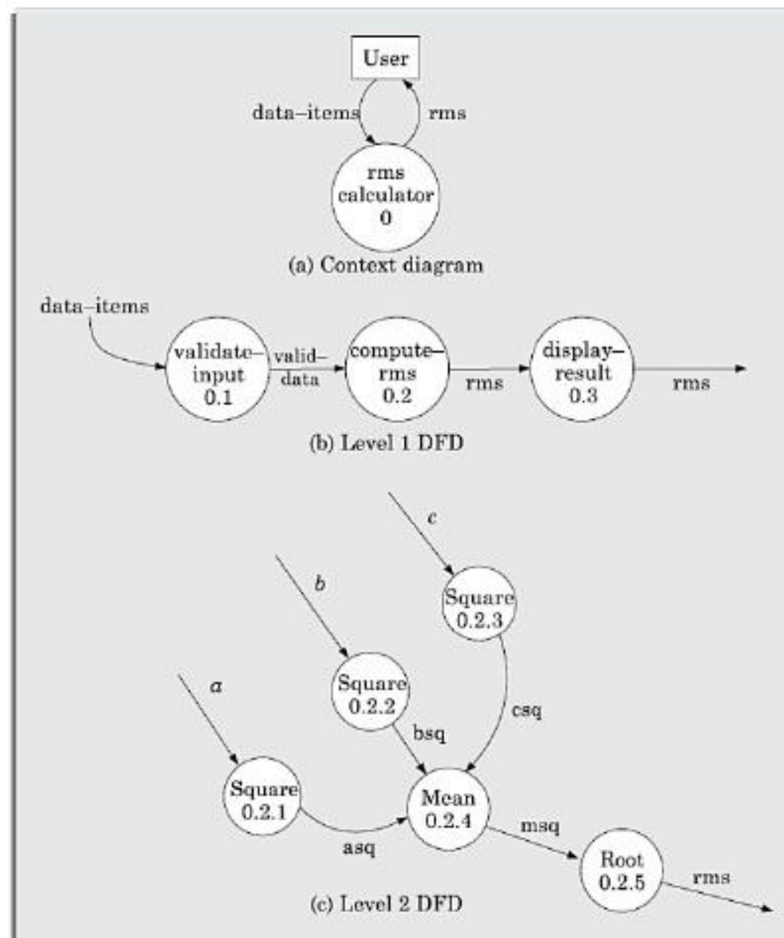
Ilustrasi 3. Jika gelembung A memanggil gelembung B atau gelembung C tergantung pada beberapa kondisi, kita hanya perlu merepresentasikan data yang mengalir antara gelembung A dan B atau gelembung A dan C dan bukan kondisi yang bergantung pada dua modul dipanggil.

- Panah aliran data tidak boleh menghubungkan dua penyimpanan data atau bahkan penyimpanan data dengan entitas eksternal. Dengan demikian, data tidak dapat mengalir dari penyimpanan data ke penyimpanan data lain atau ke entitas eksternal tanpa pemrosesan intervensi. Akibatnya, penyimpanan data harus terhubung hanya ke gelembung melalui panah aliran data.
- Semua fungsi sistem harus ditangkap oleh model DFD. Tidak ada fungsi sistem yang ditentukan dalam dokumen SRS sistem yang harus diabaikan.
- Hanya fungsi-fungsi sistem yang ditentukan dalam dokumen SRS yang harus diwakili. Artinya, perancang tidak boleh mengasumsikan fungsionalitas sistem yang tidak ditentukan oleh dokumen SRS dan kemudian mencoba merepresentasikannya dalam DFD.
- Kamus data yang tidak lengkap dan kamus data yang menunjukkan komposisi item data yang salah adalah kesalahan lain yang sering dilakukan.
- Nama data dan fungsi harus intuitif. Beberapa mahasiswa dan bahkan developer yang berlatih menggunakan nama data simbolis yang tidak berarti seperti a,b,c, dll. Nama seperti itu menghalangi pemahaman model DFD.
- Pemula biasanya mengacaukan DFD mereka dengan terlalu banyak panah aliran data. Menjadi sulit untuk memahami DFD jika ada gelembung yang dikaitkan dengan lebih dari tujuh aliran data. Ketika ada terlalu banyak data yang mengalir masuk atau keluar dari DFD, lebih baik menggabungkan item data ini menjadi item data tingkat tinggi. Gambar 6.7 menunjukkan contoh tentang bagaimana DFD

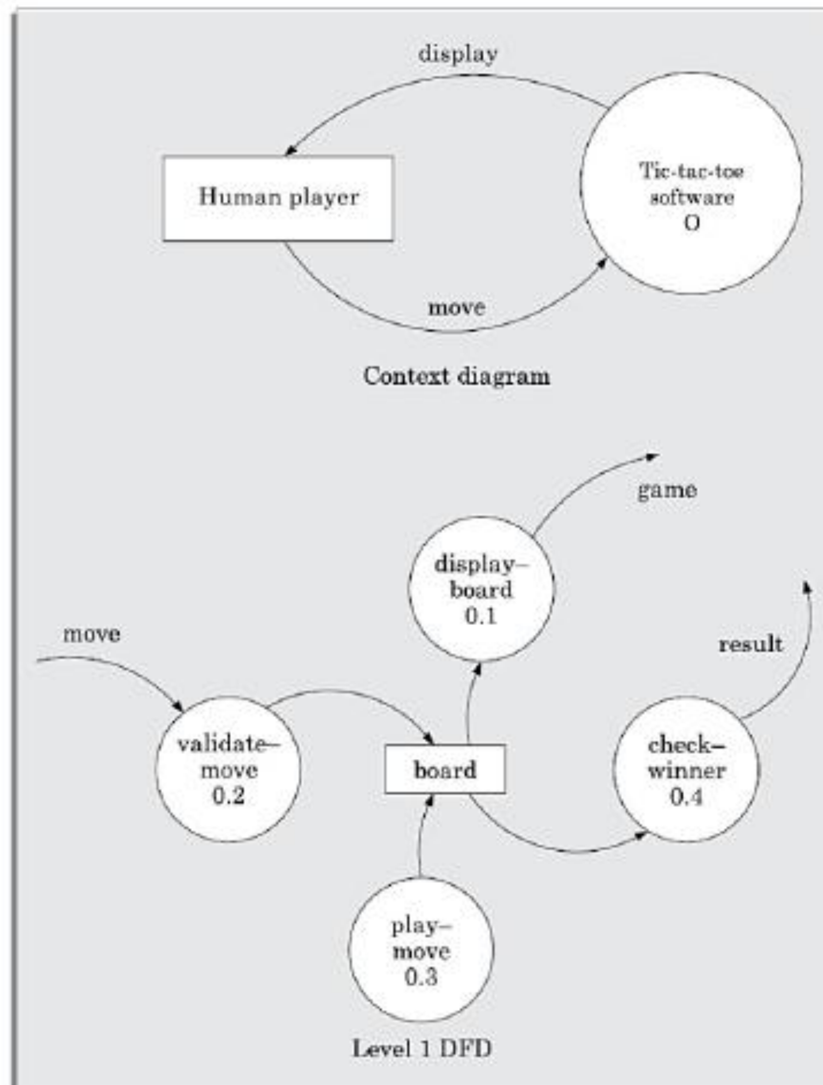
dapat disederhanakan dengan menggabungkan beberapa aliran data menjadi satu aliran data tingkat tinggi.

Contoh 6.1 (Perangkat Lunak Penghitung RMS)

Sistem perangkat lunak yang disebut perangkat lunak penghitung RMS akan membaca tiga bilangan integral dari pengguna dalam kisaran –1000 dan +1000 dan akan menentukan akar rata-rata kuadrat (RMS) dari tiga angka masukan dan menampilkannya. Dalam contoh ini, diagram konteks mudah untuk digambar. Sistem menerima tiga bilangan bulat dari pengguna dan mengembalikan hasilnya kepadanya. Ini telah ditunjukkan pada Gambar 6.8(a). Untuk menggambar DFD level 1, dari analisis sepintas deskripsi masalah, kita dapat melihat bahwa ada empat fungsi dasar yang perlu dilakukan sistem—menerima angka input dari pengguna, memvalidasi angka, menghitung akar rata-rata kuadrat dari nomor input dan, kemudian menampilkan hasilnya. Setelah merepresentasikan keempat fungsi ini pada Gambar 6.8(b), kita mengamati bahwa perhitungan kuadrat rata-rata akar pada dasarnya terdiri dari fungsi-fungsi—hitung kuadrat dari angka-angka input, hitung rata-rata, dan akhirnya hitung akarnya. Dekomposisi ini ditunjukkan pada DFD level 2 pada Gambar 6.8(c).



Gambar 6.8 Diagram konteks, DFD level 1, dan level 2 untuk Contoh 6.1.



Gambar 6.9 Diagram konteks dan DFD level 1 untuk Contoh 6.2.

Kamus data untuk model DFD dari Contoh 6.1

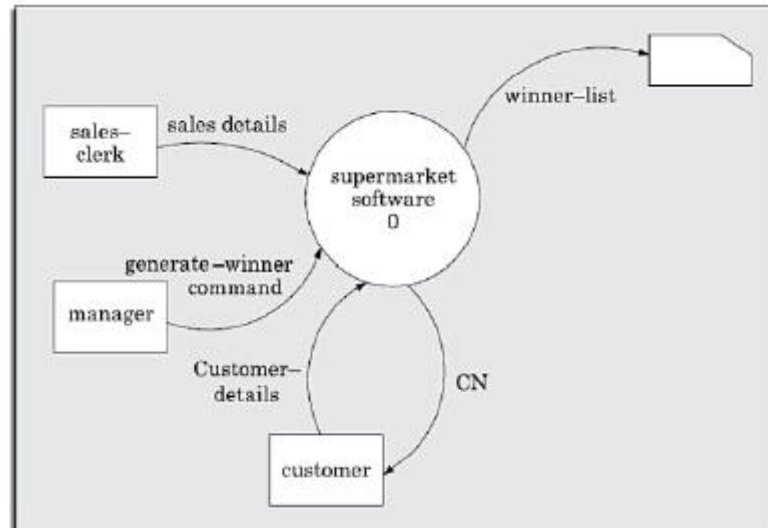
data-items: {integer}3
rms: float
valid-data:data-items
a: integer
b: integer
c: integer
asq: integer
bsq: integer
csq: integer
msq: integer

Contoh 6.1 adalah contoh yang hampir sepele dan hanya dimaksudkan untuk menggambarkan metodologi dasar. Sekarang, mari kita lakukan analisis terstruktur untuk masalah yang lebih kompleks.

Contoh 6.2 (Permainan Komputer Tic-Tac-Toe)

Tic-tac-toe adalah permainan komputer di mana seorang pemain manusia dan komputer melakukan gerakan bergantian pada kotak 3 × 3. Sebuah langkah terdiri dari menandai kotak yang sebelumnya tidak bertanda. Pemain yang pertama kali menempatkan

tiga tanda berturut-turut di sepanjang garis lurus (yaitu, di sepanjang baris, kolom, atau diagonal) di kotak menang. Segera setelah salah satu pemain manusia atau komputer menang, pesan ucapan selamat kepada pemenang akan ditampilkan. Jika tidak ada pemain yang berhasil mendapatkan tiga nilai berturut-turut di sepanjang garis lurus, dan semua kotak di papan terisi, maka permainan ditarik. Komputer selalu mencoba untuk memenangkan permainan. Diagram konteks dan DFD level 1 ditunjukkan pada Gambar 6.9.



Gambar 6.10 Diagram konteks untuk Contoh 6.3.

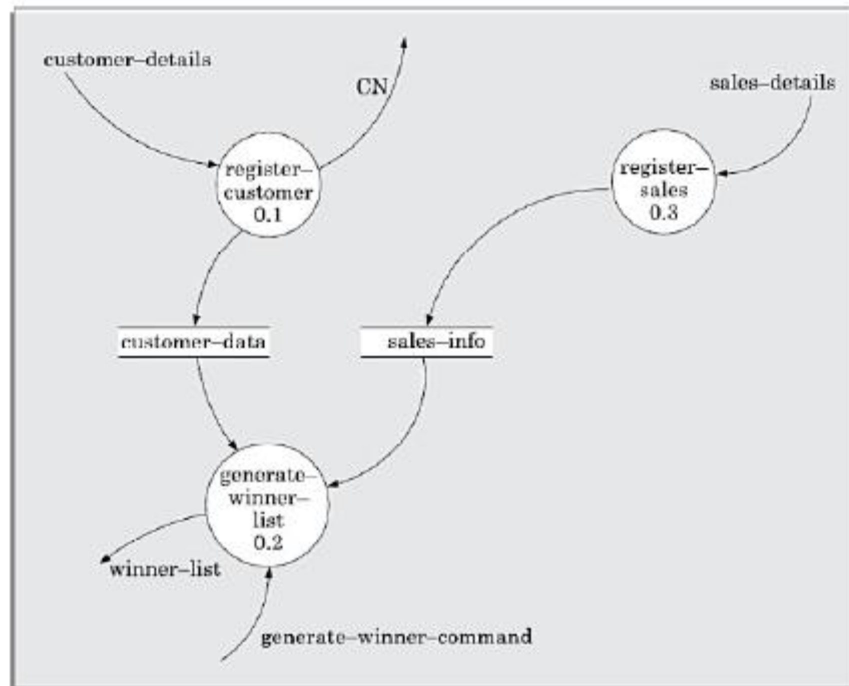
Kamus data untuk model DFD dari Contoh 6.2

move: integer /* number between 1 to 9 */
display: game+result
game: board
board: {integer}9
result: [“computer won”, “human won”, “drawn”]

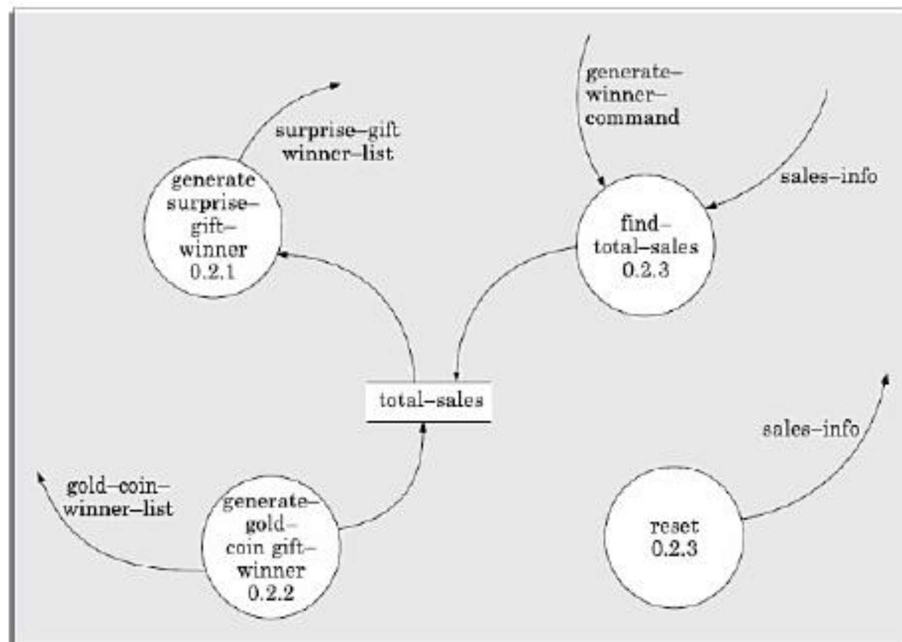
Contoh 6.3 (Skema Hadiah Supermarket)

Pasar super perlu mengembangkan perangkat lunak yang akan membantunya mengotomatiskan skema yang rencananya akan diperkenalkan untuk mendorong pelanggan tetap. Dalam skema ini, pelanggan harus terlebih dahulu mendaftar dengan memberikan alamat tempat tinggal, nomor telepon, dan nomor SIM. Setiap pelanggan yang mendaftar untuk skema ini diberi nomor pelanggan unik (CN) oleh komputer. Pelanggan dapat menunjukkan CN-nya kepada staf check out saat melakukan pembelian. Dalam hal ini, nilai pembeliannya dikreditkan terhadap CN-nya. Setiap akhir tahun, supermarket bermaksud untuk memberikan hadiah kejutan kepada 10 pelanggan yang melakukan pembelian total tertinggi sepanjang tahun. Juga, itu bermaksud untuk memberikan koin emas 22 caret kepada setiap pelanggan yang pembeliannya melebihi Rp. 10.000. Entri terhadap CN diatur ulang pada hari terakhir setiap tahun setelah daftar pemenang hadiah dibuat.

Diagram konteks untuk masalah skema hadiah supermarket pada Contoh 6.3 ditunjukkan pada Gambar 6.10. DFD level 1 pada Gambar 6.11. DFD level 2 pada Gambar 6.12.



Gambar 6.11 Diagram Level 1 untuk Contoh 6.3.



Gambar 6.12 Diagram Level 2 untuk Contoh 6.3.

Kamus data untuk model DFD dari Contoh 6.3

address: name+house#+street#+city+pin
sales-details: {item+amount}* + CN
CN: integer
customer-data: {address+CN}*
sales-info: {sales-details}*
winner-list: surprise-gift-winner-list + gold-coin-winner-list
surprise-gift-winner-list: {address+CN}*
gold-coin-winner-list: {address+CN}*

gen-winner-command: command
total-sales: {CN+integer}*

Pengamatan: Pengamatan berikut dapat dilakukan dari Contoh 6.3.

1. Fakta bahwa pelanggan diberikan kartu identitas pelanggan yang disiapkan secara manual atau bahwa pelanggan menyerahkan kartu identitas setiap kali melakukan pembelian, tidak ditunjukkan dalam DFD. Ini karena ini adalah transfer item yang terjadi di luar komputer.
2. Data generate-winner-list dengan cara mewakili informasi kontrol (yaitu, perintah ke perangkat lunak) dan tidak ada data nyata. Kita sudah memasukkannya ke dalam DFD karena menyederhanakan proses desain terstruktur akan memecahkan beberapa masalah. Kita juga bisa melakukannya tanpa data *generate-winner-list*, tetapi ini bisa membuat desainnya sedikit rumit.
3. Perhatikan pada Gambar 6.11 bahwa kita memiliki dua toko terpisah untuk data pelanggan dan data penjualan. Haruskah kita menggabungkannya menjadi satu penyimpanan data? Jawabannya adalah—Tidak, seharusnya tidak. Jika kita menggabungkannya menjadi satu penyimpanan data, desain terstruktur yang akan dilakukan berdasarkan model ini akan menjadi rumit. Data pelanggan dan data penjualan memiliki karakteristik yang sangat berbeda. Misalnya, data pelanggan setelah dibuat, tidak berubah. Di sisi lain, data penjualan sering berubah dan juga data penjualan direset pada akhir tahun, sedangkan data pelanggan tidak.

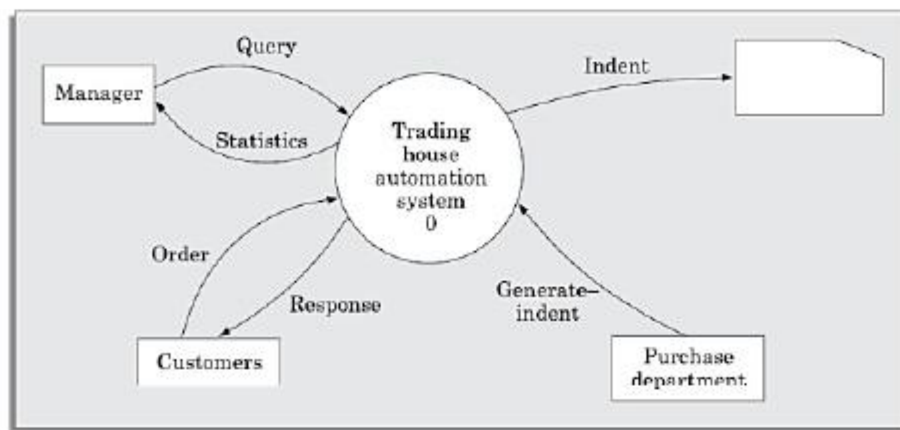
Contoh 6.4 (Trading-house Automation System (TAS))

Sebuah trading house ingin kita mengembangkan sistem komputerisasi yang akan mengotomatisasi berbagai kegiatan pembukuan yang terkait dengan bisnisnya. Berikut ini adalah fitur-fitur yang menonjol dari sistem yang akan dikembangkan:

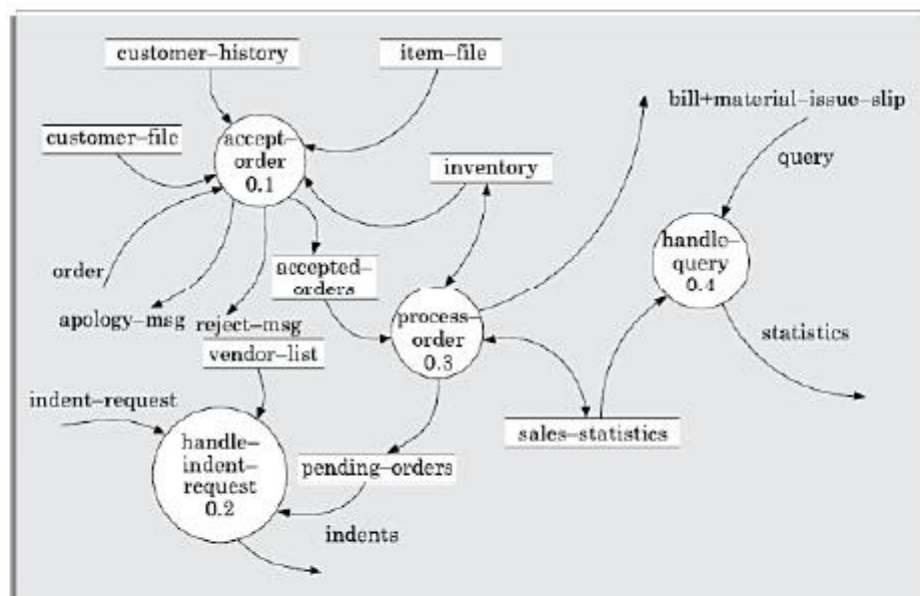
- Rumah perdagangan memiliki satu set pelanggan tetap. Pelanggan memesan dengan itu untuk berbagai jenis komoditas. Rumah perdagangan mempertahankan nama dan alamat pelanggan tetapnya. Masing-masing pelanggan tetap ini harus diberi nomor identifikasi pelanggan unik (CIN) oleh komputer. Pelanggan mengutip CIN mereka pada setiap pesanan yang mereka lakukan.
- Setelah pesanan dilakukan, sesuai praktik saat ini, departemen rekening rumah perdagangan pertama-tama memeriksa kelayakan kredit pelanggan. Kelayakan kredit pelanggan ditentukan dengan menganalisis riwayat pembayarannya ke berbagai tagihan yang dikirimkan kepadanya di masa lalu. Setelah otomatisasi, tugas ini telah dilakukan oleh komputer.
- Jika pelanggan tidak layak kredit, pesanannya tidak diproses lebih lanjut dan pesan penolakan pesanan yang sesuai dibuat untuk pelanggan.
- Jika pelanggan layak kredit, barang yang dipesannya diperiksa dengan daftar barang yang berurusan dengan rumah perdagangan. Item dalam urutan yang tidak ditangani oleh rumah perdagangan, tidak diproses lebih lanjut dan pesan permintaan maaf yang sesuai untuk pelanggan untuk item ini dibuat.
- Item dalam pesanan pelanggan yang berurusan dengan rumah perdagangan diperiksa ketersediaannya di inventaris. Jika barang tersedia dalam persediaan dalam jumlah yang diinginkan, maka:
    - Tagihan dengan alamat penerusan pelanggan dicetak.
    - Slip masalah material dicetak. Pelanggan dapat membuat slip masalah material ini di gudang dan menerima pengiriman barang.
    - Data persediaan disesuaikan untuk mencerminkan penjualan ke pelanggan.

- Jika salah satu barang yang dipesan tidak tersedia dalam persediaan dalam jumlah yang cukup untuk memenuhi pesanan, maka barang-barang yang kehabisan stok ini bersama dengan jumlah yang dipesan oleh pelanggan dan CIN disimpan dalam file "pesanan tertunda" untuk pemrosesan lebih lanjut yang akan dilakukan ketika departemen pembelian mengeluarkan perintah "generate indent".
- Departemen pembelian harus diizinkan untuk mengeluarkan perintah secara berkala untuk menghasilkan indentasi. Ketika perintah untuk menghasilkan indentasi dikeluarkan, sistem harus memeriksa file "pesanan tertunda" untuk menentukan pesanan yang tertunda dan menentukan jumlah total yang diperlukan untuk setiap item. Itu harus mengetahui alamat vendor yang memasok barang-barang ini dengan memeriksa file yang berisi detail vendor dan kemudian harus mencetak indentasi ke vendor ini.
- Sistem juga harus menjawab pertanyaan manajerial mengenai statistik berbagai item yang terjual selama periode waktu tertentu dan kuantitas yang sesuai yang terjual dan harga yang direalisasikan.

Diagram konteks untuk masalah otomatisasi rumah perdagangan ditunjukkan pada Gambar 6.13. DFD level 1 pada Gambar 6.14.



Gambar 6.13 Diagram konteks untuk Contoh 6.4.



Gambar 6.14 DFD Level 1 untuk Contoh 6.4.

Kamus data untuk model DFD dari Contoh 6.4

response: [bill + material-issue-slip, reject-msg,apology-msg]
query: period /* query from manager regarding sales statistics*/
period: [date+date,month,year,day]
date: year + month + day year: integer
month: integer day: integer customer-id: integer
order: customer-id + {items + quantity}* + order#
accepted-order: order /* ordered items available in inventory */
reject-msg: order + message /* rejection message */
pending-orders: customer-id + order# + {items+quantity}*
customer-address: name+house#+street#+city+pin
name: string
house#: string
street#: string
city: string
pin: integer
customer-id: integer
customer-file: {customer-address}* + customer-id
bill: {item + quantity + price}* + total-amount + customer-address + order#
material-issue-slip: message + item + quantity + customer-address
message: string
statistics: {item + quantity + price }*
sales-statistics: {statistics}* + date
quantity: integer
order#: integer /* unique order number generated by the program */
price: integer
total-amount: integer
generate-indent: command
indent: {item+quantity}* + vendor-address
indents: {indent}*
vendor-address: customer-address
vendor-list: {vendor-address}*
item-file: {item}*
item: string
indent-request: command

Pengamatan: Pengamatan berikut dapat dilakukan dari Contoh 6.4.

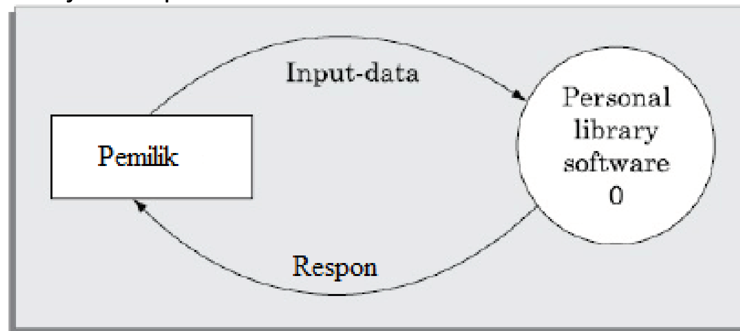
1. Dalam DFD, jika dua penyimpanan data menangani tipe data yang berbeda, mis. satu jenis data invarian dengan waktu sedangkan yang lain bervariasi dengan waktu, (misalnya alamat vendor, dan data inventaris) adalah ide yang baik untuk mewakili mereka sebagai penyimpanan data yang terpisah. Jika dua jenis data selalu diperbarui pada saat yang sama, mereka harus disimpan dalam satu penyimpanan data. Jika tidak, penyimpanan data terpisah harus digunakan untuk mereka. Data inventaris berubah setiap kali pasokan tiba dan inventaris diperbarui atau item dijual, sedangkan data vendor tetap tidak berubah.
2. Jika kita sedang mengembangkan model DFD dari suatu proses yang sudah dilakukan secara manual, maka nama-nama register yang dipertahankan dalam proses manual akan muncul sebagai penyimpanan data dalam model DFD.

Misalnya, jika TAS saat ini sedang dilakukan secara manual, maka biasanya akan ada register yang sesuai dengan pesanan yang diterima, pesanan yang tertunda, daftar vendor, dll.

3. Kita dapat mengamati bahwa DFD memungkinkan developer perangkat lunak untuk mengembangkan domain data dan model domain fungsional sistem secara bersamaan. Saat DFD disempurnakan menjadi tingkat detail yang lebih besar, analis melakukan dekomposisi fungsional implisit. Pada saat yang sama, penyempurnaan DFD secara otomatis menghasilkan penyempurnaan item data yang sesuai.
4. Data yang disimpan dalam register fisik dalam pemrosesan manual, menjadi penyimpanan data dalam representasi DFD. Oleh karena itu, untuk menentukan data mana yang harus direpresentasikan sebagai penyimpanan data, akan berguna untuk mencoba membayangkan apakah satu set item data akan dipertahankan dalam register dalam sistem manual.

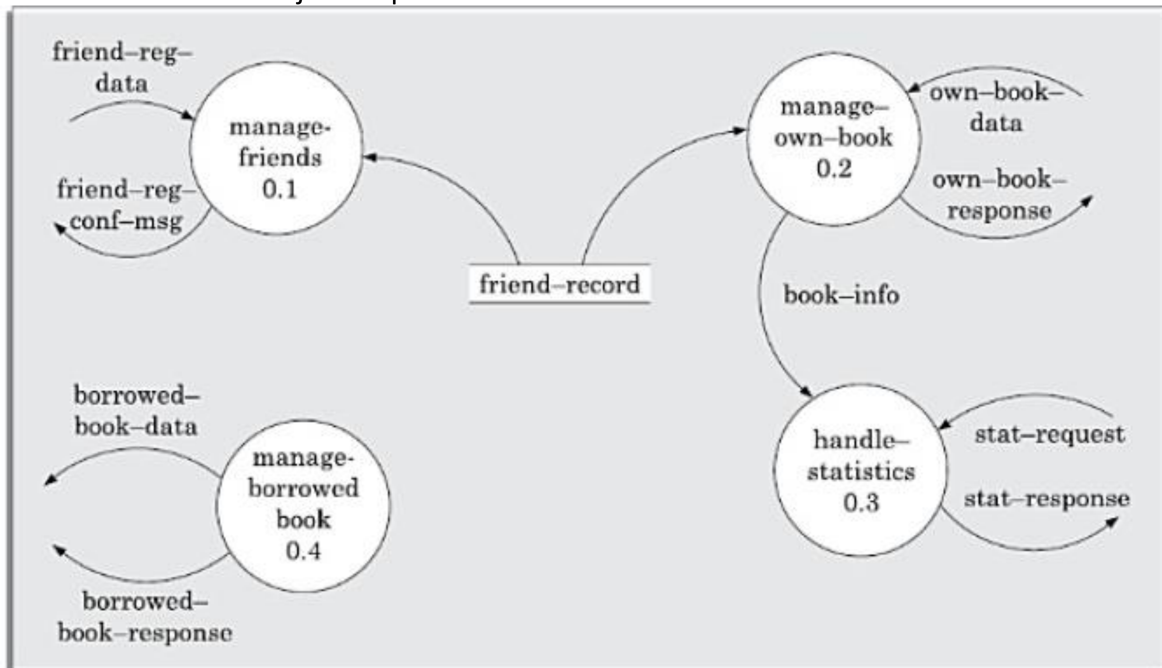
Contoh 6.5 (Perangkat Lunak Perpustakaan Pribadi)

Lakukan analisis terstruktur untuk perangkat lunak perpustakaan pribadi Contoh 6.5. Diagram konteks ditunjukkan pada Gambar 6.15.



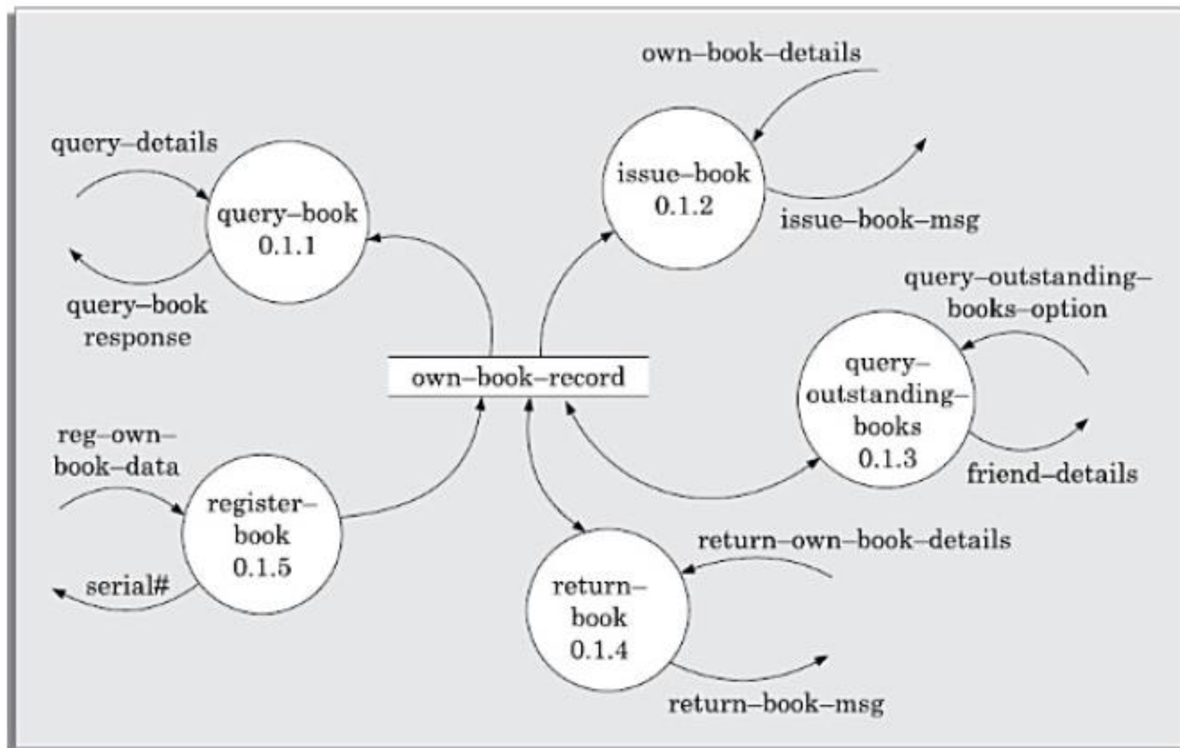
Gambar 6.15 Diagram konteks untuk Contoh 6.5.

DFD level 1 ditunjukkan pada Gambar 6.16.



Gambar 6.16 DFD Level 1 untuk Contoh 6.5.

DFD level 2 untuk gelembung manageOwnBook ditunjukkan pada Gambar 6.17.



Gambar 6.17 DFD Level 2 untuk Contoh 6.5.

Kamus data untuk model DFD Contoh 6.5

input-data: friend-reg-data + own-book-data + stat-request + borrowed-book-data
response: friend-reg-conf-msg + own-book-response + stat-response + borrowed-book-response
own-book-data: query-details + own-book-details + query-outstanding-books-option + return-own bookdetails
+ reg-own-book-data
own-book-response: query-book-response + issue-book-msg + friend-details + return-book- msg +
serial#.
borrowed-book-data: borrowed-book-details + book-return-details + display-books-option borrowed-bookresponse:
reg-msg + unreg-msg + borrowed-books-list
friend-reg-data: name + address + landline# + mobile#
own-book-details: friend-reg-data + book-title + data-of-issue
return-own-book-details: book-title + date-of-return
friend-details: name + address + landline# + mobile# + book-list
borrowed-book-details: book-title + borrow-date
serial#: integer

Pengamatan: Perhatikan bahwa karena ada lebih dari tujuh persyaratan fungsional untuk perangkat lunak perpustakaan pribadi, persyaratan terkait telah digabungkan untuk memiliki hanya lima gelembung dalam diagram level 1. Hanya DFD level 2 yang ditampilkan, karena DFD lainnya sepele dan tidak perlu digambar.

Kekurangan model DFD

Model DFD menderita beberapa kekurangan. Kekurangan penting dari model DFD adalah sebagai berikut:

- DFD yang tidak tepat meninggalkan ruang lingkup yang cukup untuk menjadi tidak tepat. Dalam model DFD, fungsi yang dinilai dilakukan oleh gelembung dari labelnya. Namun, label pendek mungkin tidak menangkap seluruh fungsi balon. Misalnya, gelembung bernama find-book-position hanya memiliki makna intuitif dan tidak menentukan beberapa hal, mis. apa yang terjadi ketika beberapa informasi input hilang atau salah. Selanjutnya, gelembung posisi buku mungkin tidak menyampaikan apa pun tentang apa yang terjadi ketika buku yang diperlukan hilang.
- Aspek kontrol yang tidak didefinisikan dengan baik tidak ditentukan oleh DFD. Misalnya, urutan input yang dikonsumsi dan output yang dihasilkan oleh gelembung tidak ditentukan. Model DFD tidak menentukan urutan eksekusi gelembung yang berbeda. Representasi aspek tersebut sangat penting untuk pemodelan sistem real-time.
- Dekomposisi: Metode melakukan dekomposisi untuk sampai pada tingkat yang berurutan dan tingkat akhir di mana dekomposisi dilakukan sangat subjektif dan bergantung pada pilihan dan penilaian analis. Karena alasan ini, bahkan untuk masalah yang sama, beberapa representasi DFD alternatif dimungkinkan. Lebih lanjut, sering kali tidak mungkin untuk mengatakan representasi DFD mana yang lebih unggul atau lebih disukai daripada yang lain.
- Diagram aliran data yang tidak tepat: Teknik diagram aliran data tidak memberikan panduan khusus tentang bagaimana tepatnya menguraikan fungsi yang diberikan ke dalam subfungsinya dan kita harus menggunakan penilaian subjektif untuk melakukan dekomposisi.

Memperluas Teknik DFD agar Dapat Berlaku untuk Sistem Real-time

Dalam sistem waktu nyata, beberapa fungsi tingkat tinggi dikaitkan dengan tenggat waktu. Oleh karena itu, suatu fungsi tidak hanya harus menghasilkan hasil yang benar tetapi juga harus menghasilkannya dengan waktu yang telah ditentukan sebelumnya. Untuk sistem waktu nyata, waktu eksekusi merupakan pertimbangan penting untuk mencapai desain yang benar. Oleh karena itu, representasi eksplisit dari aspek kontrol dan aliran peristiwa sangat penting. Salah satu teknik yang diterima secara luas untuk memperluas teknik DFD ke analisis sistem real-time adalah teknik Ward dan Mellor [1985]. Dalam notasi Ward dan Mellor, jenis proses yang hanya menangani aliran kontrol diperkenalkan. Proses ini mewakili pemrosesan kontrol dilambangkan dengan gelembung putus-putus. Alur kontrol ditampilkan menggunakan garis putus-putus/panah.

Tidak seperti Ward dan Mellor, Hatley dan Pirbhai [1987] menunjukkan representasi putus-putus dan solid pada diagram terpisah. Untuk dapat memisahkan data dan aspek pemrosesan kontrol, diagram alir kontrol (CFD) didefinisikan. Ini mengurangi kompleksitas diagram. Untuk menghubungkan pemrosesan data dan diagram pemrosesan kontrol, referensi notasi (batang padat) ke spesifikasi kontrol digunakan. CSPEC menjelaskan sebagai berikut:

- Efek dari peristiwa eksternal atau sinyal kontrol.
- Proses yang dipanggil sebagai konsekuensi dari suatu peristiwa.

Spesifikasi kontrol mewakili perilaku sistem dalam dua cara berbeda:

- Ini berisi diagram transisi keadaan (STD). STD adalah spesifikasi perilaku yang berurutan.

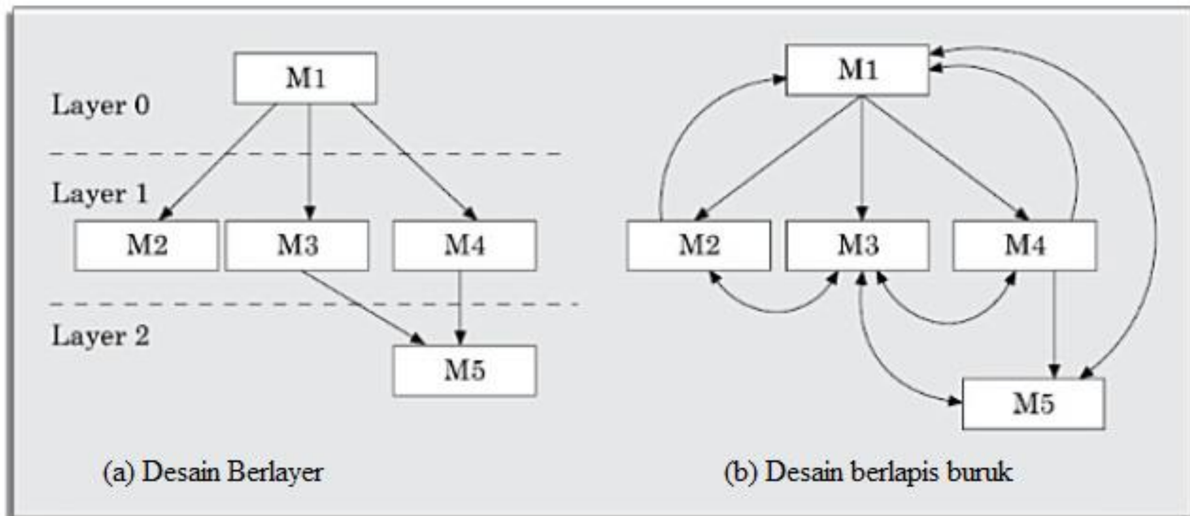
- Ini berisi tabel aktivasi pro gram (PAT). PAT adalah spesifikasi kombinatorial perilaku. PAT mewakili urutan doa gelembung dalam DFD.

6.4 DESAIN TERSTRUKTUR

Tujuan dari desain terstruktur adalah untuk mengubah hasil analisis terstruktur (yaitu, model DFD) ke dalam bagan struktur. Bagan struktur mewakili rancangan roftware. Berbagai modul yang membentuk sistem, ketergantungan modul (yaitu modul mana yang memanggil modul lain), dan parameter yang dilewatkan di antara modul yang berbeda. Representasi bagan struktur dapat dengan mudah diimplementasikan menggunakan beberapa bahasa pemrograman. Karena fokus utama dalam representasi bagan struktur adalah pada struktur modul perangkat lunak dan interaksi di antara modul yang berbeda, aspek prosedural (misalnya bagaimana fungsionalitas tertentu dicapai) tidak terwakili. Blok bangunan dasar yang menggunakan bagan struktur yang dirancang adalah sebagai berikut:

- **Kotak persegi panjang**: Kotak persegi panjang mewakili modul. Biasanya, setiap kotak persegi panjang diberi keterangan dengan nama modul yang diwakilinya.
- **Panah pemanggilan modul**: Panah yang menghubungkan dua modul menyiratkan bahwa selama eksekusi program, kontrol dilewatkan dari satu modul ke modul lainnya ke arah panah penghubung. Namun, hanya dengan melihat bagan struktur, kita tidak dapat mengatakan apakah suatu modul memanggil modul lain hanya sekali atau berkali-kali. Selain itu, hanya dengan melihat bagan struktur, kita tidak dapat mengetahui urutan pemanggilan modul yang berbeda.
- **Panah aliran data**: Ini adalah panah kecil yang muncul di samping panah pemanggilan modul. Panah aliran data dianotasi dengan nama data yang sesuai. Panah aliran data merepresentasikan fakta bahwa data yang diberi nama berpindah dari satu modul ke modul lainnya dalam arah panah.
- **Modul perpustakaan**: Modul perpustakaan biasanya diwakili oleh persegi panjang dengan tepi ganda. Perpustakaan terdiri dari modul yang sering disebut. Biasanya, ketika sebuah modul dipanggil oleh banyak modul lain, modul itu dibuat menjadi modul perpustakaan.
- **Pilihan**: Simbol berlian mewakili fakta bahwa satu modul dari beberapa modul yang terhubung dengan simbol berlian dipanggil tergantung pada hasil dari kondisi yang dilampirkan dengan simbol berlian.
- **Pengulangan**: Sebuah loop di sekitar panah aliran kontrol menunjukkan bahwa modul masing-masing dipanggil berulang kali.

Dalam bagan struktur apa pun, harus ada satu dan hanya satu modul di bagian atas, yang disebut root. Harus ada paling banyak satu hubungan kontrol antara dua modul dalam bagan struktur. Ini berarti bahwa jika modul A memanggil modul B, modul B tidak dapat memanggil modul A. Alasan utama di balik pembatasan ini adalah bahwa kita dapat mempertimbangkan modul yang berbeda dari bagan struktur untuk diatur dalam lapisan atau level. Prinsip abstraksi tidak memungkinkan modul tingkat rendah untuk menyadari keberadaan modul tingkat tinggi. Namun, dimungkinkan untuk dua modul tingkat yang lebih tinggi untuk memanggil modul tingkat yang lebih rendah yang sama. Contoh desain dengan lapisan yang benar dan desain lapisan yang buruk lainnya ditunjukkan pada Gambar 6.18.



Gambar 6.18 Contoh desain berlapis yang benar dan yang buruk.

Bagan alir versus bagan struktur

Kita semua akrab dengan representasi diagram alir dari suatu program. Diagram alir adalah teknik yang nyaman untuk mewakili aliran kontrol dalam suatu program. Bagan struktur berbeda dari bagan alir dalam tiga cara utama:

- Biasanya sulit untuk mengidentifikasi modul yang berbeda dari sebuah program dari representasi diagram alirnya.
- Pertukaran data antar modul yang berbeda tidak direpresentasikan dalam diagram alir.
- Urutan berurutan tugas yang melekat pada diagram alur ditekan dalam bagan struktur.

Transformasi Model DFD menjadi Bagan Struktur

Teknik sistematis tersedia untuk mengubah representasi DFD dari suatu masalah ke dalam struktur modul yang diwakili oleh bagan struktur. Desain terstruktur menyediakan dua strategi untuk memandu transformasi DFD menjadi bagan struktur:

- Analisis transformasi
- Analisis transaksi

Biasanya, seseorang akan mulai dengan DFD level 1, mengubahnya menjadi representasi modul menggunakan analisis transformasi atau transaksi dan kemudian melanjutkan ke DFD level yang lebih rendah. Pada setiap tingkat transformasi, penting untuk terlebih dahulu menentukan apakah transformasi atau analisis transaksi dapat diterapkan pada DFD tertentu.

Apakah akan menerapkan transformasi atau pemrosesan transaksi?

Mengingat DFD model tertentu, bagaimana seseorang memutuskan apakah akan menerapkan analisis transformasi atau analisis transaksi? Untuk ini, seseorang harus memeriksa input data ke diagram. Input data ke diagram dapat dengan mudah terlihat karena diwakili oleh panah yang menjuntai. Jika semua aliran data ke dalam diagram diproses dengan cara yang sama (yaitu jika semua panah aliran data input terjadi pada gelembung yang sama di DFD) maka analisis transformasi dapat diterapkan. Jika tidak, analisis transaksi dapat diterapkan. Biasanya, analisis transformasi hanya berlaku untuk pemrosesan yang sangat sederhana.

Harap diingat bahwa gelembung-gelembung tersebut diurai hingga mewakili pemrosesan yang sangat sederhana yang dapat diimplementasikan hanya dengan

menggunakan beberapa baris kode. Oleh karena itu, analisis transformasi biasanya dapat diterapkan pada tingkat model DFD yang lebih rendah. Setiap cara berbeda di mana data diproses sesuai dengan transaksi terpisah. Setiap transaksi sesuai dengan fungsionalitas yang memungkinkan pengguna melakukan pekerjaan yang berarti menggunakan perangkat lunak.

Analisis transformasi

Analisis transformasi mengidentifikasi komponen fungsional utama (modul) dan data input dan output untuk komponen ini. Langkah pertama dalam analisis transformasi adalah membagi DFD menjadi tiga jenis bagian:

- Memasukkan.
- Pemrosesan.
- Keluaran.

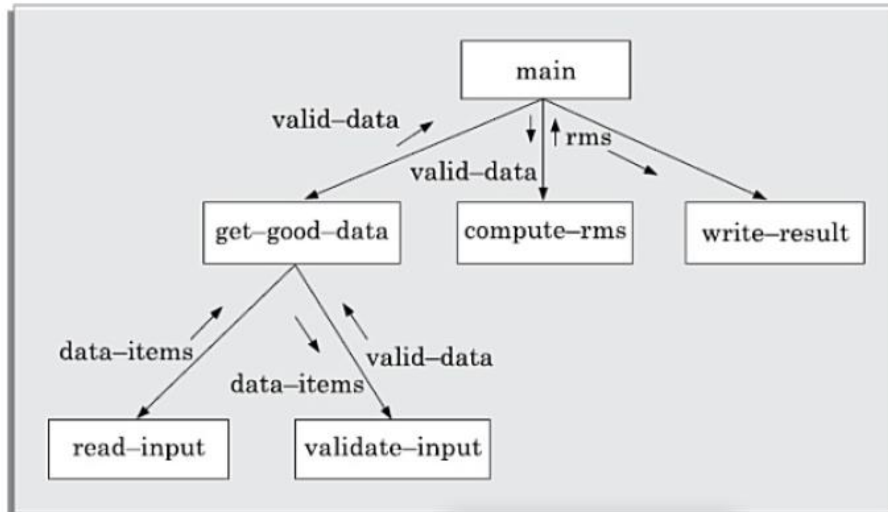
Bagian input dalam DFD mencakup proses yang mengubah data input dari fisik (misalnya, karakter dari terminal) ke bentuk logis (misalnya tabel internal, daftar, dll.). Setiap bagian input disebut cabang aferen. Bagian output dari DFD mengubah data output dari bentuk logis ke bentuk fisik. Setiap bagian keluaran disebut cabang eferen. Bagian yang tersisa dari DFD disebut transformasi pusat. Pada langkah analisis transformasi selanjutnya, bagan struktur diturunkan dengan menggambar satu komponen fungsional masing-masing untuk transformasi pusat, cabang aferen dan eferen. Ini digambar di bawah modul root, yang akan memanggil modul-modul ini.

Mengidentifikasi bagian input dan output membutuhkan pengalaman dan keterampilan. Salah satu pendekatan yang mungkin adalah menelusuri data input sampai ditemukan gelembung yang data outputnya tidak dapat disimpulkan dari inputnya saja. Proses yang memvalidasi input bukanlah transformasi sentral. Proses yang mengurutkan input atau memfilter data darinya adalah transformasi pusat. Bagan struktur tingkat pertama dihasilkan dengan merepresentasikan setiap unit input dan output sebagai sebuah kotak dan setiap transformasi pusat sebagai sebuah kotak tunggal.

Pada langkah ketiga analisis transformasi, bagan struktur disempurnakan dengan menambahkan subfungsi yang diperlukan oleh masing-masing komponen fungsional tingkat tinggi. Banyak tingkat komponen fungsional dapat ditambahkan. Proses pemecahan komponen fungsional menjadi subkomponen disebut pemfaktoran. Anjak piutang meliputi penambahan modul baca dan tulis, modul penanganan kesalahan, proses inisialisasi dan penghentian, identifikasi modul konsumen, dll. Proses anjak piutang dilanjutkan sampai semua gelembung di DFD terwakili dalam bagan struktur.

Contoh 6.6 Gambarkan bagan struktur untuk perangkat lunak RMS dari Contoh 6.1.

Dengan mengamati DFD level 1 pada Gambar 6.8, kita dapat mengidentifikasi input-validasi sebagai cabang aferen dan output tulis sebagai cabang eferen. Sisanya (yaitu, compute-rms) sebagai transformasi pusat. Dengan menerapkan langkah 2 dan langkah 3 dari analisis transformasi, kita mendapatkan bagan struktur yang ditunjukkan pada Gambar 6.19.



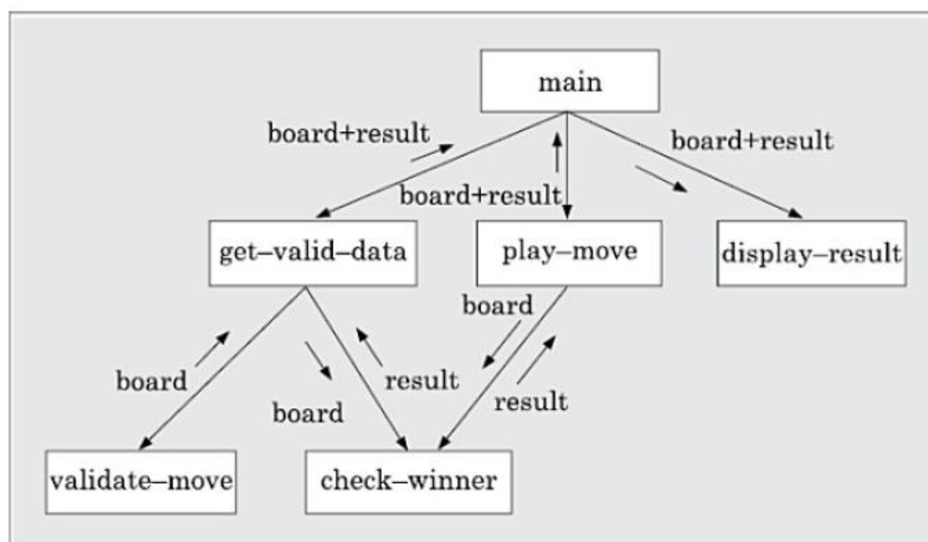
Gambar 6.19 Bagan struktur untuk Contoh 6.6.

Contoh 6.7 Gambarlah bagan struktur untuk perangkat lunak tic-tac-toe dari Contoh 6.2.

Bagan struktur perangkat lunak Tic-tac-toe ditunjukkan pada Gambar 6.20. Perhatikan bahwa gelembung cek-permainan-status, meskipun menghasilkan beberapa keluaran tidak benar-benar bertanggung jawab untuk mengubah data logis menjadi data fisik. Di sisi lain, ia melakukan pemrosesan yang melibatkan pengecekan status permainan. Itulah alasan utama, mengapa kita harus menganggapnya sebagai transformasi sentral dan bukan sebagai modul jenis eferen.

Analisis transaksi

Analisis transaksi merupakan alternatif untuk mengubah analisis dan berguna saat merancang program pemrosesan transaksi. Transaksi memungkinkan pengguna untuk melakukan beberapa jenis pekerjaan tertentu dengan menggunakan perangkat lunak. Misalnya, 'buku terbitan', 'buku kembali', 'buku kueri', dll., adalah transaksi.



Gambar 6.20 Bagan struktur untuk Contoh 6.7.

Seperti dalam analisis transformasi, pertama-tama semua data yang masuk ke DFD perlu diidentifikasi. Dalam sistem yang digerakkan oleh transaksi, item data yang berbeda dapat melewati jalur komputasi yang berbeda melalui DFD. Ini berbeda dengan sistem transformasi terpusat di mana setiap item data yang masuk ke DFD melewati langkah

pemrosesan yang sama. Setiap cara yang berbeda di mana data input diproses adalah transaksi. Cara sederhana untuk mengidentifikasi transaksi adalah sebagai berikut. Periksa data masukan. Jumlah gelembung di mana data input ke DFD adalah insiden menentukan jumlah transaksi. Namun, beberapa transaksi mungkin tidak memerlukan data input apa pun. Transaksi ini dapat diidentifikasi berdasarkan pengalaman yang diperoleh dari pemecahan sejumlah besar contoh.

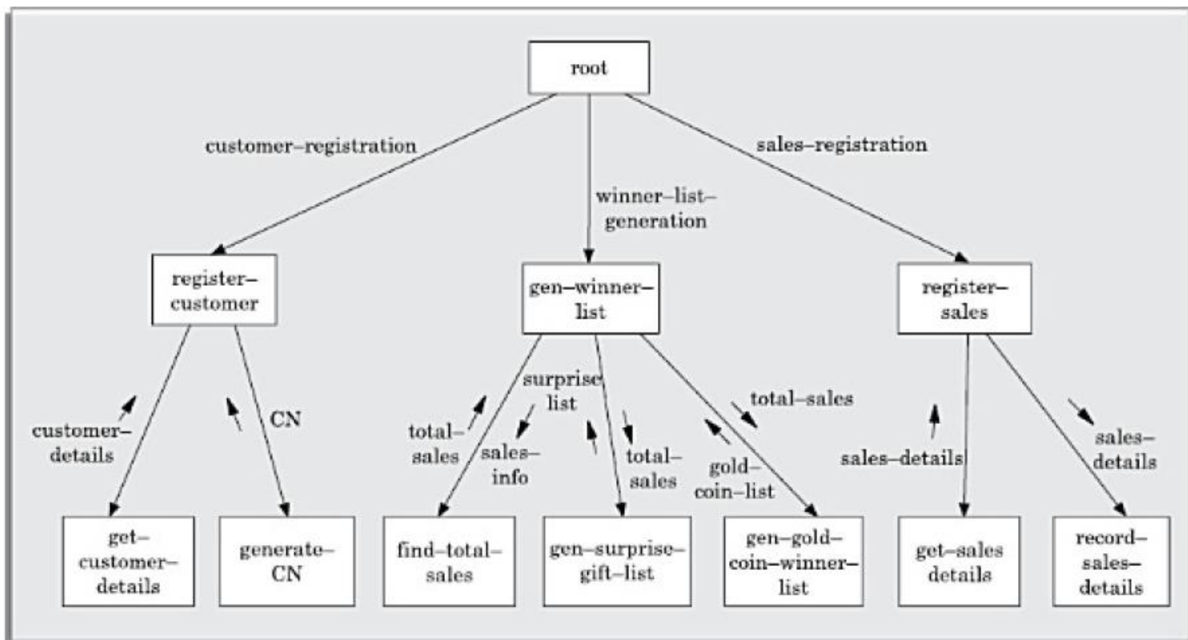
Untuk setiap transaksi yang teridentifikasi, lacak data input ke output. Semua gelembung yang dilalui adalah milik transaksi. Gelembung ini harus dipetakan ke modul yang sama pada bagan struktur. Dalam bagan struktur, gambar modul root dan di bawah modul ini gambar setiap transaksi yang diidentifikasi sebagai modul. Setiap transaksi membawa tag yang mengidentifikasi jenisnya. Analisis transaksi menggunakan tag ini untuk membagi sistem menjadi modul transaksi dan modul pusat transaksi.

Contoh 6.8 Gambarlah bagan struktur untuk perangkat lunak

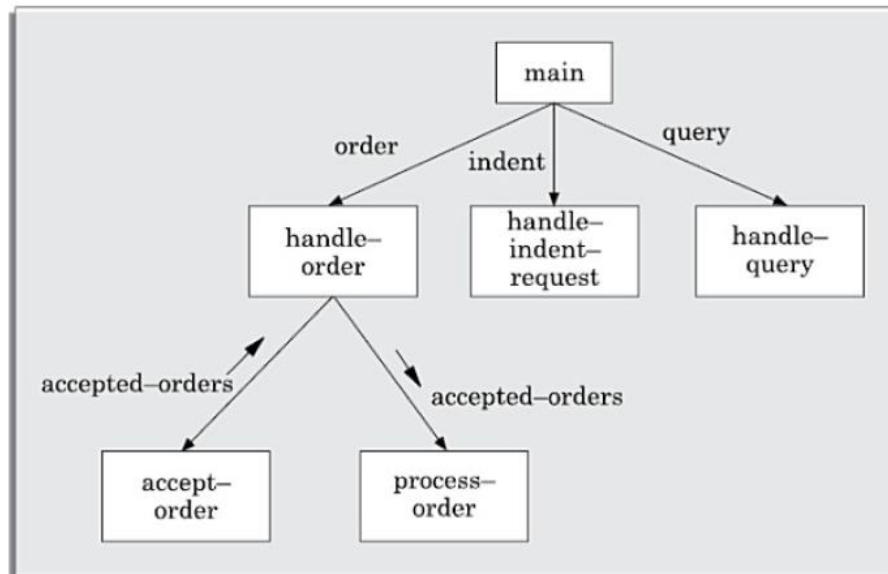
Skema Hadiah Supermarket pada Contoh 6.3. Bagan struktur untuk perangkat lunak Skema Hadiah Supermarket ditunjukkan pada Gambar 6.21.

Contoh 6.9 Gambarlah bagan struktur untuk perangkat lunak sistem otomasi trade-house (TAS) dari Contoh 6.4.

Bagan struktur untuk perangkat lunak sistem otomasi trade-house (TAS) dari Contoh 6.4 ditunjukkan pada Gambar 6.22. Dengan mengamati DFD level 1 pada Gambar 6.14, kita dapat melihat bahwa input data ke diagram ditangani oleh gelembung yang berbeda dan oleh karena itu analisis transaksi dapat diterapkan pada DFD ini. Data input ke DFD ini ditangani dalam tiga cara berbeda (accept-order, accept-indent-request, dan handle-query), tiga transaksi berbeda yang sesuai dengan ini ditunjukkan pada Gambar 6.22.



Gambar 6.21 Bagan struktur untuk Contoh 6.8.



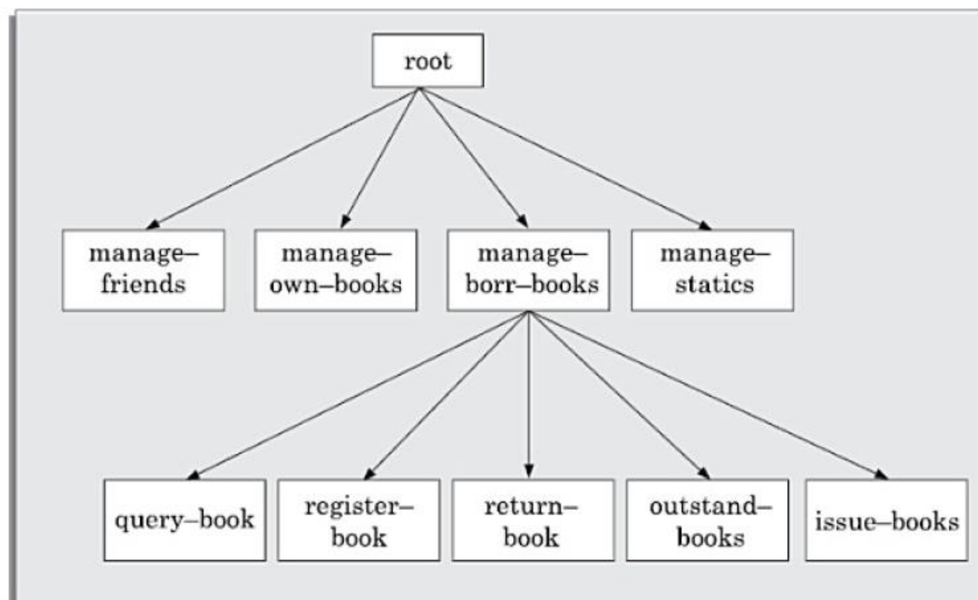
Gambar 6.22 Bagan struktur untuk Contoh 6.9.

Kata hati-hati

Kita harus melihat analisis transformasi dan transaksi sebagai pedoman, bukan aturan. Kita harus menerapkan pedoman ini dalam konteks masalah dan menangani kasus patogen dengan hati-hati.

Contoh 6.10 Gambarlah bagan struktur untuk perangkat lunak perpustakaan pribadi Contoh 6.6.

Bagan struktur untuk perangkat lunak perpustakaan pribadi ditunjukkan pada Gambar 6.23.



Gambar 6.23 Bagan struktur untuk Contoh 6.10.

6.5 DESAIN YANG DETAIL

Selama desain terperinci, deskripsi kode semu dari pemrosesan dan struktur data yang berbeda dirancang untuk modul yang berbeda dari bagan struktur. Ini biasanya dijelaskan dalam bentuk spesifikasi modul (MSPEC). MSPEC biasanya ditulis menggunakan bahasa Inggris terstruktur. MSPEC untuk modul non-daun menjelaskan kondisi yang berbeda di mana

tanggung jawab didelegasikan ke modul tingkat yang lebih rendah. MSPEC untuk modul tingkat daun harus menjelaskan dalam bentuk algoritmik bagaimana langkah-langkah pemrosesan primitif dilakukan. Untuk mengembangkan MSPEC sebuah modul, biasanya perlu mengacu pada model DFD dan dokumen SRS untuk menentukan fungsionalitas modul.

6.6 TINJAUAN DESAIN

Setelah desain selesai, desain harus ditinjau. Tim peninjau biasanya terdiri dari anggota dengan perspektif desain, implementasi, pengujian, dan pemeliharaan, yang mungkin atau mungkin bukan anggota tim pengembangan. Biasanya, anggota tim yang akan membuat kode desain, dan menguji kode, analis, dan pengelola menghadiri pertemuan peninjauan. Tim peninjau memeriksa dokumen desain terutama untuk aspek-aspek berikut:

- **Ketertelusuran**: Apakah setiap gelembung DFD dapat dilacak ke beberapa modul dalam bagan struktur dan sebaliknya. Mereka memeriksa apakah setiap kebutuhan fungsional dalam dokumen SRS dapat ditelusuri ke beberapa gelembung dalam model DFD dan sebaliknya.
- **Kebenaran**: Apakah semua algoritme dan struktur data dari desain detail sudah benar.
- **Maintainability**: Apakah desain dapat dengan mudah dipelihara di masa depan.
- **Implementasi**: Apakah desain dapat dengan mudah dan efisien diimplementasikan.

Setelah poin-poin yang diangkat oleh reviewer ditanggapi oleh desainer, dokumen desain siap untuk diimplementasikan.

6.7 RINGKASAN

- Dalam bab ini, kita membahas metodologi desain perangkat lunak berorientasi fungsi sampel yang disebut analisis terstruktur/desain terstruktur (SA/SD) yang menggabungkan fitur dari beberapa metodologi desain penting.
- Metodologi seperti SA/SD memberi kita resep untuk mengembangkan desain yang baik menurut kriteria kebaikan yang berbeda yang telah kita bahas di Bab 5. item SA/SD terdiri dari dua bagian penting—analisis terstruktur dan desain terstruktur.
- Tujuan dari analisis terstruktur adalah untuk melakukan dekomposisi fungsional dari sistem. Hasil analisis terstruktur direpresentasikan dengan menggunakan data flow diagram (DFDs). Representasi DFD sulit diimplementasikan menggunakan bahasa pemrograman tradisional. Representasi DFD dapat ditransformasikan secara sistematis menjadi representasi struktur chart. Representasi bagan struktur dapat dengan mudah diimplementasikan menggunakan bahasa pemrograman konvensional.
- Selama desain terstruktur, representasi DFD yang diperoleh selama analisis terstruktur diubah menjadi representasi diagram struktur.
- Beberapa alat CASE tersedia untuk mendukung proses desain perangkat lunak yang dilakukan dengan menggunakan metodologi desain berorientasi fungsi yang penting. Selain meletakkan DFD, bagan struktur, memelihara kamus data, dan membantu dalam analisis ketertelusuran, alat CASE ini juga dapat melakukan beberapa pemeriksaan konsistensi dasar, misalnya, alat ini biasanya dapat memeriksa apakah DFD seimbang atau tidak.

6.8 LATIHAN

1. Pilih opsi yang benar:

a. Diagram aliran data menunjukkan:
    i. Kondisi yang menjadi dasar pemrosesan data
    ii. Urutan di mana kegiatan yang berbeda dilakukan
    iii. Transformasi data melalui stasiun pemrosesan
    iv. Urutan di mana berbagai fungsi suatu program dipanggil
b. DFD menggambarkan yang mana dari berikut ini?
    i. Aliran data
    ii. Aliran kontrol
    iii. Alur pernyataan
    iv. Tidak satu pun di atas
c. Manakah dari pernyataan berikut yang tidak benar tentang diagram aliran data (DFD)?
    i. Diagram hierarki.
    ii. Mewakili struktur kode
    iii. Tidak mewakili keputusan dan aliran kontrol.
    iv. Mewakili dekomposisi fungsional.
d. Dalam pendekatan desain prosedural, selama tahap desain rinci, manakah dari berikut ini yang dilakukan?
    i. Struktur modul dirancang
    ii. Representasi aliran data dikembangkan
    iii. Struktur data dan algoritma untuk masing-masing modul dikembangkan
    iv. Bagan struktur dikembangkan

2. Apa yang Anda pahami dengan istilah "dekomposisi top-down" dalam konteks desain berorientasi fungsi?
3. Bedakan antara diagram aliran data (DFD) dan diagram alir.
4. Bedakan antara analisis terstruktur dan desain terstruktur dalam konteks desain berorientasi fungsi.
5. Tunjukkan perbedaan penting antara bagan struktur dan bagan alir sebagai teknik representasi desain.
6. Apa yang dimaksud dengan kamus data istilah dalam konteks analisis terstruktur? Bagaimana kamus data berguna selama pengembangan dan pemeliharaan perangkat lunak?
7. Buatlah representasi DFD untuk program berikut:
8. Jelaskan bagaimana model perangkat lunak DFD dapat dibuat dari kode sumbernya.
9. Apa yang Anda pahami dengan istilah "analisis terstruktur" dan "desain terstruktur"? Apa tujuan utama dari "analisis terstruktur" dan "desain terstruktur"?
10. Jelaskan bagaimana model DFD dapat membantu seseorang memahami cara kerja sistem perangkat lunak.
11. Nyatakan apakah pernyataan berikut ini BENAR atau SALAH. “Inti dari prinsip desain berorientasi fungsi yang baik adalah memetakan fungsi serupa ke dalam modul.” Berikan alasan di balik jawaban Anda.
12. Identifikasi pernyataan yang benar. Berikan alasan di balik pilihan Anda.
    a. Model DFD pada dasarnya mewakili hubungan data dan kontrol di antara elemen-elemen program.
    b. Model DFD dari suatu sistem biasanya terdiri dari banyak DFD.
    c. Model DFD adalah model desain dari suatu sistem.
    d. Model DFD tidak dapat mewakili penyimpanan data file sistem.

13. Apakah yang Anda maksud: penyeimbangan DFD Ilustrasikan jawaban Anda dengan contoh yang sesuai.
14. Apa kekurangan utama dari diagram aliran data (DFD) sebagai alat untuk melakukan analisis terstruktur?
15. Mengapa review desain penting? Misalkan Anda diharuskan untuk mereview dokumen SA/SD, buatlah daftar item yang dapat digunakan sebagai checklist untuk melakukan review.
16. Apa yang Anda pahami dari tinjauan desain? Jenis kesalahan apa yang biasanya ditunjukkan oleh pengulas?
17. Gambarlah DFD berlabel untuk perangkat lunak manajemen waktu berikut. Tunjukkan dengan jelas diagram konteks dan dekomposisi hierarkisnya hingga level 2. (Catatan: Diagram konteks adalah DFD Level 0). Sebuah perusahaan perlu mengembangkan sistem manajemen waktu untuk para eksekutifnya. Perangkat lunak harus membiarkan para eksekutif mendaftarkan jadwal janji harian mereka. Informasi yang akan disimpan termasuk orang yang mengatur pertemuan, tempat, waktu dan durasi pertemuan, dan tujuannya (misalnya, untuk pekerjaan proyek tertentu). Ketika pertemuan yang melibatkan banyak eksekutif perlu diatur, sistem harus secara otomatis menemukan slot umum dalam buku harian eksekutif terkait, dan mengatur pertemuan (yaitu, membuat entri yang relevan dalam buku harian semua eksekutif terkait) pada saat itu. Itu juga harus memberi tahu eksekutif terkait tentang pertemuan yang dijadwalkan melalui email. Jika tidak ada slot umum yang tersedia, TMS harus membantu sekretaris untuk mengatur ulang penunjukan eksekutif dengan berkonsultasi dengan eksekutif terkait untuk memberikan ruang bagi slot bersama. Untuk membantu para eksekutif memeriksa jadwal mereka untuk hari tertentu, sistem harus memiliki antarmuka grafis yang sangat mudah digunakan. Karena para eksekutif dan sekretaris memiliki komputer desktop sendiri, perangkat lunak manajemen waktu harus dapat melayani beberapa permintaan jarak jauh secara bersamaan. Banyak eksekutif relatif pemula dalam penggunaan komputer. Setiap pagi perangkat lunak manajemen waktu harus mengirim email kepada setiap eksekutif tentang janjinya untuk hari itu. Selain mendaftarkan janji dan rapat mereka, para eksekutif mungkin menandai periode cuti yang mereka rencanakan. Selain itu, para eksekutif mungkin merencanakan pekerjaan penting yang perlu mereka lakukan pada hari apa pun pada jam yang berbeda dan mempostingnya dalam daftar pertunangan harian mereka. Fitur lain yang harus didukung oleh TMS adalah sebagai berikut—TMS harus dapat menyediakan beberapa jenis statistik seperti eksekutif mana yang menghabiskan berapa banyak waktu untuk rapat. Untuk proyek mana, berapa banyak pertemuan yang diselenggarakan untuk durasi berapa dan berapa banyak jam kerja yang dicurahkan untuk itu. Juga, itu harus dapat menampilkan untuk periode waktu tertentu sebagian kecil dari waktu rata-rata yang dihabiskan setiap eksekutif untuk rapat.
18. Sebuah hotel memiliki sejumlah kamar. Setiap kamar dapat berupa tipe single bed atau double bed dan mungkin tipe AC atau non-AC. Kamar memiliki harga yang berbeda-beda tergantung tipe kamar single atau double, AC atau Non-AC. Namun tarif kamar dapat bervariasi selama bagian tahun yang berbeda tergantung pada tingkat hunian. Untuk ini, komputer harus dapat menampilkan tingkat hunian rata-rata untuk bulan tertentu, sehingga pengelola dapat merevisi tarif kamar untuk bulan berikutnya baik ke atas atau ke bawah dengan persentase tertentu. Lakukan analisis terstruktur dan desain terstruktur untuk Perangkat Lunak Otomasi Hotel ini— perangkat lunak yang

akan mengotomatiskan aktivitas pembukuan hotel bintang 5. Para tamu dapat memesan kamar di muka atau dapat memesan kamar di tempat tergantung pada ketersediaan kamar. Resepsionis akan memasukkan data yang berkaitan dengan tamu seperti waktu kedatangan mereka, pembayaran di muka, perkiraan lama menginap, dan jenis kamar yang dibutuhkan. Tergantung pada data ini dan tergantung pada ketersediaan kamar yang sesuai, komputer akan memberikan nomor kamar kepada tamu dan memberikan nomor token unik untuk setiap tamu. Jika tamu tidak dapat diakomodasi, komputer menghasilkan pesan permintaan maaf. Manajer layanan katering hotel akan memasukkan jumlah dan jenis makanan saat dikonsumsi oleh tamu, nomor token tamu, dan tanggal dan waktu yang sesuai. Ketika pelanggan bersiap untuk check-out, perangkat lunak otomatisasi hotel harus menghasilkan seluruh tagihan untuk pelanggan dan juga mencetak jumlah saldo yang harus dibayar olehnya. Selama check-out, para tamu dapat memilih untuk mendaftarkan diri mereka sendiri untuk program tamu yang sering. Tamu yang sering datang harus diberi nomor identitas yang membantu mereka mendapatkan diskon khusus untuk tagihan mereka.

19. Lakukan analisis terstruktur dan desain terstruktur (SA/SD) untuk perangkat lunak yang akan dikembangkan untuk mengotomatisasi berbagai aktivitas pembukuan toko buku kecil. Dari diskusi dengan pemilik toko buku, persyaratan pengguna berikut untuk Software Otomasi Toko Buku (BAS) ini—telah sampai di: BAS akan membantu pelanggan menanyakan apakah buku ada dalam stok. Pengguna dapat menanyakan ketersediaan buku baik dengan menggunakan judul buku atau dengan menggunakan nama penulis. Jika buku saat ini tidak sedang dijual oleh toko buku, maka pelanggan diminta untuk memasukkan detail buku secara lengkap untuk pengadaan buku di kemudian hari. Pelanggan juga dapat memberikan alamat emailnya, sehingga ia dapat dihubungi secara otomatis oleh perangkat lunak saat dan ketika salinan buku diterima. Jika buku tersedia, jumlah eksemplar yang tersedia dan nomor rak tempat buku tersebut berada harus ditampilkan. Jika buku tidak tersedia, kueri untuk buku tersebut digunakan untuk menambah bidang permintaan untuk buku tersebut. Manajer secara berkala dapat melihat bidang permintaan buku untuk sampai pada perkiraan kasar mengenai permintaan saat ini untuk buku yang berbeda. BAS harus menjaga harga berbagai buku. Segera setelah pelanggan memilih bukunya untuk dibeli, petugas penjualan akan memasukkan nomor ISBN buku tersebut. BAS harus memperbarui stok, dan menghasilkan tanda terima penjualan untuk buku tersebut. BAS harus mengizinkan karyawan untuk memperbarui inventaris setiap kali pasokan baru tiba. Juga atas permintaan pemilik toko buku, BAS harus membuat statistik penjualan (yaitu, nama buku, penerbit, nomor ISBN, jumlah eksemplar yang terjual, dan pendapatan penjualan) untuk setiap periode. Statistik penjualan akan membantu pemilik untuk mengetahui bisnis yang tepat dilakukan selama periode waktu tertentu dan juga untuk menentukan tingkat persediaan yang diperlukan untuk berbagai buku. Tingkat persediaan yang diperlukan untuk sebuah buku sama dengan jumlah eksemplar buku yang terjual selama periode satu minggu dikalikan dengan rata-rata jumlah minggu yang dibutuhkan untuk mendapatkan buku dari penerbitnya. Setiap hari pemilik toko buku akan memberikan perintah kepada BAS untuk mencetak buku-buku yang berada di bawah ambang batas dan jumlah eksemplar yang akan dibeli beserta alamat lengkap penerbitnya.
20. Lakukan analisis terstruktur dan desain terstruktur untuk Perangkat Lunak Otomasi Perusahaan Kota (CCAS) berikut yang akan dikembangkan untuk mengotomatisasi berbagai kegiatan pembukuan yang terkait dengan berbagai tanggung jawab

Perusahaan Kota di kota besar. Sebuah perusahaan kota ingin mengembangkan situs web yang dengannya penduduk dapat memperoleh informasi tentang berbagai fasilitas yang disediakan oleh perusahaan kepada warga. Karena populasi kota melebihi 5 lakh, jumlah maksimum klik bersamaan dapat mencapai 10 klik per detik. Korporasi juga berencana menggunakan situs web yang sama untuk kegiatan pemeliharaan jalan. Sebuah perusahaan kota memiliki kantor cabang di pinggiran kota yang berbeda. Warga akan mengajukan permintaan perbaikan untuk berbagai jalan kota secara online. Supervisor di setiap kantor cabang harus dapat melihat semua permintaan perbaikan baru yang berkaitan dengan wilayahnya. Segera setelah permintaan perbaikan diajukan, seorang supervisor mengunjungi jalan tersebut dan mempelajari tingkat keparahan kondisi jalan. Tergantung pada tingkat keparahan kondisi jalan dan jenis lokalitas (misalnya, area komersial, area sibuk, area yang relatif sepi, dll.), ia menentukan prioritas untuk melakukan pekerjaan perbaikan ini. Supervisor juga memperkirakan kebutuhan bahan baku untuk jumlah dan jenis personel yang dibutuhkan. Supervisor memasukkan informasi ini melalui login khusus di situs web. Berdasarkan data ini, sistem harus menjadwalkan perbaikan jalan tergantung pada prioritas pekerjaan perbaikan dan tergantung pada ketersediaan bahan baku, mesin, dan personel. Laporan jadwal ini digunakan oleh supervisor untuk mengarahkan pekerjaan perbaikan yang berbeda. Tenaga kerja dan mesin yang tersedia dimasukkan oleh administrator perusahaan kota. Dia dapat mengubah data ini kapan saja. Tentu saja, setiap perubahan pada tenaga kerja dan mesin yang tersedia akan memerlukan penjadwalan ulang proyek. Kemajuan pekerjaan dimasukkan secara berkala oleh pengawas yang dapat dilihat oleh warga di situs web. Walikota kota dapat meminta berbagai statistik perbaikan jalan seperti jumlah dan jenis perbaikan yang dilakukan selama periode waktu tertentu dan pekerjaan perbaikan yang belum selesai setiap saat dan statistik pemanfaatan tenaga kerja dan mesin perbaikan selama periode waktu tertentu. periode waktu.

21. Lakukan analisis terstruktur dan desain terstruktur untuk mengembangkan Sistem Otomasi Restoran berikut menggunakan teknik SA/SD. Seorang pemilik restoran ingin mengkomputerisasikan pemrosesan pesanan, penagihan, dan aktivitas akuntansinya. Dia juga mengharapkan komputer untuk menghasilkan laporan statistik tentang penjualan item yang berbeda. Tujuan utama dari komputerisasi ini adalah untuk membuat pemesanan persediaan lebih akurat sehingga masalah kelebihan persediaan dihindari serta masalah tidak tersedianya bahan yang dibutuhkan untuk memenuhi pesanan untuk beberapa item populer diminimalkan. Komputer harus menjaga harga semua item dan juga mendukung perubahan harga oleh manajer. Setiap kali ada barang yang dijual, petugas penjualan akan memasukkan kode barang dan jumlah yang terjual. Komputer harus menghasilkan tagihan setiap kali makanan dijual. Setiap kali bahan dikeluarkan untuk persiapan makanan, data harus dimasukkan ke dalam komputer. Pesanan pembelian dibuat setiap hari, setiap kali stok untuk bahan apa pun turun di bawah nilai ambang batas. Komputer harus menghitung nilai ambang batas untuk setiap item berdasarkan konsumsi rata-rata bahan ini selama tiga hari terakhir dan dengan asumsi bahwa stok minimal dua hari harus dipertahankan untuk semua bahan. Setiap kali bahan yang dipesan tiba, data faktur mengenai jumlah dan harga dimasukkan. Jika saldo kas cukup tersedia, komputer harus segera mencetak cek terhadap faktur. Data penerimaan dan pengeluaran penjualan bulanan harus dibuat setiap kali manajer meminta untuk melihatnya.

22. Lakukan analisis dan desain terstruktur untuk perangkat lunak Sistem Informasi Peradilan (JIS) berikut. Kejaksaan Agung telah meminta kita untuk mengembangkan Sistem Informasi Peradilan (JIS), untuk membantu menangani kasus-kasus pengadilan dan juga untuk membuat kasus-kasus pengadilan yang lalu mudah diakses oleh para pengacara dan hakim. Untuk setiap kasus pengadilan, nama terdakwa, alamat terdakwa, jenis kejahatan (misalnya, pencurian, pembakaran, dll), ketika dilakukan (tanggal), di mana dilakukan (lokasi), nama petugas yang menangkap, dan tanggal penangkapan dicatat oleh panitera pengadilan. Setiap kasus pengadilan diidentifikasi dengan nomor identifikasi kasus unik (CIN) yang dihasilkan oleh komputer. Panitera menetapkan tanggal sidang untuk setiap kasus. Untuk ini registrar mengharapkan komputer untuk menampilkan slot kosong pada setiap hari kerja selama kasus dapat dijadwalkan. Setiap kali kasus ditunda, alasan penundaan dimasukkan oleh panitera dan dia menetapkan tanggal sidang baru. Jika sidang berlangsung pada hari apa pun untuk suatu kasus, panitera memasukkan ringkasan proses pengadilan dan menetapkan tanggal sidang yang baru. Juga, pada penyelesaian kasus pengadilan, ringkasan putusan dicatat dan kasus ditutup tetapi rincian kasus dipertahankan untuk referensi di masa mendatang. Data lain yang dipelihara tentang suatu kasus termasuk nama hakim ketua, jaksa penuntut umum, tanggal mulai, dan tanggal penyelesaian yang diharapkan dari suatu persidangan. Para hakim harus dapat menelusuri kasus-kasus lama untuk mendapatkan pedoman dalam penilaian mereka. Pengacara juga harus diizinkan untuk menelusuri kasus lama, tetapi harus dikenakan biaya untuk setiap kasus lama yang mereka telusuri. Dengan menggunakan perangkat lunak JIS, Panitera pengadilan harus dapat menanyakan hal-hal berikut:
    a. Kasus-kasus pengadilan yang sedang menunggu keputusan. Menanggapi permintaan ini, komputer harus mencetak kasus tertunda yang diurutkan berdasarkan CIN. Untuk setiap kasus yang tertunda, data berikut harus dicantumkan—tanggal dimulainya kasus, nama terdakwa, alamat, rincian kejahatan, nama pengacara, nama jaksa penuntut umum, dan nama hakim yang hadir.
    b. Kasus-kasus yang telah diselesaikan selama periode tertentu. Keluaran dalam kasus ini harus secara kronologis mencantumkan tanggal mulai kasus, CIN, tanggal putusan dijatuhkan, nama hakim yang hadir, dan ringkasan putusan.
    c. Kasus-kasus yang akan disidangkan pada tanggal tertentu.
    d. Status kasus tertentu (kasus diidentifikasi oleh CIN)
23. Berbagai kegiatan perpustakaan lembaga yang berkaitan dengan penerbitan dan pengembalian buku oleh anggota perpustakaan dan berbagai pertanyaan mengenai buku seperti yang tercantum di bawah ini akan diotomatisasi. Lakukan analisis terstruktur dan desain terstruktur untuk perangkat lunak Sistem Informasi Perpustakaan (LIS) ini:
    a. Perpustakaan memiliki 10.000 buku. Setiap buku diberi nomor identifikasi unik (disebut nomor ISBN). Petugas Perpustakaan harus dapat memasukkan detail buku ke dalam LIS melalui antarmuka yang sesuai.
    b. Ada empat kategori anggota perpustakaan—mahasiswa sarjana, mahasiswa pascasarjana, sarjana riset, dan anggota fakultas.
    c. Setiap anggota perpustakaan diberi nomor kode keanggotaan perpustakaan yang unik. Setiap mahasiswa S1 dapat menerbitkan hingga 2 buku untuk durasi 1 bulan.

d. Setiap mahasiswa pascasarjana dapat menerbitkan hingga 4 buku untuk durasi 1 bulan.
e. Setiap peneliti dapat menerbitkan hingga 6 buku dengan durasi 3 bulan.
f. Setiap anggota fakultas dapat menerbitkan hingga 10 buku selama enam bulan.
g. LIS harus menjawab pertanyaan pengguna mengenai apakah buku tertentu tersedia. Jika buku tersedia, LIS harus menampilkan nomor rak tempat buku tersedia dan jumlah eksemplar yang tersedia.
h. LIS mendaftarkan setiap buku yang diterbitkan untuk seorang anggota. Saat anggota mengembalikan buku, LIS menghapus buku dari rekening anggota dan membuat buku tersedia untuk terbitan mendatang.
i. Anggota harus diperbolehkan untuk memesan buku yang telah diterbitkan. Ketika buku yang dipesan dikembalikan, LIS harus mencetak slip untuk anggota yang bersangkutan untuk mendapatkan buku yang diterbitkan dan harus melarang penerbitan buku kepada anggota lain untuk jangka waktu tujuh hari atau sampai anggota yang telah memesan buku mendapatkannya.
j. Ketika seorang anggota mengembalikan sebuah buku, LIS mencetak tagihan untuk biaya penalti untuk buku-buku yang lewat jatuh tempo. LIS menghitung biaya penalti dengan mengalikan jumlah hari keterlambatan buku dengan tingkat penalti.
k. LIS mencetak pesan pengingat untuk anggota yang bukunya sudah lewat jatuh tempo, atas permintaan Pustakawan.
l. LIS harus memungkinkan Pustakawan untuk membuat dan menghapus catatan anggota. Setiap anggota harus diberikan nomor identifikasi keanggotaan unik yang dapat digunakan anggota untuk menerbitkan, mengembalikan, dan memesan buku.

24. Lakukan SA/SD untuk perangkat lunak pengolah kata berikut.
    a. Perangkat lunak pengolah kata harus dapat membaca teks dari file ASCII atau file HTML dan menyimpan teks yang diformat sebagai file HTML dalam disk.
    b. Perangkat lunak pengolah kata harus menanyakan kepada pengguna tentang jumlah karakter dalam baris output dari teks yang diformat.
    c. Pengguna harus diizinkan untuk memilih nomor antara 1 dan 132.
    d. Perangkat lunak pengolah kata harus memproses teks input dengan cara berikut.
        i. Setiap baris keluaran harus berisi persis jumlah karakter yang ditentukan oleh pengguna (termasuk yang kosong).
        ii. Perangkat lunak pengolah kata adalah untuk membenarkan teks ke kiri dan kanan sehingga tidak ada yang kosong di ujung kiri dan kanan baris kecuali baris pertama dan mungkin baris terakhir paragraf. Perangkat lunak pengolah kata harus melakukan ini dengan menyisipkan tambahan kosong di antara kata-kata.
        iii. Teks input dari file ASCII harus terdiri dari kata-kata yang dipisahkan oleh satu atau lebih kosong dan karakter khusus PP, yang menunjukkan akhir paragraf dan awal paragraf lainnya.
        iv. Baris pertama setiap paragraf harus diberi indentasi lima spasi dan rata kanan.
        v. Baris terakhir dari setiap paragraf harus dibiarkan rata.

e. Pengguna harus dapat menelusuri dokumen dan menambah, mengubah, atau menghapus kata. Dia juga harus dapat menandai kata apa pun sebagai huruf tebal, miring, superskrip, atau subskrip.
f. Pengguna dapat meminta untuk melihat jumlah karakter, kata, baris, dan paragraf yang digunakan dalam dokumen.
g. Pengguna harus dapat menyimpan dokumennya dengan nama yang ditentukan olehnya.

25. Diperlukan untuk mengembangkan paket perangkat lunak editor grafis yang dapat digunakan untuk membuat/memodifikasi beberapa jenis entitas grafis. Singkatnya, editor grafis harus mendukung fitur-fitur berikut: (Mereka yang tidak terbiasa dengan editor grafis, silakan lihat fitur Gambar Grafis yang tersedia baik dalam perangkat lunak MS-Word atau PowerPoint. Anda juga dapat memeriksa paket Gambar Grafis lainnya yang dapat diakses untuk Anda. Pemahaman tentang fitur standar Editor Grafis akan membantu Anda memahami berbagai fitur yang diperlukan.)
    a. Editor grafis harus mendukung pembuatan beberapa jenis objek geometris seperti lingkaran, elips, persegi panjang, garis, teks, dan poligon.
    b. Setiap objek yang dibuat dapat dipilih dengan mengklik tombol mouse pada objek tersebut. Objek yang dipilih harus ditampilkan dalam warna yang disorot.
    c. Objek yang dipilih dapat diedit, yaitu, karakteristik terkaitnya seperti bentuk geometris, lokasi, warna, gaya isian, lebar garis, gaya garis, dll. dapat diubah. Untuk teks, isi teks dapat diubah.
    d. Objek yang dipilih dapat disalin, dipindahkan, atau dihapus.
    e. Editor grafis harus mengizinkan pengguna untuk menyimpan gambar yang dibuatnya pada disk dengan nama yang akan dia tentukan. Editor grafis juga harus mendukung pemuatan gambar yang dibuat sebelumnya dari disk.
    f. Pengguna harus dapat menentukan area persegi apa pun di layar yang akan diperbesar untuk memenuhi seluruh layar.
    g. Fungsi layar pas membuat seluruh gambar pas dengan layar dengan secara otomatis\ menyesuaikan nilai zoom dan geser.
    h. Fungsi pan harus memungkinkan gambar yang ditampilkan digeser ke segala arah dengan jumlah tertentu.
    i. Editor grafis harus mendukung pengelompokan. Grup hanyalah sekumpulan objek gambar termasuk grup lain yang ketika dikelompokkan berperilaku sebagai satu kesatuan. Fitur ini sangat berguna ketika Anda ingin memanipulasi beberapa entitas dengan cara yang sama. Objek gambar dapat menjadi anggota langsung paling banyak satu grup. Seharusnya dimungkinkan untuk melakukan beberapa operasi pengeditan pada grup seperti memindahkan, menghapus, dan menyalin.
    j. Satu set 10 papan klip harus disediakan yang dapat digunakan untuk menyalin berbagai jenis entitas yang dipilih (termasuk grup) untuk digunakan di masa mendatang dalam menempelkannya di tempat yang berbeda bila diperlukan.

26. Lakukan SA/SD untuk perangkat lunak katalog komponen Perangkat Lunak berikut. Perangkat lunak katalog komponen perangkat lunak: Perangkat lunak katalog komponen perangkat lunak terdiri dari katalog komponen perangkat lunak dan berbagai fungsi yang didefinisikan pada katalog komponen ini. Katalog komponen perangkat lunak harus menyimpan rincian komponen yang berpotensi dapat digunakan kembali. Komponen yang dapat digunakan kembali dapat berupa desain atau kode. Desain mungkin telah dibangun menggunakan notasi desain yang berbeda

seperti UML, ERD, desain terstruktur, dll. Demikian pula, kode mungkin telah ditulis menggunakan bahasa pemrograman yang berbeda. Seorang pembuat katalog dapat memasukkan komponen ke dalam katalog, dapat menghapus komponen dari katalog, dan dapat mengaitkan informasi penggunaan kembali dengan komponen katalog dalam bentuk kumpulan kata kunci. Pengguna katalog dapat menanyakan tentang ketersediaan komponen menggunakan kata kunci tertentu untuk mendeskripsikan komponen tersebut. Untuk membantu mengelola katalog komponen (yaitu, secara berkala membersihkan komponen yang tidak digunakan) perangkat lunak katalog harus memelihara informasi seperti berapa kali komponen telah digunakan, dan berapa kali komponen muncul dalam kueri tetapi tidak digunakan. Karena jumlah komponen biasanya cenderung sangat tinggi, diinginkan untuk dapat mengklasifikasikan berbagai jenis komponen secara hierarkis. Seorang pengguna harus dapat menelusuri komponen di setiap kategori.

27. Manajer supermarket ingin mengembangkan perangkat lunak otomatisasi. Supermarket menyediakan satu set barang. Pelanggan mengambil barang yang mereka inginkan dari konter yang berbeda dalam jumlah yang dibutuhkan. Pelanggan menyajikan barang-barang ini kepada petugas penjualan. Petugas penjualan memasukkan nomor kode barang-barang tersebut beserta jumlah/unitnya masing-masing. Lakukan analisis terstruktur dan desain terstruktur untuk mengembangkan Supermarket Automation Software (SAS) ini.
    a. SAS harus pada akhir transaksi penjualan mencetak tagihan yang berisi nomor seri transaksi penjualan, nama barang, nomor kode, jumlah, harga satuan, dan harga barang. Tagihan harus menunjukkan jumlah total yang harus dibayar.
    b. SAS harus menjaga inventaris berbagai item supermarket. Manajer atas permintaan harus dapat melihat detail inventaris. Untuk mendukung manajemen persediaan, persediaan suatu barang harus dikurangi setiap kali suatu barang terjual. SAS juga harus mendukung opsi di mana karyawan dapat memperbarui inventaris saat pasokan baru tiba.
    c. SAS harus mendukung pencetakan statistik penjualan untuk setiap barang yang dijual supermarket untuk hari tertentu atau periode tertentu. Statistik penjualan harus menunjukkan jumlah barang yang dijual, harga yang direalisasikan, dan keuntungan.
    d. Manajer supermarket harus dapat mengubah harga barang yang dijual karena harga barang yang berbeda bervariasi setiap hari.
28. Sebuah perusahaan transportasi ingin mengkomputerisasi berbagai aktivitas penyimpanan buku yang terkait dengan operasinya. Lakukan analisis terstruktur dan desain terstruktur untuk mengembangkan perangkat lunak Transport Company Computerization (TCC):
    a. Sebuah perusahaan transportasi memiliki sejumlah truk.
    b. Perusahaan transportasi ini memiliki kantor pusat yang berlokasi di ibu kota dan memiliki kantor cabang di beberapa kota lain.
    c. Perusahaan transportasi menerima kiriman berbagai ukuran di (diukur dalam meter kubik) kantor yang berbeda untuk diteruskan ke kantor cabang yang berbeda di seluruh negeri.
    d. Setelah kiriman tiba di kantor perusahaan transportasi, rincian volume, alamat tujuan, alamat pengirim, dll., dimasukkan ke dalam komputer. Komputer akan menghitung biaya transportasi tergantung pada volume kiriman dan tujuannya dan akan mengeluarkan tagihan untuk kiriman tersebut.

e. Setelah volume tujuan tertentu menjadi 500 meter kubik, sistem Komputerisasi akan secara otomatis membagikan truk yang tersedia berikutnya.
f. Sebuah truk tetap berada di kantor cabang sampai kantor cabang memiliki cukup kargo untuk memuat truk secara penuh.
g. Manajer harus dapat melihat status truk yang berbeda setiap saat.
h. Manajer harus dapat melihat penggunaan truk selama periode waktu tertentu.
i. Ketika truk tersedia dan kiriman yang diperlukan tersedia untuk pengiriman, sistem komputer harus mencetak rincian nomor kiriman, volume, nama dan alamat pengirim, serta nama dan alamat penerima untuk diteruskan bersama truk.
j. Manajer perusahaan transportasi dapat menanyakan status kiriman tertentu dan rincian volume kiriman yang ditangani ke tujuan tertentu dan pendapatan terkait yang dihasilkan.
k. Manajer juga harus dapat melihat rata-rata masa tunggu untuk kiriman yang berbeda. Statistik ini penting baginya karena dia biasanya memesan truk baru ketika rata-rata masa tunggu kiriman menjadi tinggi karena tidak tersedianya truk. Juga, manajer ingin melihat waktu idle rata-rata truk di cabang selama periode tertentu untuk perencanaan masa depan.

29. Gambarlah diagram alir data level 0 (level konteks) dan level 1 untuk perangkat lunak manajemen arsip akademik mahasiswa berikut ini.
    a. Satu set kursus dibuat. Setiap mata kuliah terdiri dari nomor mata kuliah yang unik, jumlah SKS, dan silabus.
    b. Siswa diterima di kursus. Rincian setiap mahasiswa termasuk nomor roll, alamat, nomor semester dan mata kuliah yang terdaftar untuk semester tersebut.
    c. Nilai mahasiswa untuk berbagai unit yang dia beri kredit dimasukkan.
    d. Setelah nilai dimasukkan, rata-rata tertimbang semester (SWA) dihitung.
    e. Nilai mahasiswa baru-baru ini ditambahkan ke nilai sebelumnya dan rata-rata tertimbang berdasarkan poin kredit untuk berbagai unit dihitung.
    f. Nilai untuk semester saat ini diformat dan dicetak.
    g. SWA muncul di laporan.
    h. Pemeriksaan harus dilakukan untuk menentukan apakah seorang mahasiswa harus ditempatkan dalam daftar Wakil Rektor. Ini ditentukan berdasarkan apakah mahasiswa mendapat nilai SWA 85 atau lebih tinggi.
    i. Jika SWA lebih rendah dari 50, mahasiswa ditempatkan pada posisi bersyarat.
30. Lakukan analisis terstruktur dan desain terstruktur (SA/SD) untuk alat CASE berikut untuk Perangkat Lunak Analisis Terstruktur yang akan dikembangkan untuk mengotomatisasi berbagai aktivitas yang terkait dengan pengembangan alat CASE untuk analisis perangkat lunak terstruktur.
    a. Alat kasus harus mendukung antarmuka grafis dan fitur berikut.
    b. Pengguna harus dapat menggambar gelembung, penyimpanan data, dan entitas dan menghubungkannya menggunakan panah aliran data. Panah aliran data dianotasi dengan nama data yang sesuai.
    c. Harus mendukung pengeditan diagram aliran data.
    d. Harus dapat membuat diagram secara hierarkis.
    e. Pengguna harus dapat menentukan kesalahan penyeimbangan kapan pun diperlukan.
    f. Perangkat lunak harus dapat membuat kamus data secara otomatis.

g. Harus mendukung pencetakan diagram pada berbagai printer.
h. Harus mendukung kueri item data dan nama fungsi.
i. Diagram yang cocok dengan kueri harus ditampilkan.

31. Lakukan analisis terstruktur dan desain terstruktur (SA/SD) untuk perangkat lunak yang akan dikembangkan untuk mengotomatisasi berbagai aktivitas yang terkait dengan pengembangan alat CASE untuk desain perangkat lunak terstruktur. Ringkasan persyaratan alat CASE untuk Desain Terstruktur ini adalah sebagai berikut:
    a. Alat kasus harus mendukung antarmuka grafis dan fitur berikut.
    b. Seharusnya dimungkinkan untuk mengimpor model DFD yang dikembangkan oleh program lain. Pengguna harus dapat menerapkan analisis transformasi dan transaksi ke model DFD yang diimpor.
    c. Pengguna harus dapat menggambar modul, panah kontrol, dan panah aliran data. Juga simbol untuk modul perpustakaan harus disediakan. Panah aliran data dianotasi dengan nama data yang sesuai.
    d. Modul harus diatur dalam tingkat hierarkis.
    e. Pengguna harus dapat memodifikasi desainnya. Harap dicatat bahwa ketika dia menghapus panah aliran data, nama data yang dianotasi secara otomatis akan dihapus.
    f. Untuk perangkat lunak besar, modul dapat diatur secara hierarkis dan mengklik modul harus dapat menunjukkan organisasi internalnya.
    g. Pengguna harus dapat menyimpan desainnya dan juga dapat memuat desain yang telah dibuat sebelumnya.
32. Agen pengiriman surat kabar dan majalah lokal telah meminta kita untuk mengembangkan perangkat lunak baginya untuk mengotomatisasi berbagai kegiatan administrasi yang terkait dengan bisnisnya. Lakukan analisis dan desain terstruktur untuk Perangkat Lunak Otomasi Agen Surat Kabar ini.
    a. Perangkat lunak ini akan digunakan oleh manajer kantor berita dan orang-orang pengantarnya.
    b. Untuk setiap orang pengantar, sistem harus mencetak setiap hari publikasi yang akan dikirimkan ke setiap alamat.
    c. Pelanggan biasanya berlangganan satu atau lebih surat kabar dan majalah. Mereka diizinkan untuk mengubah pemberitahuan berlangganan mereka dengan memberikan pemberitahuan satu minggu sebelumnya. Pelanggan harus dapat memulai langganan baru dan menangguhkan langganan untuk item tertentu baik sementara atau permanen melalui browser web. Mempertimbangkan basis pelanggan yang besar, setidaknya 10 akses pelanggan secara bersamaan harus didukung.
    d. Untuk setiap orang pengantar, sistem harus mencetak setiap hari publikasi yang akan dikirimkan ke setiap alamat.
    e. Sistem juga harus mencetak untuk agen berita informasi mengenai siapa yang menerima apa dan ringkasan informasi bulan ini.
    f. Setiap awal bulan tagihan dicetak oleh sistem untuk dikirimkan ke pelanggan. Tagihan ini harus dihitung oleh sistem secara otomatis.
    g. Pelanggan dapat meminta untuk menghentikan pengiriman kepada mereka untuk jangka waktu tertentu ketika mereka keluar dari stasiun.
    h. Pelanggan dapat meminta untuk berlangganan surat kabar/majalah baru, mengubah daftar langganan mereka, atau menghentikan langganan mereka sama sekali.

- i. Pelanggan biasanya membayar iuran bulanan mereka baik dengan cek atau tunai. Setelah nomor cek atau uang tunai yang diterima dimasukkan ke dalam sistem, tanda terima untuk pelanggan harus dicetak.
- j. Jika ada pelanggan yang memiliki tunggakan terutang selama satu bulan, pesan pengingat sopan dicetak untuknya dan langganannya dihentikan jika iurannya tetap terutang selama lebih dari dua bulan.
- k. Perangkat lunak harus menghitung dan mencetak jumlah yang harus dibayarkan kepada setiap petugas pengiriman. Setiap pengantar barang mendapat 2,5 persen dari nilai publikasi yang dibawakannya.

33. Lakukan SA/SD untuk Sistem Informasi Jurusan Universitas berikut. Perangkat lunak ini menyangkut mengotomatisasi kegiatan kantor departemen universitas. Kantor departemen di universitas yang berbeda melakukan banyak kegiatan pembukuan perangkat lunak yang akan dikembangkan target untuk mengotomatisasi kegiatan tersebut.
    - a. Berbagai rincian mengenai setiap mahasiswa seperti nama, alamat, program studi yang terdaftar, dll dimasukkan pada saat ia mengambil penerimaan.
    - b. Setiap awal semester, mahasiswa melakukan registrasi mata kuliah. Sistem informasi harus memungkinkan sekretaris departemen untuk memasukkan data mengenai pendaftaran mata kuliah mahasiswa. Saat sekretaris memasukkan nomor urut setiap siswa, sistem komputer harus membuka formulir untuk mahasiswa yang bersangkutan dan harus melacak mata kuliah yang telah dia selesaikan dan mata kuliah yang dia miliki, dll.
    - c. Pada akhir semester, instruktur meninggalkan informasi penilaian mereka di kantor yang dimasukkan sekretaris di komputer. Sistem informasi harus dapat menghitung nilai rata-rata untuk setiap mahasiswa untuk semester dan rata-rata nilai kumulatif (CGPA) dan mencetak lembar nilai untuk setiap siswa.
    - d. Sistem informasi juga melacak inventaris Departemen, seperti peralatan, lokasinya, furnitur, dll.
    - e. Departemen memiliki hibah tahunan dan Departemen membelanjakannya untuk membeli peralatan, buku, alat tulis, dll. Selain itu, selain hibah tahunan yang diperoleh Departemen dari Universitas, Departemen mendapatkan dana dari berbagai layanan konsultasi yang diberikannya kepada organisasi yang berbeda. Sistem informasi Departemen perlu melacak rekening Departemen.
    - f. Sistem informasi juga harus melacak proyek penelitian Departemen, publikasi oleh fakultas, dll. Informasi ini dikunci oleh sekretaris Departemen.
    - g. Sistem informasi harus mendukung permintaan informasi terkini tentang setiap mahasiswa dengan memasukkan nomor gulungannya. Ini juga harus mendukung permintaan perincian rekening buku kas. Output dari query ini harus mencakup pendapatan, pengeluaran, dan saldo.
34. Lakukan SA/SD untuk mengembangkan perangkat lunak untuk mengotomatisasi aktivitas toko suku cadang mobil kecil. Toko suku cadang mobil kecil menjual suku cadang untuk kendaraan dari beberapa merek dan model. Selain itu, setiap suku cadang biasanya diproduksi oleh beberapa industri kecil. Untuk memperlancar penjualan dan pemesanan persediaan, pemilik toko telah meminta kita untuk mengembangkan perangkat lunak toko suku cadang motor berikut. Lakukan SA/SD untuk Perangkat Lunak Toko Suku Cadang Motor (MPSS) ini. Toko suku cadang motor berurusan dengan sejumlah besar suku cadang motor dari berbagai produsen dan berbagai jenis kendaraan. Beberapa bagian motor sangat kecil dan beberapa

berukuran cukup besar. Pemilik toko memelihara berbagai bagian di rak yang dipasang di dinding dan bernomor. Pemilik toko mempertahankan persediaan sesedikit mungkin untuk setiap item sebagai wajar, untuk mengurangi overhead persediaan setelah terinspirasi oleh filosofi *just-in-time* (JIT). Dengan demikian, satu masalah penting yang dihadapi pemilik toko adalah dapat memesan barang segera setelah jumlah barang dalam persediaan berkurang di bawah nilai ambang batas. Pemilik toko ingin mempertahankan suku cadang untuk dapat mempertahankan penjualan selama sekitar satu minggu. Untuk menghitung nilai ambang batas untuk setiap item, perangkat lunak harus dapat menghitung jumlah rata-rata penjualan suku cadang selama satu minggu untuk setiap suku cadang. Di penghujung hari, pemilik toko akan meminta komputer untuk membuat item yang akan dipesan. Komputer harus mencetak nomor suku cadang, jumlah yang diperlukan dan alamat vendor yang memasok suku cadang tersebut. Komputer juga harus menghasilkan pendapatan untuk setiap hari dan pada akhir bulan, komputer harus membuat grafik yang menunjukkan penjualan untuk setiap hari dalam sebulan.

35. Lakukan analisis terstruktur dan desain terstruktur untuk Toko Obat berikut
    a. Perangkat lunak otomatisasi (MSA)
    b. Sebuah toko obat eceran berurusan dengan sejumlah besar obat-obatan yang dibeli dari berbagai produsen. Pemilik toko menyimpan obat-obatan yang berbeda di rak yang dipasang di dinding dan diberi nomor.
    c. Pemilik toko mempertahankan persediaan sesedikit mungkin untuk setiap item sebagai wajar, untuk mengurangi overhead persediaan setelah terinspirasi oleh filosofi just-in-time (JIT).
    d. Dengan demikian, satu masalah penting yang dihadapi pemilik toko adalah dapat memesan barang segera setelah jumlah barang dalam persediaan berkurang di bawah nilai ambang batas. Pemilik toko ingin mempertahankan obat-obatan untuk dapat mempertahankan penjualan selama sekitar satu minggu. Untuk menghitung nilai ambang batas untuk setiap item, perangkat lunak harus dapat menghitung jumlah rata-rata penjualan obat selama satu minggu untuk setiap bagian.
    e. Di penghujung hari, pemilik toko akan meminta komputer untuk membuat item yang akan dipesan. Komputer harus mencetak deskripsi obat, jumlah yang dibutuhkan, dan alamat vendor yang memasok obat. Pemilik toko harus dapat menyimpan nama, alamat, dan nomor kode obat yang ditangani masing-masing vendor.
    f. Setiap kali pasokan baru tiba, pemilik toko akan memasukkan nomor kode barang, jumlah, nomor batch, tanggal kedaluwarsa, dan nomor vendor. Perangkat lunak harus mencetak cek yang mendukung vendor untuk item yang disediakan.
    g. Ketika pemilik toko membeli obat baru yang belum ditangani sebelumnya, ia harus dapat memasukkan rincian obat seperti nama dagang obat, nama generik, vendor yang dapat memasok obat ini, harga jual satuan dan harga beli. Komputer harus membuat nomor kode untuk obat ini yang akan ditempelkan oleh pemilik toko di rak tempat obat ini disimpan. Pemilik toko harus dapat menanyakan tentang obat baik menggunakan nama generik atau nama dagang dan perangkat lunak harus menampilkan nomor kode dan jumlah yang ada.
    h. Setiap akhir hari pemilik toko akan memberikan perintah untuk membuat daftar obat-obatan yang telah kadaluwarsa. Itu juga harus menyiapkan daftar

barang kadaluarsa dari vendor sehingga pemilik toko dapat meminta vendor untuk mengganti barang-barang ini. Saat ini, aktivitas ini saja membutuhkan banyak tenaga dari pemilik toko dan merupakan motivasi utama untuk upaya otomatisasi.

i. Setiap kali ada penjualan, pemilik toko akan memasukkan nomor kode setiap obat dan jumlah yang sesuai yang dijual. MSA harus mencetak tanda terima kas.

j. Komputer juga harus menghasilkan pendapatan dan laba untuk periode tertentu. Itu juga harus menunjukkan pembayaran vendor-bijaksana untuk periode tersebut.

36. Hall Management Center (HMC) mahasiswa IIT telah meminta kita untuk mengembangkan perangkat lunak berikut untuk mengotomatisasi berbagai kegiatan pembukuan yang terkait dengan operasi sehari-harinya.

    a. Setelah mahasiswa masuk, dia menunjukkan catatan dari unit penerimaan, bersama dengan namanya, alamat tetap, nomor telepon kontak, dan foto. Dia kemudian diberikan aula, dan juga nomor kamar tertentu. Surat yang menunjukkan ruang yang dialokasikan ini dikeluarkan untuk mahasiswa yang bersangkutan.
    b. Siswa dikenakan biaya kekacauan setiap bulan. Manajer mess akan memasukkan ke perangkat lunak total biaya untuk setiap mahasiswa dalam sebulan pada rekening mess.
    c. Setiap kamar memiliki sewa kamar tetap. Aula yang baru dibangun memiliki harga sewa yang lebih tinggi dibandingkan dengan beberapa aula yang lebih tua.
    d. Kamar twin sharing memiliki harga sewa yang lebih rendah.
    e. Setiap aula menyediakan fasilitas tertentu untuk mahasiswa seperti ruang baca, ruang bermain, ruang TV, dll. Jumlah tetap dikenakan pada setiap mahasiswa dalam hitungan ini.
    f. Jumlah total yang dikumpulkan dari setiap mahasiswa aula untuk biaya mess diserahkan kepada manajer mess setiap bulan. Untuk ini, komputer perlu mencetak lembar dengan jumlah total karena setiap manajer kekacauan dicetak.
    g. Cek tercetak dikeluarkan untuk setiap manajer dan tanda tangan diperoleh dari mereka pada lembar.
    h. Setiap kali seorang mahasiswa datang untuk membayar iurannya, total iurannya dihitung sebagai jumlah biaya mess, biaya fasilitas, dan sewa kamar.
    i. Mahasiswa harus dapat menyampaikan berbagai jenis keluhan menggunakan web browser di kamar mereka atau di Lab.
    j. Pengaduan tersebut dapat berupa permintaan perbaikan seperti lampu sekering, keran air tidak berfungsi, filter air tidak berfungsi, perbaikan ruangan, dll. Juga dapat mengajukan pengaduan mengenai perilaku petugas, staf mess, dll. Untuk operasi 24 jam ini dari perangkat lunak diperlukan.
    k. HMC menerima hibah tahunan dari Institut untuk gaji staf dan pemeliharaan aula dan kebun. Ketua HMC harus diberikan dukungan untuk distribusi hibah di antara aula yang berbeda. Pengawas dari aula yang berbeda harus dapat memasukkan rincian pengeluaran mereka terhadap alokasi.
    l. Petugas pengendali harus dapat melihat keseluruhan hunian kamar.

m. Kepala setiap aula harus dapat mengetahui penghuni aulanya. Ia juga harus dapat melihat keluhan yang diajukan oleh mahasiswa dan memposting laporan tindakan yang diambil (ATR) untuk setiap keluhan.
n. Aula mempekerjakan pembantu dan tukang kebun. Karyawan sementara ini menerima gaji tetap setiap hari.
o. Petugas Aula memasuki setiap cuti yang diambil oleh petugas atau tukang kebun dari terminal yang terletak di kantor aula. Pada akhir setiap bulan, daftar gabungan gaji yang harus dibayarkan kepada setiap karyawan aula bersama dengan cek untuk setiap karyawan dicetak.
p. HMC mengeluarkan biaya kecil seperti pekerjaan perbaikan yang dilakukan, langganan surat kabar dan majalah, dll. Pengeluaran ini harus dimungkinkan.
q. Setiap kali seorang staf baru direkrut, rinciannya termasuk gaji hariannya dimasukkan. Setiap kali seorang staf pergi, catatannya harus dapat dihapus.
r. Petugas harus dapat melihat laporan rekening setiap saat. Kepala penjara akan mencetak laporan rekening konsolidasi tahunan, menandatangani dan menyerahkannya ke administrasi Institut untuk persetujuan dan verifikasi audit.

37. Perangkat lunak keamanan IIT: Kantor keamanan IIT membutuhkan perangkat lunak untuk mengontrol dan memantau lalu lintas kendaraan yang masuk dan keluar kampus. Fungsionalitas yang diperlukan dari perangkat lunak adalah sebagai berikut— Setiap kendaraan di kampus IIT akan didaftarkan ke sistem. Untuk itu, setiap dosen, staf, dan mahasiswa yang memiliki satu atau lebih kendaraan harus mengisi formulir di kantor keamanan yang merinci nomor STNK, model, dan detail terkait lainnya untuk kendaraan yang mereka miliki. Data ini akan dimasukkan ke komputer oleh staf keamanan setelah pemeriksaan uji tuntas. Setiap kali kendaraan masuk atau keluar kampus, kamera yang dipasang di dekat gerbang pemeriksaan akan menentukan nomor registrasi kendaraan yang masuk (atau keluar) dan model kendaraan dan dimasukkan ke dalam sistem. Jika kendaraan tersebut adalah kendaraan kampus, maka gerbang pemeriksaan harus terangkat secara otomatis untuk membiarkannya masuk atau keluar, tergantung kasusnya. Berbagai rincian mengenai keluar masuknya kendaraan kampus seperti nomor, pemilik, tanggal dan waktu masuk/keluar akan disimpan dalam database untuk keperluan statistik. Untuk setiap kendaraan luar yang memasuki kampus, pengemudi akan diminta untuk mengisi formulir yang merinci tujuan masuk. Informasi ini akan segera dimasukkan oleh petugas keamanan di pintu gerbang dan bersama dengan informasi ini, informasi yang diperoleh dari kamera seperti nomor model kendaraan, nomor registrasi, dan foto akan disimpan dalam database. Ketika kendaraan luar meninggalkan kampus, rincian keluar seperti tanggal dan waktu keluar akan secara otomatis terdaftar dalam database. Untuk setiap kendaraan eksternal yang tetap berada di dalam kampus selama lebih dari 8 jam, pengemudi akan dihentikan oleh petugas keamanan yang berjaga di gerbang, ditanyai kepuasannya, dan tanggapannya akan dimasukkan ke dalam sistem. Mengingat telah terjadi beberapa insiden ngebut dan mengemudi dengan kasar di masa lalu, petugas keamanan yang berjaga di berbagai persimpangan lalu lintas dan titik sensitif lainnya di kampus akan diberdayakan untuk menelepon nomor registrasi kendaraan yang salah ke gerbang utama. Petugas keamanan di gerbang utama akan memasukkan informasi ini ke dalam komputer. Pengemudi kendaraan yang salah akan ditanyai di gerbang pemeriksaan saat keluar dan kendaraan yang akan masuk akan dilarang jika tanggapannya tidak memuaskan. Untuk setiap kendaraan kampus yang salah,

pengemudi akan ditanyai saat pintu keluar berikutnya, dan jika jawabannya tidak memuaskan, surat harus dikeluarkan kepada dekan (urusan kampus) (merinci tanggal, waktu, titik lalu lintas di mana insiden itu terjadi, dan sifat pelanggarannya) untuk menangani staf atau mahasiswa yang bersangkutan, sesuai dengan kasusnya. Petugas keamanan harus dapat melihat statistik yang berkaitan dengan jumlah kendaraan yang masuk dan keluar dari kampus (selama satu hari, bulan atau tahun) dan jumlah kendaraan yang saat ini berada di dalam kampus telah meninggalkan kampus, dan jika memang ada, waktu keberangkatannya harus ditampilkan. Petugas keamanan dapat menanyakan apakah kendaraan tertentu saat ini berada di dalam kampus. Petugas keamanan juga bisa menanyakan jumlah total kendaraan yang dimiliki warga kampus. Karena keamanan kampus adalah operasi kritis, keamanan yang memadai terhadap serangan cyber pada perangkat lunak keamanan harus dipastikan. Juga, mengingat kritisnya operasi, waktu henti lebih dari 5 menit biasanya tidak dapat diterima. Penyederhanaan implementasi: Saat menulis kode, perintah untuk gerbang pemeriksaan hanya perlu ditampilkan di terminal dan masukan dari kamera dapat disimulasikan melalui entri keyboard.

38. Perangkat lunak komputerisasi perusahaan kurir (CCC): Sebuah perusahaan kurir ingin komputerisasi berbagai kegiatan pembukuan yang terkait dengan operasi sehari-hari. Perusahaan kurir memiliki cabang di kota-kota paling penting di India. Diusulkan agar kantor cabang yang berbeda dilengkapi dengan komputer dan printer masing-masing. Perangkat lunak yang dikembangkan akan ditempatkan di komputer di setiap kantor cabang dan terhubung melalui Internet. Rincian lainnya adalah sebagai berikut:
    a. Di setiap kantor cabang dan gerai ritel lainnya, perusahaan kurir menerima kiriman dengan berbagai berat dan ukuran (diukur dalam meter kubik). Biaya saat ini Rp. 5.000 per meter kubik untuk jarak hingga 500 km. Untuk jarak yang lebih jauh, 10 persen dikenakan biaya tambahan untuk setiap 100 km atau sebagian darinya. Untuk paket dengan berat lebih dari 100 kg per meter kubik, dikenakan retribusi tambahan sebesar 10 persen untuk setiap 20 kg/meter kubik. Untuk barang-barang kecil dan surat-surat, Rp. 50 per 100 gram dikenakan biaya. Saat ini, hanya paket-paket yang memiliki tujuan ke kota tempat kantor cabang berada yang diterima.
    b. Ketika seorang pelanggan mencoba untuk memesan kiriman di salah satu titik ritel atau kantor cabang, rincian kiriman seperti volume, berat, alamat tujuan, alamat pengirim, dll dimasukkan ke dalam komputer oleh petugas penjualan. Komputer akan menghitung biaya untuk kiriman dan mencetak tagihan yang menunjukkan id unik yang menunjukkan nomor kiriman, yang diberikan ke kiriman. Pelanggan harus dapat melacak status pengiriman konsinyasi secara online dengan menggunakan id unik.
    c. Perusahaan kurir memiliki sejumlah truk yang digunakan untuk mengangkut kiriman antar kantor cabang.
    d. Ketika volume kiriman untuk tujuan tertentu (kantor cabang) menjadi 500 meter kubik, sistem harus secara otomatis mengalokasikan truk berikutnya yang tersedia di kantor cabang. Karena, tidak ada kiriman yang harus ditunda, setiap kali kiriman tidak dapat dikirim ke tujuannya dalam waktu 3 hari sejak diterima, itu harus secara otomatis diteruskan ke kantor cabang mana pun yang lebih dekat dengan tujuan. Ketika sebuah truk dialokasikan untuk pengiriman kiriman ke kantor cabang, sistem komputer harus mencetak rincian nomor kiriman, volume, nama dan alamat pengirim, serta nama dan alamat penerima.

Hasil cetak ini harus dibawa oleh sopir truk untuk keperluan pemantauan dan pembebasan cukai.

39. Ketika sebuah truk mencapai kantor cabang, status kedatangannya diperbarui. Sebuah truk tetap berada di kantor cabang sampai kantor cabang memiliki muatan yang cukup untuk memuat truk tersebut setidaknya hingga 80 persen dari kapasitasnya. Kantor transportasi di kantor cabang dapat memasukkan biaya bahan bakar dan perbaikan truk.
    a. Pengeluaran rutin perusahaan kurir termasuk gaji staf, biaya sewa kantor cabang, dan biaya perawatan truk. Perusahaan dengan bijaksana menggunakan keuntungannya untuk mendirikan kantor cabang baru dan membeli truk tambahan.
    b. Perangkat lunak harus menjaga rincian setiap karyawan seperti nama, alamat, nomor telepon, gaji pokok, dan tunjangan lainnya. Ini akan membantu manajer rekening di setiap kantor cabang untuk menghasilkan slip gaji semua karyawan setiap bulan dan secara otomatis mengkreditkan gaji ke rekening bank masing-masing.
    c. Semua pembayaran dan penerimaan masuk ke dalam sistem. Rekening laba-rugi konsolidasi (dengan mempertimbangkan semua cabang dan seluruh operasi) diharapkan disiapkan oleh sistem. Manajer harus dapat melihat pendapatan cabang yang dihasilkan, pengiriman yang ditangani, pengeluaran, dll.
    d. Manajer harus dapat melihat status truk yang berbeda setiap saat, mis. kantor cabang di mana ia menunggu, atau dua kantor cabang di mana ia sedang mengangkut kiriman.
    e. Manajer harus dapat melihat penggunaan truk (secara keseluruhan dan juga untuk truk individu) selama periode waktu tertentu. Penggunaan truk harus diberikan dalam hal faktor beban (pemanfaatan kapasitas rata-rata) dan jumlah kilometer yang ditempuh selama periode tertentu.
    f. Manajer perusahaan kurir dapat menanyakan status kiriman tertentu dan rincian volume kiriman yang diangkut antara dua cabang dan pendapatan terkait yang dihasilkan.
    g. Manajer harus dapat melihat masa tunggu rata-rata untuk kiriman selama periode waktu tertentu (hari, bulan, atau tahun) dan untuk berbagai pasangan tujuan sumber. Statistik ini penting bagi manajer, karena dia biasanya memesan truk baru ketika rata-rata masa tunggu kiriman menjadi tinggi karena tidak tersedianya truk. Juga, manajer ingin melihat waktu idle rata-rata truk selama periode waktu tertentu untuk perencanaan masa depan.
    h. Perusahaan kurir ingin perangkat lunak menjadi modular dan sangat dapat dikonfigurasi, sehingga dapat menjual salinan perangkat lunak ke perusahaan kurir lain di negara tersebut.
40. Pusat manajemen aula siswa: Pusat Manajemen Aula (HMC) mahasiswa IIT meminta kita untuk mengembangkan perangkat lunak berikut untuk mengotomatiskan berbagai kegiatan pembukuan yang terkait dengan operasi sehari-harinya.
    a. Setelah seorang mahasiswa masuk, ia akan memberikan catatan dari unit penerimaan kepada petugas di HMC, bersama dengan namanya, alamat tetap, nomor telepon kontak, dan foto. Dia kemudian diberikan aula, dan juga nomor kamar tertentu. Surat yang menunjukkan ruang yang dialokasikan ini harus dikeluarkan untuk mahasiswa yang bersangkutan.

b. Siswa dikenakan biaya kekacauan setiap bulan. Manajer mess akan memasukkan ke perangkat lunak total biaya untuk setiap mahasiswa dalam sebulan pada rekening mess.
c. Setiap kamar memiliki sewa kamar tetap. Kamar-kamarnya memiliki tempat duduk single atau twin-sharing. Aula yang baru dibangun memiliki harga sewa yang lebih tinggi dibandingkan dengan beberapa aula yang lebih tua. Kamar twin sharing memiliki harga sewa yang lebih rendah.
d. Setiap aula menyediakan fasilitas tertentu untuk mahasiswa seperti ruang baca, ruang bermain, ruang TV, dll. Jumlah tetap dikenakan pada setiap mahasiswa dalam hitungan ini.
e. Jumlah total yang dikumpulkan dari setiap mahasiswa aula untuk biaya mess diserahkan kepada manajer mess setiap bulan. Untuk ini, komputer perlu mencetak lembar yang menunjukkan jumlah total yang harus dibayar untuk setiap manajer mess.
f. Cek tercetak dikeluarkan untuk setiap manajer dan tanda tangan diperoleh dari masing-masing pada lembar.
g. Setiap kali seorang mahasiswa datang untuk membayar iurannya, total iurannya harus dihitung sebagai jumlah biaya mess, biaya fasilitas, dan sewa kamar dan ditampilkan. Jumlah tersebut akan dibayar oleh mahasiswa baik dalam bentuk tunai atau cek, dan ini akan dimasukkan oleh petugas rekening ke dalam sistem.
h. Mahasiswa harus dapat menyampaikan berbagai jenis keluhan menggunakan web browser di kamar mereka atau di Lab.
i. Pengaduan tersebut dapat berupa permintaan perbaikan seperti lampu sekering, keran air tidak berfungsi, filter air tidak berfungsi, perbaikan ruangan tertentu, dll. Bisa juga pengaduan mengenai perilaku petugas, petugas mess, dll. operasi jam perangkat lunak diperlukan dan waktu henti harus dapat diabaikan. Mengingat bahwa sekitar 10.000 mahasiswa tinggal di asrama, waktu respons situs web harus dapat diterima bahkan di bawah 1000 klik simultan.
j. HMC menerima hibah tahunan dari Institut untuk pemeliharaan aula, taman, dan menyediakan fasilitas untuk siswa. Ketua HMC harus dilengkapi dengan fungsi yang akan mendukung distribusi hibah di antara aula yang berbeda dan cek harus dicetak berdasarkan hibah yang diberikan ke aula. Pengawas dari aula yang berbeda harus dapat memasukkan rincian pengeluaran mereka terhadap alokasi.
k. Kepala balai masing-masing harus dapat mengetahui total hunian aula dan jumlah kursi kosong. Ia juga harus dapat melihat keluhan yang diajukan oleh mahasiswa dan memposting laporan tindakan yang diambil (ATR) untuk setiap keluhan.
l. Aula mempekerjakan pembantu dan tukang kebun. Karyawan sementara ini menerima gaji tetap setiap hari.
m. Petugas Aula memasuki setiap cuti yang diambil oleh petugas atau tukang kebun dari terminal yang terletak di kantor aula. Pada akhir setiap bulan, daftar gabungan gaji yang harus dibayarkan kepada setiap karyawan aula bersama dengan cek untuk setiap karyawan dicetak.

n. HMC mengeluarkan biaya kecil seperti pekerjaan perbaikan yang dilakukan, langganan koran dan majalah, dll. Jika memungkinkan untuk memasukkan biaya ini harus didebit dari hibah tahunannya.
o. Setiap kali seorang staf baru direkrut, rinciannya termasuk gaji hariannya dimasukkan. Setiap kali seorang staf pergi, catatannya harus dapat dihapus. Atas perintah khusus dari petugas pengendali, gaji untuk berbagai karyawan HMC dan aula harus dihitung dan slip gaji dan cek harus dicetak untuk dibagikan kepada karyawan.
p. Petugas harus dapat melihat laporan rekening setiap saat. Kepala penjara akan mencetak laporan rekening konsolidasi tahunan, menandatangani dan menyerahkannya ke administrasi Institut untuk persetujuan dan verifikasi audit. Perangkat lunak harus sangat aman untuk mencegah kemungkinan berbagai jenis penipuan dan penyimpangan keuangan.

41. Kembangkan diagram SA/SD untuk perangkat lunak berikut yang diperlukan oleh toko persewaan video.
    a. Toko ini memiliki banyak koleksi video CD dan DVD dalam format VHS dan MP4 serta CD musik.
    b. Seseorang dapat menjadi anggota dengan menyetorkan Rp. 1000 dan mengisi nama, alamat, dan nomor telepon. Seorang anggota dapat membatalkan keanggotaannya dan mengambil kembali depositnya, jika dia tidak memiliki iuran terutang terhadapnya.
    c. Setiap kali toko membeli barang baru, rincian seperti tanggal pengadaan dan harga dimasukkan. Biaya sewa harian juga dimasukkan oleh pengelola. Setelah melewati satu tahun, biaya sewa harian secara otomatis dibelah dua.
    d. Seorang anggota dapat meminjam paling banyak satu CD video dan satu CD musik setiap kali. Rinciannya dimasukkan oleh petugas toko dan tanda terima yang menunjukkan biaya sewa harian dicetak.
    e. Setiap kali seorang anggota mengembalikan barang pinjamannya, jumlah yang harus dibayar akan ditampilkan. Setelah jumlah dibayar, item ditandai dikembalikan.
    f. Jika pelanggan kehilangan atau merusak barang apa pun, harga penuh barang tersebut dibebankan kepadanya dan barang tersebut dikeluarkan dari inventaris.
    g. Jika suatu barang tidak diterbitkan selama lebih dari satu tahun, barang tersebut dijual kepada anggota dengan harga 10 persen dari harga pembelian dan barang tersebut dikeluarkan dari inventaris.
    h. Manajer dapat setiap saat memeriksa rekening untung/rugi
42. Kembangkan diagram SA/SD untuk perangkat lunak Elevator Controller berikut. Perangkat lunak pengontrol lift mendapat input dari pengguna lift melalui sakelar tombol tekan yang dipasang di dalam lift dan di dekat pintu lift di setiap lantai. Kontroler menghasilkan output dengan memberikan perintah kepada pengontrol motor dan pengontrol pintu lift. Di setiap lantai, hanya dua sakelar yang dipasang. Sakelar ditandai dengan panah atas dan bawah yang menunjukkan permintaan untuk pergi ke arah atas atau bawah. Di dalam lift, ada panel sakelar, dengan satu sakelar berlabel untuk setiap lantai. Juga ada saklar berhenti darurat. Seorang pengguna di lantai dapat meminta lift dan menunjukkan arah perjalanan yang diperlukan dengan menekan tombol yang sesuai. Permintaan lift yang datang dari berbagai lantai diantrekan oleh pengontrol dan melayani permintaan dalam jarak terpendek terlebih

dahulu, jika lift dalam keadaan idle. Setelah lift berhenti di suatu lantai, pengguna dapat menekan tombol berlabel nomor lantai untuk meminta lift menuju ke lantai tersebut. Setelah tombol permintaan lantai ditekan, pengontrol lift akan habis waktu setelah satu menit dan kemudian mulai menutup pintu lift. Penutupan pintu lift yang berhasil ditunjukkan oleh sinyal yang dihasilkan oleh sensor kontak. Lift mulai bergerak ke arah yang diinginkan setelah pintu lift tertutup sepenuhnya. Pengguna di dalam lift dapat menghentikan pintu lift agar tidak menutup dengan menekan sakelar berhenti darurat sebelum lift mulai bergerak. Setelah lift mulai bergerak, menekan sakelar berhenti darurat tidak akan berpengaruh. Di setiap lantai, ada sensor sentuh yang menunjukkan kepada pengontrol bahwa lift telah mencapai lantai dan pengontrol memerintahkan motor untuk berhenti setelah lantai yang diperlukan tercapai. Setelah 30 detik mencapai lantai, pintu lift dibuka dengan mengeluarkan perintah ke pengontrol pintu. Jika ada kegagalan daya setiap saat selama gerakan lift, lift kembali ke mode aman di mana ia mematikan motor dan pengaturan cadangan mekanis menggeser lift ke lantai dasar dan pegangan pembuka pintu manual diaktifkan, yang pengguna dapat menggunakan untuk membuka pintu lift.

BAB 7
PEMODELAN OBJEK MENGGUNAKAN UML

Dalam beberapa tahun terakhir, gaya pengembangan perangkat lunak berorientasi objek telah menjadi sangat populer dan saat ini banyak digunakan di industri maupun di kalangan akademis. Sejak dimulai pada awal tahun delapan puluhan, teknologi objek telah mengalami kemajuan pesat. Dari awal yang sederhana di awal tahun delapan puluhan, kemajuan teknologi objek mengumpulkan momentum di tahun sembilan puluhan dan teknologi sekarang mendekati kedewasaan. Mengingat luasnya penggunaan dan popularitas teknologi objek baik di industri maupun akademisi, penting untuk mempelajari teknologi ini dengan baik.

Diketahui bahwa menguasai bahasa pemrograman berorientasi objek seperti Java atau C++ jarang melengkapi seseorang dengan keterampilan yang diperlukan untuk mengembangkan perangkat lunak berorientasi objek yang berkualitas baik—penting untuk mempelajari keterampilan desain berorientasi objek dengan baik. Setelah desain yang baik telah tercapai, mudah untuk mengkodekannya menggunakan bahasa berorientasi objek. Sekarang bahkan menjadi mungkin untuk secara otomatis menghasilkan banyak kode dari desain dengan menggunakan alat CASE. Untuk sampai pada solusi desain berorientasi objek (OOD) yang memuaskan untuk suatu masalah, perlu untuk membuat beberapa jenis model. Tapi, orang mungkin bertanya: "Apa hubungannya pemodelan dengan desain?" Mari kita jawab pertanyaan ini sebagai berikut:

Sebuah model dibangun dengan berfokus hanya pada beberapa aspek dari masalah dan mengabaikan sisanya. Model suatu masalah disebut model analisis. Di sisi lain, model solusi (kode) disebut model desain. Model desain biasanya diperoleh dengan melakukan penyempurnaan berulang terhadap model analisis menggunakan metodologi desain. Perhatikan bahwa setiap desain adalah model dari solusi, sedangkan setiap model dari masalah adalah model analisis. Dalam bab ini, kita akan membahas bagaimana mendokumentasikan model menggunakan bahasa pemodelan. Dalam bab berikutnya, kita akan membahas proses desain yang dapat digunakan untuk menyempurnakan model analisis menjadi model desain secara iteratif. Dalam konteks konstruksi model, kita perlu memahami dengan cermat perbedaan antara bahasa pemodelan dan proses desain, karena kita akan sering menggunakan kedua istilah ini dalam diskusi kita.

- **Bahasa pemodelan**: Bahasa pemodelan terdiri dari serangkaian notasi yang menggunakan model desain dan analisis yang didokumentasikan.
- **Proses desain:** Proses desain membahas masalah berikut: "Dengan deskripsi masalah, bagaimana cara sistematis mencari solusi desain untuk masalah itu?" Dengan kata lain, proses desain terdiri dari prosedur langkah demi langkah (atau resep) yang dengannya deskripsi masalah dapat diubah menjadi solusi desain. Sebuah proses desain, kadang-kadang, juga disebut sebagai metodologi desain. Dalam teks ini, kita akan menggunakan istilah proses desain dan metodologi desain secara bergantian.

Sebuah model dapat didokumentasikan menggunakan bahasa pemodelan seperti bahasa pemodelan terpadu (UML). Selama dekade terakhir, UML telah menjadi sangat populer. UML juga telah diterima oleh ISO sebagai standar untuk pemodelan sistem berorientasi objek. Dalam Bab ini, kita membahas sintaks dan semantik UML. Namun, sebelum membahas seluk beluk sintaks dan semantik UML, kita haru meninjau beberapa konsep dasar dan terminologi yang telah dikaitkan dengan orientasi objek.

7.1 KONSEP OBJEK-ORIENTASI DASAR

Prinsip-prinsip orientasi objek telah didasarkan pada beberapa konsep sederhana. Konsep-konsep ini secara bergambar ditunjukkan pada Gambar 7.1. Setelah membahas konsep dasar ini, kita akan memeriksa beberapa istilah teknis terkait.

Gambar 7.1 Konsep-konsep penting yang digunakan dalam pendekatan berorientasi objek.

Konsep dasar

Beberapa konsep penting yang membentuk landasan paradigma berorientasi objek secara bergambar telah ditunjukkan pada Gambar 7.1.

Objek

Dalam pendekatan berorientasi objek, akan lebih mudah untuk membayangkan kerja perangkat lunak dalam kaitannya dengan sekumpulan objek yang berinteraksi. Ini analog dengan cara manipulasi objek terjadi dalam sistem manual dunia nyata untuk menyelesaikan beberapa pekerjaan. Misalnya, pertimbangkan sistem perpustakaan yang dioperasikan secara manual. Untuk menerbitkan buku, daftar penerbitan perlu diisi dan kemudian tanggal pengembalian perlu dicap pada buku. Dalam perangkat lunak otomatisasi perpustakaan berorientasi objek, aktivitas analog yang melibatkan objek buku dan objek register masalah berlangsung.

Setiap objek dalam program berorientasi objek biasanya mewakili entitas dunia nyata yang nyata seperti anggota perpustakaan, buku, register masalah, dll. Namun saat memecahkan masalah, terkadang menjadi menguntungkan untuk mempertimbangkan entitas konseptual tertentu (mis., penjadwal, pengontrol, dll.) sebagai objek juga. Ini menyederhanakan solusi dan membantu mencapai desain yang baik. Keuntungan utama dari mempertimbangkan sistem sebagai satu set objek adalah sebagai berikut:

Ketika sistem dianalisis, dikembangkan, dan diimplementasikan dalam bentuk objek, menjadi mudah untuk memahami desain dan implementasi sistem, karena objek memberikan dekomposisi yang sangat baik dari masalah besar menjadi bagian-bagian kecil. Setiap objek pada dasarnya terdiri dari beberapa data yang bersifat pribadi untuk objek dan satu set fungsi (disebut sebagai operasi atau metode) yang beroperasi pada data tersebut. Aspek ini secara bergambar telah diilustrasikan pada Gambar 7.2. Perhatikan bahwa data objek hanya dapat diakses oleh metode objek. Akibatnya, satu-satunya jalur akses ke data untuk objek eksternal adalah melalui pemanggilan metode objek. Faktanya, metode suatu objek memiliki otoritas tunggal untuk beroperasi pada data yang bersifat pribadi untuk objek tersebut. Dengan kata

lain, tidak ada objek yang dapat mengakses data objek lain secara langsung. Oleh karena itu, setiap objek dapat dianggap menyembunyikan data internalnya dari objek lain. Namun, suatu objek dapat mengakses data pribadi objek lain dengan memanggil metode yang didukung oleh objek tersebut. Mekanisme penyembunyian data dari objek lain ini dikenal sebagai prinsip penyembunyian data atau abstraksi. Penyembunyian data mempromosikan kohesi tinggi dan kopling rendah di antara objek, dan karena itu dianggap sebagai prinsip penting yang dapat membantu seseorang untuk sampai pada desain yang cukup baik.

Seperti yang telah disebutkan, setiap objek menyimpan beberapa data dan mendukung operasi tertentu pada data yang disimpan. Sebagai contoh, pertimbangkan objek libraryMember dari aplikasi otomatisasi perpustakaan. Data pribadi objek libraryMember dapat berupa:

- nama anggota
- Nomor keanggotaan
- alamat
- nomor telepon
- alamat email
- tanggal saat diterima sebagai anggota
- tanggal kedaluwarsa keanggotaan
- buku luar biasa

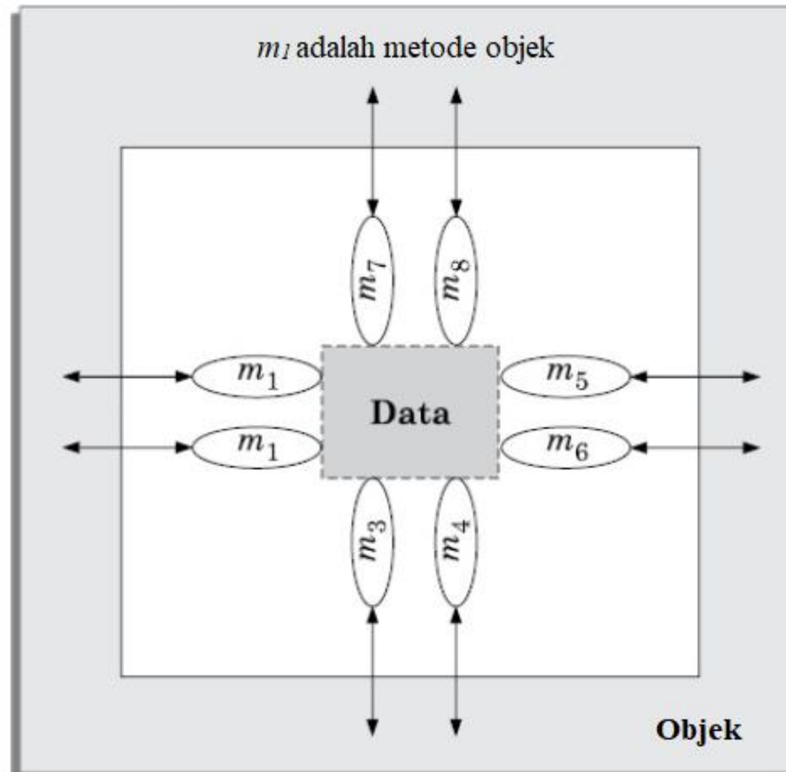
Operasi yang didukung oleh objek libraryMember dapat berupa:

- buku edisi
- temukan-buku-luar biasa
- temukan-buku-terlambat
- buku kembali
- temukan-keanggotaan-detail

Data yang disimpan secara internal dalam suatu objek disebut atributnya, dan operasi yang didukung oleh objek disebut metodenya. Meskipun terminologi yang terkait dengan orientasi objek sederhana dan terdefinisi dengan baik, sebuah kata peringatan di sini: istilah 'objek' sering digunakan agak longgar dalam literatur dan juga dalam teks ini. Seringkali 'objek' berarti satu entitas. Namun, di lain waktu, kita akan menggunakannya untuk merujuk ke sekelompok objek (kelas) yang serupa. Dalam teks ini, biasanya konteks penggunaan akan menyelesaikan ambiguitas, jika ada.

Benarkah program berorientasi objek hanya terdiri dari objek?

Sejauh ini, kita mengklaim bahwa program berorientasi objek hanya terdiri dari objek dan eksekusi program melibatkan interaksi objek-objek ini. Namun, program yang ditulis menggunakan bahasa pemrograman saat ini seperti Java, C++, dll. bekerja melalui interaksi objek dan data primitif. Bahasa pemrograman ini menganggap bahwa data primitif bukanlah objek dan perbedaan dibuat antara ini dalam beberapa cara. Untuk mendapatkan perspektif ini, kita perlu memeriksa sedikit sejarah pemrograman.



Gambar 7.2 Model suatu objek

Bahasa pemrograman berorientasi objek pertama adalah Smalltalk. Ini dikembangkan pada 1970-an di pusat penelitian Xerox di AS. Itu adalah bahasa berorientasi objek murni dalam arti bahwa aplikasi yang dikembangkan dalam bahasa ini hanya terdiri dari objek. Dengan kata lain, tidak ada perbedaan yang dibuat antara objek dan contoh tipe data primitif (int, float, dll.). Ini dimaksudkan untuk mendapatkan penerimaan programmer yang lebih baik. Obrolan ringan tidak diterima secara luas di antara para pemrogram, karena itu mengharuskan programmer untuk belajar dan menggunakan paradigma pengembangan program yang sama sekali berbeda. Namun, dalam bahasa pemrograman generasi selanjutnya seperti C++ dan Java, contoh tipe data primitif diperlakukan berbeda dari objek.

Misalnya, dalam bahasa ini, objek dilewatkan sebagai argumen referensi ke metode, sedangkan tipe data primitif diteruskan dengan nilai. Motivasi di balik perbedaan ini adalah untuk membuat bahasa berorientasi objek muncul sebagai perluasan kecil untuk bahasa prosedural daripada membawa pendekatan baru yang radikal. Dalam hal ini, C++ dirancang untuk mempertahankan pendekatan prosedural C dan hanya memperluasnya dengan konstruksi orientasi objek. Tentu saja, konstruksi berorientasi objek yang berbeda diterjemahkan ke dalam kode C oleh preprocessor. Ini memudahkan programmer prosedural untuk bermigrasi ke C++. Ini mungkin alasan penting mengapa C++ mendapatkan popularitas yang jauh lebih besar dibandingkan dengan bahasa pemrograman berorientasi objek murni. Saat ini, bahasa pemrograman berorientasi objek seperti Java dan C++ membedakan tipe data primitif dari objek dan memperlakukannya dengan sangat berbeda.

Kelas

Objek serupa membentuk kelas. Artinya, semua objek yang membentuk kelas memiliki atribut dan metode yang serupa. Misalnya, himpunan semua anggota perpustakaan akan membentuk kelas LibraryMember dalam aplikasi otomatisasi perpustakaan. Dalam hal ini, setiap objek anggota perpustakaan memiliki atribut seperti nama anggota, nomor keanggotaan, alamat anggota, dll. Dan juga memiliki metode seperti buku terbitan, buku

kembali, dll. Setelah kita mendefinisikan sebuah kelas, itu dapat digunakan sebagai template untuk pembuatan objek.

Sekarang mari kita selidiki pertanyaan penting apakah suatu kelas adalah tipe data abstrak (ADT). Untuk dapat menjawab pertanyaan ini, pertama-tama kita harus mengetahui definisi dasar dari ADT. Pertama-tama kita akan membahas hal yang sama secara singkat. Ada dua hal yang melekat pada ADT—data abstrak dan tipe data. Dalam teori bahasa pemrograman, tipe data mengidentifikasi sekelompok variabel yang memiliki perilaku tertentu. Tipe data dapat dipakai untuk membuat variabel. Misalnya, int adalah tipe dalam bahasa C++. Ketika kita menulis sebuah instruksi int i; sebuah instance dari int akan benar-benar dibuat. Dari sini, kita dapat menyimpulkan sebagai berikut— ADT adalah tipe di mana data yang terkandung dalam setiap entitas yang diinstansiasi diabstraksikan (disembunyikan) dari entitas lain. Sekarang mari kita periksa apakah suatu kelas mendukung dua mekanisme ADT.

Data abstrak: Data suatu objek hanya dapat diakses melalui metodenya. Dengan kata lain, cara yang tepat untuk menyimpan data secara internal (tumpukan, larik, antrian, dll.) dalam objek diabstraksikan (tidak diketahui oleh objek lain).

Tipe data: Dalam terminologi bahasa pemrograman, tipe data mendefinisikan kumpulan nilai data dan satu set operasi yang telah ditentukan sebelumnya pada nilai-nilai tersebut. Dengan demikian, tipe data dapat dipakai untuk membuat variabel tipe itu. Kita bisa membuat instance kelas menjadi objek. Oleh karena itu, kelas adalah tipe data.

Dapat disimpulkan dari diskusi di atas bahwa kelas adalah ADT. Namun, ADT tidak perlu berupa kelas, karena untuk menjadi kelas mereka harus mendukung pewarisan dan properti orientasi objek lainnya.

Metode

Operasi (seperti create, issue, return, dll.) yang didukung oleh sebuah objek diimplementasikan dalam bentuk metode. Perhatikan bahwa kita membedakan antara istilah operasi dan metode. Meskipun istilah 'operasi' dan 'metode' terkadang digunakan secara bergantian, ada perbedaan teknis antara kedua istilah ini yang akan dijelaskan berikut ini.

Operasi adalah tanggung jawab khusus kelas, dan tanggung jawab diimplementasikan menggunakan metode. Namun, ada kalanya berguna untuk memiliki beberapa metode untuk mengimplementasikan tanggung jawab. Dalam hal ini, semua metode memiliki nama yang sama (yaitu, nama operasi), tetapi daftar parameter setiap metode harus berbeda agar kompiler dapat menentukan metode yang tepat untuk diikat pada pemanggilan metode. Oleh karena itu nama metode kelebihan beban dengan beberapa implementasi operasi. Sebagai contoh, perhatikan berikut ini. Misalkan salah satu tanggung jawab kelas bernama Circle adalah membuat instance dari dirinya sendiri. Asumsikan bahwa kelas menyediakan tiga definisi untuk operasi create—int create(), int create(int radius) dan int create(float x, float y, int radius);. Dalam hal ini, create adalah metode yang kelebihan beban. Implementasi tanggung jawab suatu kelas melalui beberapa metode dengan nama metode yang sama disebut metode overloading.

Metode adalah satu-satunya cara yang tersedia untuk objek lain dalam perangkat lunak untuk mengakses dan memanipulasi data objek lain. Kumpulan pesan yang valid ke suatu objek merupakan protokolnya. Sekarang mari kita coba memahami perbedaan antara pesan dan metode. Di Smalltalk, sebuah objek dapat meminta layanan (yaitu, memanggil metode) objek lain dengan mengirimkan pesan kepada mereka. Idenya adalah bahwa mekanisme pengiriman pesan akan menyebabkan kopling yang lemah di antara objek. Meskipun ini adalah fitur penting dari Smalltalk, namun para programmer yang mencoba untuk bermigrasi dari pemrograman prosedural ke pemrograman berorientasi objek,

menganggapnya sebagai perubahan paradigma dan oleh karena itu sulit untuk diterima. Selanjutnya, C++ mengimplementasikan pesan lewat pemanggilan metode (mirip dengan pemanggilan fungsi). Ini dengan cepat diterima oleh para programmer. Kemudian bahasa berorientasi objek seperti Java telah mengikuti sifat yang sama mempertahankan fitur pemanggilan metode, yang biasanya dikaitkan dengan bahasa prosedural.

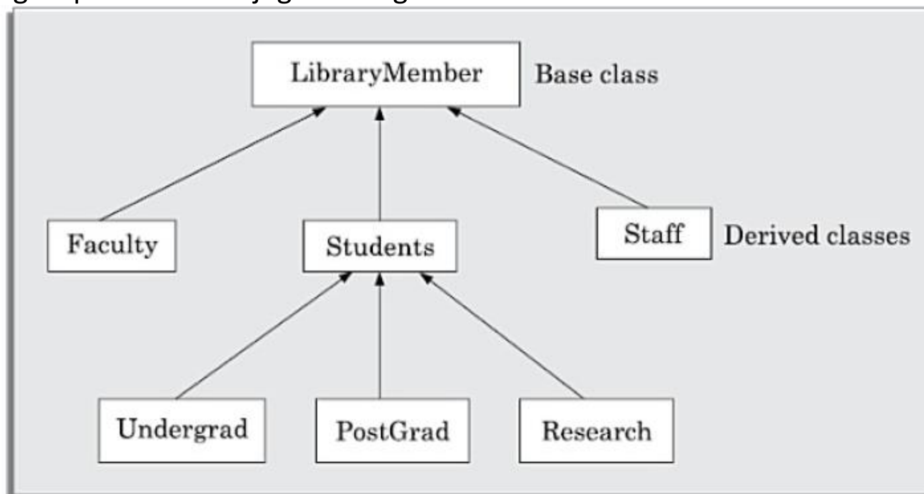
Hubungan Kelas

Kelas dalam solusi pemrograman dapat dihubungkan satu sama lain dalam empat cara berikut:

- Warisan
- Asosiasi dan tautan
- Agregasi dan komposisi
- Ketergantungan

Warisan

Fitur pewarisan memungkinkan seseorang untuk mendefinisikan kelas baru dengan secara bertahap memperluas fitur dari kelas yang ada. Kelas asli disebut kelas dasar (juga disebut kelas super atau kelas induk) dan kelas baru yang diperoleh melalui pewarisan disebut kelas turunan (juga disebut subkelas atau kelas anak). Kelas turunan dikatakan mewarisi fitur dari kelas dasar. Contoh pewarisan ditunjukkan pada Gambar 7.3. Pada Gambar 7.3, perhatikan bahwa kelas Fakultas, Mahasiswa, dan Staf telah diturunkan dari LibraryMember kelas dasar melalui hubungan pewarisan (perhatikan jenis panah khusus yang digunakan untuk menggambarnya). Hubungan pewarisan antara anggota perpustakaan dan fakultas alternatif dapat dinyatakan sebagai berikut-A fakultas adalah jenis anggota perpustakaan. Jadi, hubungan pewarisan ini juga kadang disebut relasi.



Gambar 7.3 Contoh sistem informasi perpustakaan.

Sebuah kelas dasar dikatakan sebagai generalisasi dari kelas turunannya. Ini berarti bahwa kelas dasar harus hanya berisi properti-properti (yaitu, data dan metode) yang umum untuk semua kelas turunannya. Misalnya, pada Gambar 7.3 setiap kelas turunan mendukung metode buku terbitan, dan metode ini juga didukung oleh kelas dasar. Dengan kata lain, setiap kelas turunan adalah kelas dasar khusus yang memperluas fungsionalitas kelas dasar dengan cara tertentu. Setiap kelas turunan dapat dianggap sebagai spesialisasi dari kelas dasarnya karena memodifikasi atau memperluas sifat dasar dari kelas dasar dengan cara tertentu. Oleh karena itu, hubungan pewarisan dapat dipandang sebagai hubungan generalisasi-spesialisasi.

Perhatikan bahwa pada Gambar 7.3 kelas-kelas Fakultas, Mahasiswa, dan Staf adalah semua tipe khusus anggota perpustakaan. Beberapa hal yang umum di antara para anggota.

Ini termasuk atribut seperti id keanggotaan, nama dan alamat anggota, tanggal penerbitan buku, dll. Namun, untuk kategori anggota yang berbeda, prosedur penerbitannya berbeda karena jenis anggota yang berbeda menerbitkan buku untuk jangka waktu yang berbeda. Sebisa mungkin dapat saya sampaikan bahwa kelas-kelas Fakultas, Staf, dan Mahasiswa adalah kelas-kelas khusus Anggota Perpustakaan. Menggunakan hubungan pewarisan, kelas yang berbeda dapat diatur dalam hierarki kelas (atau pohon kelas) seperti yang ditunjukkan pada Gambar 7.3.

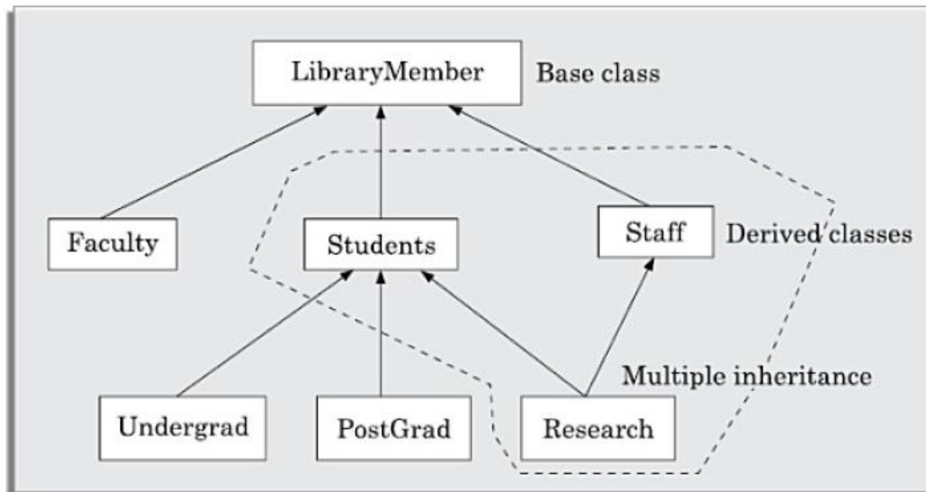
Selain mewarisi semua data dan metode dari kelas dasar, kelas turunan biasanya mendefinisikan beberapa data dan metode baru. Kelas turunan bahkan dapat mendefinisikan kembali metode yang sudah ada di kelas dasar. Redefinisi metode yang sudah ada di kelas dasarnya disebut sebagai metode overriding. Ketika definisi baru dari metode yang ada di kelas dasar disediakan di kelas turunan, metode tersebut dikatakan ditimpa di kelas turunan.

Hubungan pewarisan yang ada di antara kelas-kelas tertentu dalam sistem otomasi perpustakaan ditunjukkan pada Gambar 7.3. Seperti yang ditunjukkan, LibraryMember adalah base class untuk kelas turunan Fakultas, Mahasiswa, dan Staf. Demikian pula Student adalah kelas dasar untuk kelas turunan Undergrad, Postgrad, dan Research. Setiap kelas turunan mewarisi semua data dan metode dari kelas dasar, dan mendefinisikan beberapa data dan metode tambahan atau memodifikasi beberapa data dan metode yang diwarisi. Hubungan pewarisan telah direpresentasikan pada Gambar 7.3 menggunakan panah terarah yang ditarik dari kelas turunan ke kelas dasarnya. Metode kelas dasar ditimpa oleh kelas turunan diilustrasikan pada Gambar 7.3, LibraryMember kelas dasar mungkin mendefinisikan data berikut—nama anggota, alamat, dan nomor keanggotaan perpustakaan. Meskipun kelas fakultas, mahasiswa, dan staf mewarisi data ini, mereka perlu mendefinisikan kembali metode penerbitan buku mereka masing-masing karena untuk perpustakaan tertentu yang dimodelkan, jumlah buku yang dapat dipinjam dan durasi peminjaman berbeda untuk kategori perpustakaan yang berbeda. anggota.

Warisan adalah mekanisme dasar yang harus didukung oleh setiap bahasa pemrograman berorientasi objek. Jika suatu bahasa mendukung ADT, tetapi tidak mendukung pewarisan, maka itu disebut bahasa berbasis objek dan bukan berorientasi objek. Contoh bahasa pemrograman berbasis objek adalah Ada. Sekarang mari kita coba memahami mengapa kita membutuhkan hubungan warisan sejak awal. Tidak bisakah kita memprogram juga tanpa menggunakan hubungan pewarisan? Dua keuntungan penting menggunakan mekanisme pewarisan dalam pemrograman termasuk penggunaan kembali kode dan kesederhanaan desain program.

Mari kita periksa bagaimana penggunaan kembali kode dan kesederhanaan desain terjadi saat menggunakan mekanisme pewarisan. Jika metode atau data tertentu ada di beberapa kelas, maka alih-alih mendefinisikan metode dan data ini di setiap kelas secara terpisah, metode dan data ini hanya didefinisikan satu kali di kelas dasar dan kemudian diwarisi oleh masing-masing subkelasnya. Misalnya, pada contoh Sistem Informasi Perpustakaan pada Gambar 7.4, setiap kategori anggota (yaitu, Fakultas, Mahasiswa, dan Staf) perlu menyimpan data berikut —nama-anggota, alamat-anggota, dan nomor-anggota. Oleh karena itu, data ini didefinisikan dalam kelas dasar LibraryMember dan diwarisi oleh semua subkelasnya. Keuntungan lain yang diperoleh dari penggunaan mekanisme pewarisan adalah penyederhanaan konseptual yang dibawa melalui pengurangan jumlah fitur independen dari kelas yang berbeda dan pemahaman tambahan dari kelas yang berbeda menjadi mungkin. Jadi, pewarisan dapat dianggap sebagai penggunaan mekanisme abstraksi yang telah kita bahas di Bab 1. Kelas pada akar hierarki pewarisan (misalnya LibraryMember pada Gambar 7.3) adalah yang paling sederhana untuk dipahami— karena memiliki jumlah data paling

sedikit dan anggota metode dibandingkan dengan semua kelas lain dalam hierarki. Kelas-kelas di tingkat daun hierarki pewarisan memiliki jumlah fitur maksimum (anggota data dan metode) karena mereka mewarisi fitur dari semua leluhurnya, dan karena itu menjadi yang paling sulit untuk dipahami. Dalam hierarki kelas besar, lebih mudah untuk terlebih dahulu memahami kelas akar dan kemudian secara rekursif memahami kelas turunannya dalam hierarki hingga kelas tingkat daun tercapai.



Gambar 7.4 Contoh pewarisan berganda.

Warisan berganda

Konstruksi hubungan kelas untuk masalah yang diberikan terdiri dari mengidentifikasi dan mewakili empat jenis hubungan — pewarisan, komposisi/agregasi, asosiasi, dan ketergantungan. Namun, kadang-kadang mungkin terjadi bahwa beberapa fitur dari suatu kelas mirip dengan satu kelas dan beberapa fitur lain dari kelas tersebut serupa dengan yang ada di kelas lain. Dalam hal ini, akan berguna jika kelas diizinkan untuk mewarisi fitur dari kedua kelas. Menggunakan fitur pewarisan berganda, sebuah kelas dapat mewarisi fitur dari beberapa kelas dasar. Multiple inheritance adalah mekanisme dimana subclass dapat mewarisi atribut dan metode dari lebih dari satu kelas dasar.

Perhatikan contoh kelas berikut yang diturunkan dari dua kelas dasar melalui penggunaan mekanisme pewarisan berganda. Dalam contoh ini, di lembaga akademik mahasiswa penelitian juga dapat dipekerjakan sebagai staf lembaga, maka beberapa karakteristik kelas Penelitian mirip dengan kelas mahasiswa (misalnya, setiap mahasiswa akan memiliki nomor gulung) sementara beberapa karakteristik lainnya (misalnya, memiliki gaji pokok dan nomor karyawan, dll.) mungkin mirip dengan kelas Staf. Menggunakan pewarisan berganda kelas Penelitian dapat mewarisi fitur dari kelas mahasiswa dan Staf. Gambar 7.4 menunjukkan kelas Penelitian diturunkan dari kelas mahasiswa dan Staf dengan menggambar panah pewarisan ke kedua kelas induk dari kelas Penelitian.

Asosiasi dan tautan

Asosiasi adalah jenis umum dari hubungan antar kelas. Ketika dua kelas diasosiasikan, mereka dapat saling membantu (yaitu memanggil metode satu sama lain) untuk melayani permintaan pengguna. Secara lebih teknis, kita dapat mengatakan bahwa jika satu kelas dikaitkan dengan yang lain secara dua arah, maka objek yang sesuai dari dua kelas saling mengenal id (identitas). Akibatnya, menjadi mungkin bagi objek dari satu kelas untuk memanggil metode dari objek yang sesuai dari kelas lain. Perhatikan contoh berikut.

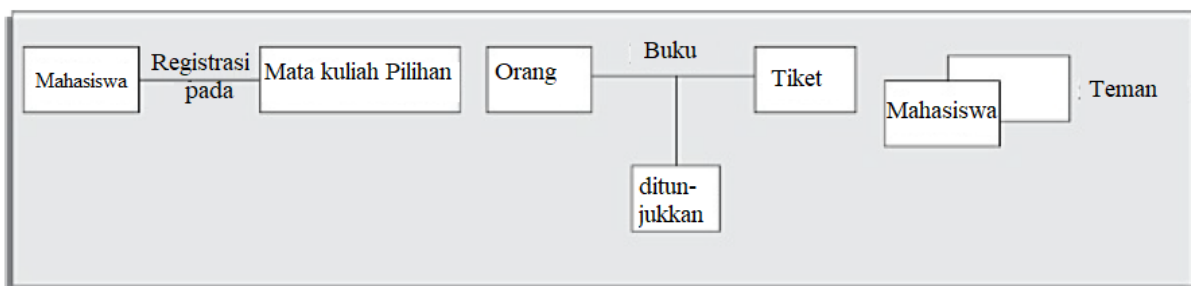
Seorang mahasiswa dapat mendaftar dalam satu mata pelajaran pilihan. Dalam contoh ini, kelas Student diasosiasikan dengan kelas ElectiveSubject. Oleh karena itu, objek

ElectiveSubject (misalnya MachineLearning) akan mengetahui id semua objek mahasiswa yang telah terdaftar untuk Subjek dan dapat memanggil metodenya seperti printName, printRoll, dan enterGrade. Hubungan ini telah direpresentasikan dalam Gambar 6(a). Ketika suatu objek mengetahui beberapa objek lain, ia harus menyimpan id-nya secara internal. Misalnya, untuk objek kelas ElectiveSubject e1 untuk memanggil metode printName() salah satu mahasiswa terdaftar s1 , itu harus mengeksekusi kode s1 .printName(). Dengan demikian, seharusnya menyimpan id s1 dari mahasiswa yang terdaftar sebagai atribut. Hubungan asosiasi dapat bersifat dua arah atau searah. Artinya, kedua kelas terkait saling mengenal (saling menyimpan id). Secara grafis telah menunjukkan hubungan antara kelas mahasiswa dan Mata Pelajaran Pilihan pada Gambar 7.5(a).

Pertimbangkan contoh lain dari asosiasi antara dua kelas: Anggota Perpustakaan meminjam Buku. Di sini, meminjam adalah hubungan antara kelas LibraryMember dan kelas Book. Hubungan asosiasi akan menyiratkan bahwa dengan memberikan sebuah buku, adalah mungkin untuk menentukan peminjam dan sebaliknya.

Asosiasi n-ary

Asosiasi biner antar kelas sangat umum ditemui dalam masalah desain. Namun, mungkin ada situasi di mana tiga atau lebih kelas yang berbeda dapat terlibat dalam sebuah asosiasi. Sebagai contoh asosiasi ternary, pertimbangkan hal berikut—Seseorang memesan tiket untuk pertunjukan tertentu. Di sini, ada asosiasi di antara kelas Person, Ticket, dan Show. Contoh hubungan asosiasi terner ini secara gambar ditunjukkan pada Gambar 7.5(b).



Gambar 7.5 Contoh (a) biner (b) ternary (c) asosiasi unary.

Sebuah kelas dapat memiliki hubungan asosiasi dengan dirinya sendiri. Ini disebut asosiasi rekursif atau asosiasi tunggal. Sebagai contoh, pertimbangkan yang berikut—dua mahasiswa mungkin berteman. Di sini, sebuah asosiasi bernama persahabatan ada di antara pasangan objek dari kelas Siswa. Ini telah diperlihatkan dalam Gambar 7.5(c). Dalam asosiasi unary, dua (atau lebih) objek yang berbeda dari kelas yang sama dihubungkan oleh hubungan asosiasi. Ketika dua kelas diasosiasikan, hubungan antara dua objek dari kelas yang bersesuaian disebut link. Sebuah asosiasi menggambarkan sekelompok link serupa. Atau, kita dapat mengatakan bahwa tautan dapat dianggap sebagai turunan dari hubungan asosiasi. Sekarang mari kita coba mengidentifikasi hubungan asosiasi dari teks deskripsi suatu masalah.

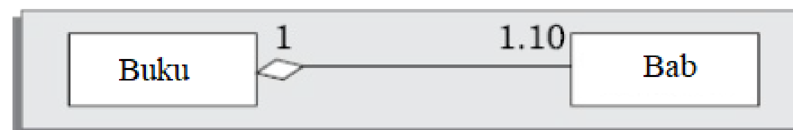
Contoh 7.1 Perhatikan kutipan berikut dari deskripsi masalah. Identifikasi hubungan asosiasi antar kelas dan hubungan asosiasi yang sesuai di antara objek dari analisis deskripsi. "Seseorang bekerja untuk sebuah perusahaan. Ram bekerja untuk Infosys. Hari bekerja untuk TCS."

Jawaban: Dalam contoh ini, hubungan asosiasi bernama works for ada di antara kelas Person a n d Company. R a m berfungsi untuk Infosys, ini menyiratkan bahwa ada tautan antara objek Ram dan objek Infosys. Demikian pula, bekerja untuk link ada antara objek Hari dan TCS.

Selama menjalankan sistem, tautan baru dapat terbentuk di antara objek-objek dari kelas terkait dan beberapa tautan yang ada dapat dibubarkan. Perhatikan bahwa beberapa objek mungkin tidak memiliki tautan asosiasi ke objek mana pun dari kelas terkait. Beberapa objek dari kelas terkait mungkin tidak memiliki tautan. Misalnya dalam perjalanan waktu, Ram dapat mengundurkan diri dari Infosys dan bergabung dengan Wipro. Dalam hal ini, hubungan antara Ram dan Infosys terputus dan hubungan antara Ram dan Wipro terbentuk. Tapi, anggaplah Ram tidak bergabung dengan pekerjaan lain setelah meninggalkan Infosys. Dalam hal ini, Ram tidak memiliki karya untuk ditautkan dengan objek Perusahaan apa pun. Dengan demikian, tautan bersifat waktu yang bervariasi (atau dinamis). Hubungan asosiasi antara dua kelas bersifat statis. Jika dua kelas diasosiasikan, maka hubungan asosiasi ada di semua titik waktu. Sebaliknya, hubungan antar objek bersifat dinamis. Tautan antara objek dari kelas terkait dapat terbentuk dan dibubarkan saat program dijalankan. Sebuah asosiasi antara dua kelas berarti bahwa nol atau lebih link dapat hadir di antara objek dari kelas terkait setiap saat selama eksekusi. Secara matematis, tautan dapat dianggap sebagai tupel. Perhatikan contoh berikut. “Amit telah meminjam buku Graph Theory.” Di sini, tautan bernama pinjaman telah dibuat antara objek Amit dan buku Teori Grafik. Tautan ini juga dapat dinyatakan sebagai pasangan terurut dari instance objek {Amit,Graph Theory}.

Komposisi dan agregasi

Komposisi dan agregasi mewakili sebagian/keseluruhan hubungan antar objek. Objek yang berisi objek lain disebut objek komposit. Sebagai contoh, pertimbangkan hal berikut—Sebuah objek Buku dapat memiliki hingga sepuluh Bab. Dalam hal ini, objek Buku dikatakan terdiri dari hingga sepuluh objek Bab. Hubungan komposisi/agregasi juga dapat dibaca sebagai berikut—Sebuah Buku memiliki hingga sepuluh objek Bab (ditunjukkan pada Gambar 7.6). Oleh karena itu, hubungan komposisi/agregasi disebut juga memiliki hubungan. Agregasi/komposisi dapat terjadi dalam hierarki level. Artinya, suatu objek yang terkandung dalam objek lain mungkin sendiri berisi beberapa objek lain. Hubungan komposisi dan agregasi tidak bisa refleksif. Artinya, suatu objek tidak dapat berisi objek dengan tipe yang sama dengan dirinya sendiri.



Gambar 7.6 Contoh hubungan agregasi.

Ketergantungan

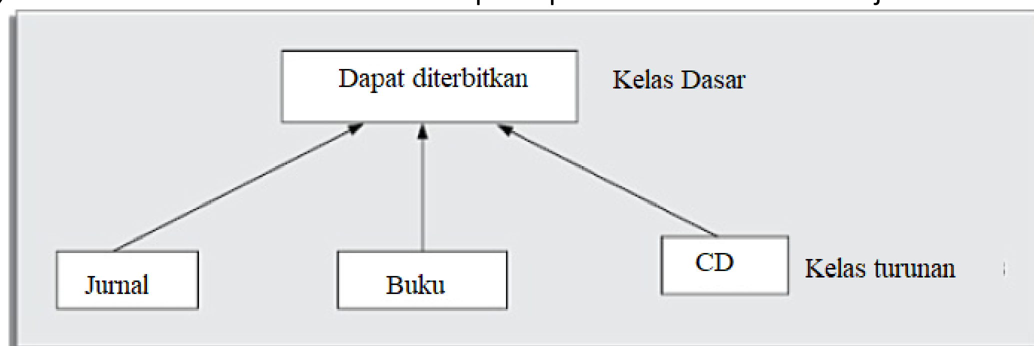
Sebuah kelas dikatakan bergantung pada kelas lain, jika ada perubahan pada kelas yang terakhir mengharuskan perubahan dilakukan pada kelas dependen. Hubungan ketergantungan antara dua kelas menunjukkan bahwa setiap perubahan yang dibuat pada kelas independen akan memerlukan perubahan yang sesuai untuk dilakukan pada kelas dependen. Ketergantungan antar kelas dapat muncul karena berbagai penyebab. Dua alasan penting adanya ketergantungan antara dua kelas adalah sebagai berikut:

- Sebuah metode kelas mengambil objek dari kelas lain sebagai argumen.
- Sebuah kelas mengimplementasikan sebuah kelas antarmuka. Dalam hal ini, ketergantungan muncul karena alasan berikut. Jika beberapa properti dari kelas antarmuka diubah, maka perubahan juga diperlukan untuk kelas yang mengimplementasikan kelas antarmuka.

Kelas abstrak

Kelas yang tidak dimaksudkan untuk menghasilkan instance dari dirinya sendiri disebut kelas abstrak. Dengan kata lain, kelas abstrak tidak dapat dipakai. Jika kelas abstrak tidak dapat dipakai untuk membuat objek, lalu apa gunanya mendefinisikan kelas abstrak? Kelas abstrak hanya ada sehingga perilaku umum untuk berbagai kelas dapat difaktorkan ke dalam satu lokasi umum, di mana mereka dapat didefinisikan satu kali. Definisi kelas abstrak membantu mendorong kode yang dapat digunakan kembali ke dalam hierarki kelas, sehingga meningkatkan penggunaan kembali kode. Dengan menggunakan kelas abstrak, penggunaan kembali kode dapat ditingkatkan dan upaya yang diperlukan untuk mengembangkan perangkat lunak diturunkan.

Kelas abstrak biasanya mendukung metode generik. Metode ini membantu untuk menstandarisasi nama metode dan parameter input dan output di kelas turunan. Subkelas dari kelas abstrak diharapkan menyediakan implementasi konkret untuk metode ini. Misalnya, dalam perangkat lunak otomasi perpustakaan Issuable dapat berupa kelas abstrak (lihat Gambar 7.7) dan kelas konkret Book, Journal, dan CD diturunkan dari kelas Issuable. Kelas Issuable dapat mendefinisikan beberapa metode generik seperti masalah. Karena prosedur penerbitan untuk buku, jurnal, dan CD akan berbeda, metode penerbitan harus diganti di kelas Buku, Jurnal, dan CD. Meskipun ini tidak membantu dalam penggunaan kembali kode, tetapi membantu memiliki implementasi standar dari metode masalah di berbagai kelas konkret. Perhatikan bahwa Issuable adalah kelas abstrak dan tidak dapat dipakai. Di sisi lain, Buku, Jurnal, dan CD adalah kelas konkret dan dapat dipakai untuk membuat objek.



Gambar 7.7 Contoh kelas abstrak.

Bagaimana Mengidentifikasi Hubungan Kelas?

Misalkan kita ingin menulis kode untuk masalah pemrograman sederhana. Bagaimana kita mengidentifikasi kelas dan hubungannya dari deskripsi ini, sehingga kita dapat menulis kode yang diperlukan? Hal ini dapat dilakukan dengan analisis yang cermat terhadap kalimat-kalimat yang diberikan dalam uraian masalah. Kata benda dalam sebuah kalimat sering menunjukkan kelas. Sebaliknya, hubungan antar kelas biasanya ditunjukkan dengan adanya kata kunci tertentu. Berikut ini adalah contoh beberapa kata kunci (ditampilkan dalam huruf miring) yang menunjukkan hubungan khusus antara dua kelas A dan B:

Komposisi

- B adalah bagian tetap dari A
- A terdiri dari Bs
- A adalah himpunan permanen dari Bs

Pengumpulan

- B adalah bagian dari A
- A mengandung B
- A adalah kumpulan dari Bs

Warisan

- A adalah sejenis B
- A adalah spesialisasi dari B
- A berperilaku seperti B

Asosiasi

- A delegasi ke B
- A membutuhkan bantuan dari B
- A berkolaborasi dengan B. Di sini berkolaborasi dengan dapat berupa berbagai macam kolaborasi yang mungkin di antara kelas-kelas seperti mempekerjakan, kredit, mendahului, berhasil, dll.

Konsep Kunci Lainnya

Sekarang adalah beberapa konsep utama lain yang digunakan dalam pendekatan pengembangan program berorientasi objek:

Abstraksi

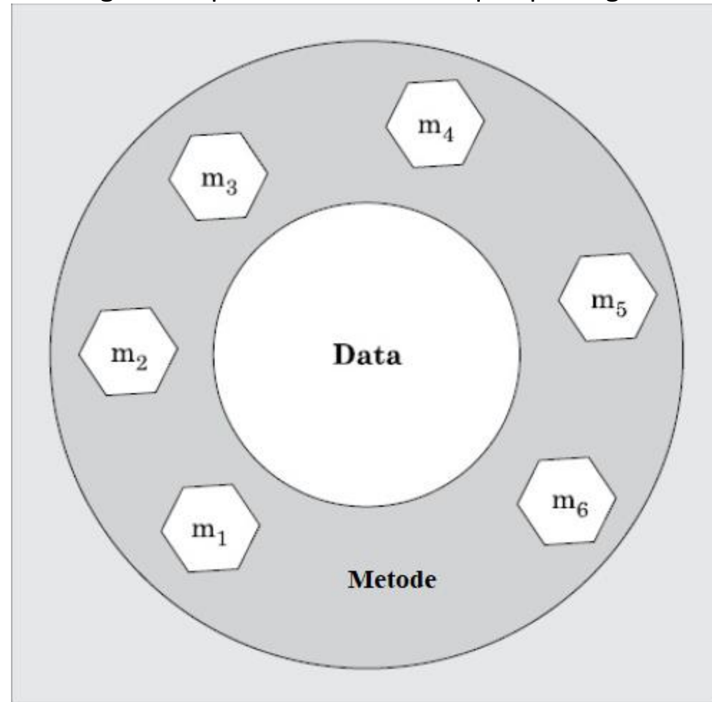
Mari kita rekapitulasi dulu bagaimana mekanisme abstraksi bekerja. Abstraksi adalah pemeriksaan selektif aspek-aspek tertentu dari suatu masalah sementara mengabaikan semua aspek yang tersisa dari suatu masalah. Dengan kata lain, tujuan utama penggunaan mekanisme abstraksi adalah untuk mempertimbangkan hanya aspek-aspek masalah yang relevan dengan tujuan tertentu dan untuk menekan semua aspek masalah yang tidak relevan. Mekanisme abstraksi memungkinkan kita untuk merepresentasikan masalah dengan cara yang lebih sederhana dengan hanya mempertimbangkan aspek-aspek yang relevan dengan tujuan tertentu dan menghilangkan semua detail lain yang tidak relevan.

Banyak abstraksi yang berbeda dari masalah yang sama dapat dibangun tergantung pada tujuan abstraksi dibuat. Mekanisme abstraksi tidak hanya membantu developer untuk memahami dan menghargai masalah dengan lebih baik saat mencari solusi, tetapi juga dapat membantu pemahaman yang lebih baik tentang desain sistem oleh tim pemeliharaan. Abstraksi didukung dalam dua cara berbeda dalam desain berorientasi objek (OOD). Ini adalah sebagai berikut:

Abstraksi fitur: Hirarki kelas dapat dilihat sebagai mendefinisikan beberapa level (hierarki) abstraksi, di mana setiap kelas merupakan abstraksi dari subkelasnya. Artinya, setiap kelas adalah representasi (abstrak) yang disederhanakan dari kelas turunannya dan hanya mempertahankan fitur-fitur yang umum untuk semua kelas turunannya dan mengabaikan fitur-fitur lainnya. Dengan demikian, mekanisme pewarisan dapat dianggap menyediakan abstraksi fitur.

Abstraksi data: Objek itu sendiri dapat dianggap sebagai entitas abstraksi data, karena ia mengabstraksikan cara yang tepat di mana ia menyimpan berbagai item data pribadinya dan hanya menyediakan seperangkat metode ke objek lain untuk mengakses dan memanipulasi item data ini. Dengan kata lain, kita dapat mengatakan bahwa abstraksi data menyiratkan bahwa setiap objek bersembunyi (abstrak menjauh) dari objek lain dengan cara yang tepat di mana ia menyimpan informasi internalnya. Ini membantu dalam mengembangkan program berkualitas baik, karena menyebabkan objek memiliki kopling rendah satu sama lain, karena mereka tidak secara langsung mengakses data apa pun yang dimiliki satu sama lain. Setiap objek hanya menyediakan satu set metode, yang dapat digunakan objek lain untuk mengakses dan memanipulasi informasi pribadi objek ini. Misalnya, objek tumpukan mungkin menyimpan data internalnya baik dalam bentuk larik nilai atau dalam bentuk daftar tertaut. Objek lain tidak akan tahu bagaimana tepatnya objek ini menyimpan datanya (yaitu data diabstraksi) dan bagaimana ia memanipulasi datanya secara internal. Apa yang akan mereka ketahui adalah kumpulan metode seperti push, pop, dan top-of-stack yang diberikannya ke objek lain untuk mengakses dan memanipulasi data.

Keuntungan penting dari prinsip abstraksi data adalah mengurangi kopling di antara berbagai objek, Oleh karena itu, ini mengarah pada pengurangan kompleksitas keseluruhan desain, dan membantu dalam perawatan yang mudah dan penggunaan kembali kode. Abstraksi adalah mekanisme yang kuat untuk mengurangi kompleksitas yang dirasakan dari desain perangkat lunak. Analisis data yang dikumpulkan dari beberapa proyek pengembangan perangkat lunak menunjukkan bahwa produktivitas perangkat lunak berbanding terbalik dengan kompleksitas perangkat lunak yang dirasakan. Oleh karena itu, penggunaan abstraksi secara implisit, seperti yang terjadi dalam pengembangan berorientasi objek, adalah cara yang menjanjikan untuk meningkatkan produktivitas developer perangkat lunak.



Gambar 7.8 Representasi skematik dari konsep enkapsulasi.

Enkapsulasi

Data suatu objek dienkapsulasi dalam metodenya. Untuk mengakses data internal ke suatu objek, objek lain harus memanggil metodenya, dan tidak dapat mengakses data secara langsung. Konsep ini secara skematis ditunjukkan pada Gambar 7.8. Amati dari Gambar 7.8 bahwa tidak ada cara bagi suatu objek untuk mengakses data pribadi ke objek lain, selain dengan memanggil metodenya. Enkapsulasi menawarkan tiga keuntungan penting berikut:

Perlindungan dari akses data yang tidak sah: Fitur enkapsulasi melindungi variabel objek agar tidak dirusak oleh objek lain secara tidak sengaja. Perlindungan ini mencakup perlindungan dari akses yang tidak sah dan juga perlindungan dari masalah yang timbul dari akses bersamaan ke data seperti kebuntuan dan nilai yang tidak konsisten.

Penyembunyian data: Enkapsulasi menyiratkan bahwa data struktur internal suatu objek disembunyikan, sehingga semua interaksi dengan objek menjadi sederhana dan terstandarisasi. Ini memfasilitasi penggunaan kembali kelas di berbagai proyek. Selanjutnya, jika data internal atau badan metode dari suatu kelas dimodifikasi, kelas lain tidak akan terpengaruh. Ini mengarah pada perawatan dan koreksi bug yang lebih mudah.

Kopling lemah: Karena objek tidak secara langsung mengubah data internal satu sama lain, mereka digabungkan dengan lemah. Kopling yang lemah di antara objek meningkatkan pemahaman desain karena setiap objek dapat dipelajari dan dipahami secara terpisah dari objek lain.

Polimorfisme

Polimorfisme secara harfiah berarti poli (banyak) morfisme (bentuk). Ingatlah bahwa dalam Kimia, intan, grafit, dan batu bara disebut bentuk karbon polimorfik. Artinya, meskipun berlian, batu bara, dan grafit pada dasarnya adalah karbon, mereka berperilaku sangat berbeda. Dalam cara yang analog dalam paradigma berorientasi objek, polimorfisme menunjukkan bahwa suatu objek dapat merespon (berperilaku) sangat berbeda bahkan ketika operasi yang sama dipanggil di atasnya tergantung pada objek polimorfik yang tepat yang panggilannya akan terikat. Ada dua jenis utama polimorfisme dalam orientasi objek:

Polimorfisme statis: Polimorfisme statis terjadi ketika beberapa metode menerapkan operasi yang sama. Dalam polimorfisme jenis ini, ketika suatu metode dipanggil (nama metode yang sama tetapi jenis parameter yang berbeda), perilaku (tindakan) yang berbeda akan diamati. Jenis polimorfisme ini juga disebut sebagai pengikatan statis, karena metode yang tepat untuk diikat pada pemanggilan metode ditentukan pada waktu kompilasi (statis). Mari kita coba memahami pengikatan statis melalui contoh berikut. Misalkan sebuah kelas bernama Circle memiliki tiga definisi untuk operasi create—int create(), int create(int radius), dan int create(float x, float y, int radius). Ingat bahwa ketika beberapa metode mengimplementasikan operasi yang sama, maka mekanisme yang digunakan disebut metode overloading. (Perhatikan perbedaan antara operasi dan metode.) Ketika operasi yang sama (misalnya buat) diimplementasikan oleh beberapa metode, nama metode dikatakan kelebihan beban. Satu definisi dari operasi create tidak mengambil argumen apa pun (create()) dan membuat lingkaran dengan parameter default. Definisi kedua mengambil titik pusat dan jari-jari lingkaran sebagai parameternya (create(float x, float y, float radius). Asumsikan bahwa dalam kedua kasus di atas, gaya isian akan disetel ke nilai default “no fill". Definisi ketiga dari operasi create mengambil titik tengah, radius, dan gaya isian sebagai inputnya. Ketika metode create dipanggil, tergantung pada parameter yang diberikan dalam pemanggilan, metode pencocokan dapat dengan mudah ditentukan selama kompilasi dengan memeriksa daftar parameter panggilan dan panggilan akan terikat secara statis. Jika metode create dipanggil tanpa parameter, maka lingkaran default akan dibuat. Jika hanya pusat dan radius yang disediakan, maka lingkaran yang sesuai akan dibuat. dibuat tanpa tipe isian, dan seterusnya. Definisi kelas Circle dengan metode create yang kelebihan beban ditunjukkan pada Gambar 7. 9.

Polimorfisme dinamis: Polimorfisme dinamis juga disebut pengikatan dinamis. Dalam pengikatan dinamis, metode yang tepat yang akan dipanggil (terikat) pada pemanggilan metode hanya dapat diketahui pada saat run time (secara dinamis) dan tidak dapat ditentukan pada waktu kompilasi. Artinya, perilaku pasti yang akan dihasilkan pada pemanggilan metode tidak dapat diprediksi pada waktu kompilasi dan hanya dapat diamati pada waktu proses.

```
class Circle{
        private float x, y, radius;
        private int fillType;

        public create();
        public create(float x, float y, float radius);
        public create(float x, float y, float radius, int fillType);
    }
```

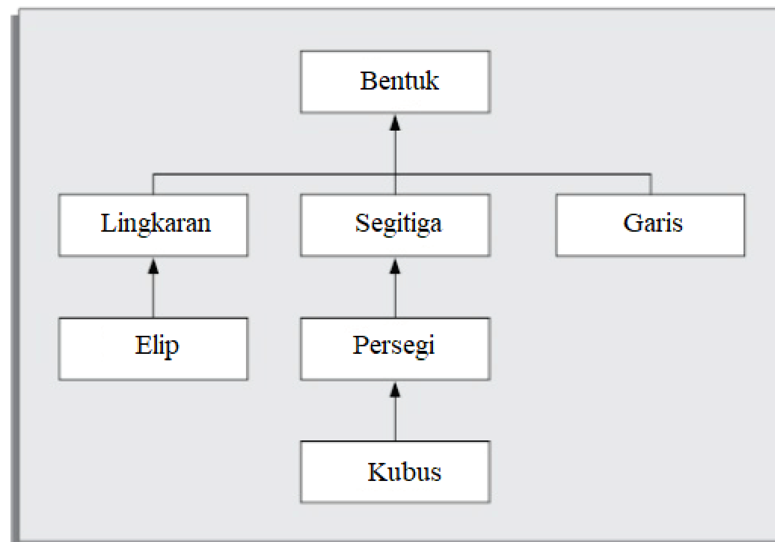
Gambar 7.9 Kelas lingkaran dengan metode create yang kelebihan beban.

Sekarang mari kita jelaskan bagaimana pengikatan dinamis bekerja dalam program berorientasi objek. Pengikatan dinamis didasarkan pada dua konsep penting:

- Penugasan suatu objek ke objek lain yang kompatibel.
- Penggantian metode dalam hierarki kelas.

Penugasan ke objek yang kompatibel

Dalam orientasi objek, objek dari kelas turunan kompatibel dengan objek dari kelas dasar. Artinya, objek dari kelas turunan dapat ditugaskan ke objek kelas dasar, tetapi tidak sebaliknya. Juga sebuah objek tidak dapat ditugaskan ke objek dari kelas saudara atau objek dari kelas yang sama sekali tidak terkait karena alasan yang jelas. Ini adalah prinsip penting dalam orientasi objek dan dikenal sebagai prinsip Substitusi Liskov. Untuk memahami prinsip ini, ingatlah bahwa kelas turunan mendefinisikan beberapa atribut tambahan dan penugasan objek kelas dasar ke objek kelas turunan dapat membuat atribut tersebut tidak terdefinisi.



Gambar 7.10 Hirarki kelas objek geometris.

Penggantian metode

Kita sudah mengetahui tentang prinsip overriding metode di mana kelas turunan memberikan definisi baru ke metode kelas dasar. Sekarang mari kita memahami cara kerja pengikatan dinamis dengan memanfaatkan dua mekanisme di atas. Misalkan kita telah mendefinisikan hierarki kelas dari bentuk geometris yang berbeda untuk paket gambar grafis seperti yang ditunjukkan pada Gambar 7.10. Seperti dapat dilihat dari gambar, Shape adalah kelas abstrak dan kelas Circle, Rectangle, dan Line diturunkan langsung darinya dan selanjutnya kelas Ellipse, Square dan Cube telah diturunkan untuk membentuk hierarki pewarisan. Sekarang, anggaplah metode draw dideklarasikan di kelas Shape dan diganti di setiap kelas turunan. Selanjutnya, misalkan satu set berbagai jenis objek Shape telah dibuat satu per satu. Dengan prinsip substitusi Liskov, objek Shape yang dibuat dapat disimpan dalam array tipe Shape. Jika berbagai jenis objek geometris yang menyusun gambar disimpan dalam larik bertipe Bentuk, maka panggilan ke metode menggambar untuk setiap objek akan berhati-hati untuk menampilkan elemen gambar yang sesuai. Artinya, panggilan menggambar yang sama ke objek Bentuk akan menangani menggambar objek gambar yang sesuai. Perhatikan bahwa karena pengikatan dinamis, panggilan ke metode draw dari kelas bentuk akan menangani tampilan objek gambar yang sesuai yang berada di larik bentuk. Hal ini diilustrasikan dalam segmen kode yang ditunjukkan pada Gambar 7.11.

```
Traditional code
Shape s[100];

for(i=0;i<100;i++){
    if(s[i]==Circle) then
            draw_circle();
    else if(shape==Rectangle) then
            draw_rectangle();
            —
            —
            —
}
```

```
Object–oriented code
Shape s[100];

for(i=0;i<100;i++)
        shape.draw();
```

Gambar 7.11 Kode tradisional versus kode berorientasi objek yang menggabungkan fitur pengikatan dinamis.

Keuntungan polimorfisme dengan membandingkan segmen kode program berorientasi objek dan program tradisional untuk menggambar berbagai objek grafik di layar dapat dianalisis (ditunjukkan pada Gambar 7.11). Menggunakan pengikatan dinamis, seorang programmer dapat memanggil metode generik suatu objek dan meninggalkan cara yang tepat di mana pesan ini akan ditangani akan diputuskan' tergantung pada objek yang saat ini ditugaskan ke objek penerima. Dengan pengikatan dinamis, objek turunan baru dapat ditambahkan dengan sedikit perubahan pada program. Keuntungan utama dari pengikatan dinamis adalah bahwa hal itu mengarah pada pemrograman yang elegan dan memfasilitasi penggunaan kembali dan pemeliharaan kode.

Dapat dilihat Gambar 7.11 bahwa penggunaan pengikatan dinamis, kode berorientasi objek jauh lebih ringkas, dapat dimengerti, dan menarik secara intelektual dibandingkan dengan kode prosedural yang setara. Selanjutnya, misalkan dalam segmen program contoh, kemudian ditemukan perlu untuk menangani gambar grafik primitif baru, katakanlah elips. Kemudian, kode prosedur harus diubah dengan menambahkan klausa if-then-else yang baru. Namun, dalam kasus program berorientasi objek, kode tidak perlu diubah, hanya kelas baru yang disebut Ellipse yang harus diturunkan dalam hierarki Bentuk. Sekarang kita dapat meringkas mekanisme pengikatan dinamis sebagai berikut: Bahkan ketika metode dari suatu objek dari kelas dasar dipanggil, metode yang ditimpa yang sesuai dari kelas turunan akan dipanggil tergantung pada objek yang tepat yang mungkin telah ditetapkan saat dijalankan. -waktu ke objek kelas dasar.

Genericity

Genericity adalah kemampuan untuk membuat parameter definisi kelas. Misalnya, saat mendefinisikan tumpukan kelas dari berbagai jenis elemen seperti tumpukan integer, tumpukan karakter, tumpukan titik-mengambang, dll.; generikitas memungkinkan kita untuk mendefinisikan kelas generik dari tipe stack dan kemudian membuat instance-nya baik sebagai integer stack, character stack, atau floating-point stack sesuai kebutuhan. Ini dapat dicapai dengan menetapkan nilai yang sesuai untuk parameter yang digunakan dalam definisi kelas generik.

Ketentuan Teknis Terkait

Berikut ini akan dibahas beberapa istilah yang berkaitan dengan orientasi objek sebagai berikut:

Presistensi

Objek biasanya dihancurkan setelah program menyelesaikan eksekusinya. Objek persisten disimpan secara permanen. Artinya, mereka hidup di eksekusi yang berbeda. Sebuah objek dapat dibuat persisten dengan mempertahankan salinan objek dalam penyimpanan sekunder atau dalam database.

Agen

Objek pasif adalah objek yang melakukan beberapa tindakan hanya ketika diminta melalui pemanggilan beberapa metodenya. Agen (juga disebut objek aktif), di sisi lain, memantau peristiwa yang terjadi dalam aplikasi dan mengambil tindakan secara mandiri. Agen digunakan dalam aplikasi seperti pengecualian pemantauan. Misalnya, dalam aplikasi basis data seperti akuntansi, agen dapat memantau neraca dan akan memberi tahu pengguna setiap kali ketidakkonsistenan muncul dalam neraca karena beberapa transaksi yang tidak tepat terjadi.

Widget

Istilah widget adalah singkatan dari objek jendela. Widget adalah objek primitif yang digunakan untuk desain antarmuka pengguna grafis (GUI). Primitif desain antarmuka pengguna grafis (widget) yang lebih kompleks dapat diturunkan dari widget dasar menggunakan mekanisme pewarisan. Widget memelihara data internal seperti geometri jendela, warna latar belakang dan warna dasar depan jendela, bentuk dan ukuran kursor, dll. Metode yang didukung oleh widget memanipulasi data yang disimpan dan melakukan operasi seperti mengubah ukuran jendela, membuat ikon window, destroy window, dll. Widget menjadi komponen standar desain GUI. Hal ini telah memunculkan teknik pengembangan antarmuka pengguna berbasis komponen. Kita akan membahas lebih lanjut tentang widget dan pengembangan antarmuka pengguna berbasis komponen di Bab 9 di mana kita membahas desain GUI.

Kelebihan dan Kekurangan OOD

Seperti halnya teknik lainnya, OOD memiliki kelebihan dan kekurangannya sendiri. Berikut adalah ulasannya.

Kelebihan OOD

Dalam beberapa dekade terakhir sejak OOD muncul, ia telah diterima secara luas di industri maupun di kalangan akademis. Alasan utama popularitas OOD adalah karena ia memenuhi janji-janji berikut:

- Penggunaan kembali kode dan desain
- Peningkatan produktivitas
- Kemudahan pengujian dan pemeliharaan
- Kode dan pemahaman desain yang lebih baik memungkinkan pengembangan program besar

Dari semua keuntungan yang disebutkan di atas, biasanya disepakati bahwa keuntungan utama OOD adalah peningkatan produktivitas—yang terjadi karena berbagai faktor, seperti berikut ini:

- Penggunaan kembali kode dengan menggunakan perpustakaan kelas yang telah dikembangkan sebelumnya
- Penggunaan kembali kode karena warisan
- Abstraksi yang lebih sederhana dan lebih intuitif, yaitu, manajemen masalah bawaan dan kompleksitas kode yang lebih baik
- Dekomposisi masalah yang lebih baik

Beberapa hasil penelitian menunjukkan bahwa ketika perusahaan mulai mengembangkan perangkat lunak menggunakan paradigma berorientasi objek, beberapa proyek pertama dikenakan biaya lebih tinggi daripada proyek yang dikembangkan secara

tradisional. Ini mungkin karena terbiasa dengan teknik baru dan membangun perpustakaan kelas yang dapat digunakan kembali dalam proyek-proyek berikutnya. Setelah menyelesaikan beberapa proyek, penghematan biaya menjadi mungkin. Menurut laporan pengalaman, lingkungan pengembangan berorientasi objek yang mapan dapat membantu mengurangi biaya pengembangan sebanyak 20 persen hingga 50 persen dibandingkan lingkungan pengembangan tradisional.

Kekurangan OOD

Berikut ini adalah beberapa kelemahan menonjol yang melekat pada paradigma objek:

- Prinsip-prinsip abstraksi, penyembunyian data, pewarisan, dll. memang menimbulkan overhead waktu berjalan karena kode tambahan yang dihasilkan karena fitur-fitur ini. Ini menyebabkan program berorientasi proyek berjalan sedikit lebih lambat daripada program prosedural yang setara.
- Konsekuensi penting dari orientasi objek adalah bahwa data yang terpusat dalam implementasi prosedural, tersebar di berbagai objek dalam implementasi berorientasi objek. Oleh karena itu, lokalitas spasial data menjadi lemah dan ini menyebabkan rasio kehilangan cache yang lebih tinggi dan akibatnya waktu akses memori yang lebih besar. Ini akhirnya muncul sebagai peningkatan waktu menjalankan program.

Seperti yang dapat kita lihat, peningkatan waktu berjalan adalah kelemahan utama dari orientasi objek dan produktivitas yang lebih tinggi adalah keuntungan utama. Di masa sekarang, komputer telah menjadi sangat cepat, dan overhead run time yang kecil tidak menjadi masalah sama sekali. Akibatnya, kelebihan OOD menutupi kekurangannya.

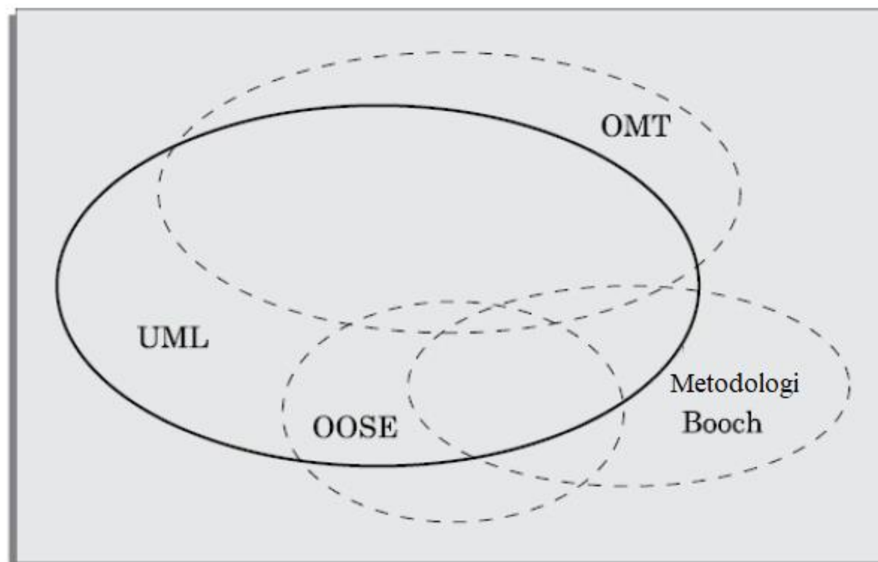
7.2 *UNIFIED MODELLING LANGUAGE* (UML)

Sesuai dengan namanya, UML adalah bahasa untuk mendokumentasikan model. Seperti halnya bahasa lain, UML memiliki sintaks (seperangkat simbol dasar dan aturan pembentukan kalimat) dan semantik (makna simbol dan kalimat dasar). Ini menyediakan satu set notasi grafis dasar (misalnya persegi panjang, garis, elips, dll) yang dapat dikombinasikan dengan cara tertentu untuk mendokumentasikan hasil desain dan analisis. Penting untuk diingat bahwa UML bukanlah desain sistem atau metodologi pengembangan dengan sendirinya, juga tidak terikat pada metodologi tertentu. UML hanyalah bahasa untuk mendokumentasikan model. Sebelum munculnya UML, setiap metodologi desain tidak hanya menentukan langkah-langkah desain yang sama sekali berbeda, tetapi masing-masing terikat pada beberapa bahasa pemodelan desain tertentu. Misalnya, metodologi OMT memiliki metodologi desainnya sendiri dan memiliki rangkaian notasi yang unik. Begitu pula dengan metodologi Booch, dan seterusnya. Situasi ini menyulitkan seseorang yang akrab dengan satu metodologi untuk memahami solusi desain yang dikembangkan dan didokumentasikan menggunakan metodologi lain. Secara umum, penggunaan kembali solusi desain di berbagai metodologi sulit dilakukan. UML dimaksudkan untuk mengatasi masalah yang melekat pada teknik pemodelan yang ada. UML dapat digunakan untuk mendokumentasikan hasil analisis dan desain berorientasi objek yang telah diperoleh dengan menggunakan metodologi apapun.

Salah satu tujuan developer UML adalah menjaga agar notasi UML tidak bergantung pada metodologi desain tertentu, sehingga dapat digunakan bersama dengan metodologi desain tertentu. Dalam hal ini, UML berbeda dari pendahulunya (misalnya, OMT, metodologi Booch, dll.) di mana notasi yang didukung oleh bahasa pemodelan terkait erat dengan metodologi desain yang sesuai.

Asal UML

Pada akhir tahun delapan puluhan dan awal tahun sembilan puluhan, ada perkembangan teknik dan notasi desain berorientasi objek. Banyak dari ini telah menjadi sangat populer dan digunakan secara luas. Namun, notasi yang mereka gunakan dan paradigma desain khusus yang mereka anjurkan, berbeda satu sama lain dalam banyak hal. Dengan begitu banyak teknik populer untuk dipilih, tidak jarang menemukan tim proyek yang berbeda dalam organisasi yang sama menggunakan metodologi yang berbeda dan mendokumentasikan analisis berorientasi objek dan hasil desain mereka menggunakan notasi yang berbeda. Notasi beragam yang digunakan untuk mendokumentasikan solusi desain ini menimbulkan banyak kebingungan di antara anggota tim dan membuatnya sangat sulit untuk menggunakan kembali desain di seluruh proyek dan mengkomunikasikan ide di seluruh tim proyek.



Gambar 7.12 Representasi skematis dari dampak teknik pemodelan objek yang berbeda pada UML.

UML dikembangkan untuk menstandarisasi sejumlah besar notasi pemodelan berorientasi objek yang ada di awal tahun sembilan puluhan. Yang utama yang digunakan saat itu adalah sebagai berikut:

- OMT [Rumbaugh 1991]
- Metodologi Booch [Booch 1991]
- OOSE [Jacobson 1992]
- Metodologi Odell [Odell 1992]
- Metodologi Shlaer dan Mellor[Shlaer 1992]

Tak perlu dikatakan bahwa UML telah meminjam banyak konsep dari teknik pemodelan ini. Konsep dan notasi terutama dari tiga metodologi pertama telah banyak digunakan. Pengaruh berbagai teknik pemodelan objek pada UML ditunjukkan secara skematis pada Gambar 7.12. Seperti yang ditunjukkan pada Gambar 7.12, OMT memiliki pengaruh paling besar pada UML.

UML diadopsi oleh object management group (OMG) sebagai standar de facto pada tahun 1997. Sebenarnya, OMG bukanlah badan perumus standar, tetapi merupakan asosiasi industri yang mencoba memfasilitasi perumusan awal standar. OMG bertujuan untuk mempromosikan notasi dan teknik konsensus dengan harapan jika penggunaannya meluas, maka secara otomatis akan menjadi standar. Untuk informasi lebih lanjut tentang OMG, lihat www.omg.org. Dengan meluasnya penggunaan UML, ISO mengadopsi standar UML (ISO

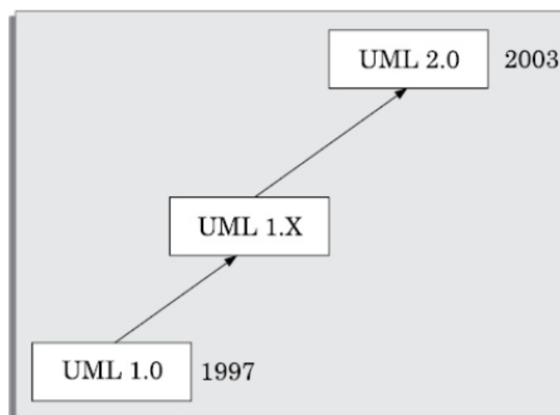
19805) pada tahun 2005, dan dengan UML ini telah menjadi standar resmi; ini semakin meningkatkan penggunaan UML. UML lebih kompleks daripada pendahulunya. Ini wajar dan diharapkan karena dimaksudkan agar lebih komprehensif dan dapat diterapkan pada gamut masalah yang lebih luas daripada notasi apa pun yang ada sebelum UML.

UML berisi kumpulan notasi ekstensif untuk membantu mendokumentasikan beberapa aspek (tampilan) dari solusi desain melalui banyak jenis diagram. UML telah berhasil digunakan untuk memodelkan masalah besar dan kecil. Keanggunan UML, adopsinya oleh OMG, dan selanjutnya oleh ISO serta dukungan industri yang kuat telah membantu UML mendapatkan penerimaan yang tersebar luas. UML sekarang digunakan di lembaga akademis dan penelitian serta di sejumlah besar proyek pengembangan perangkat lunak di seluruh dunia. Sangat menarik untuk dicatat bahwa penggunaan UML tidak terbatas pada industri perangkat lunak saja. Sebagai contoh penggunaan UML di luar masalah pengembangan perangkat lunak, beberapa mobil

pabrikan berencana untuk menggunakan UML untuk inisiatif "build-to-order" mereka. Banyak notasi UML yang sulit digambar dengan tangan di atas kertas dan paling baik digambar menggunakan alat CASE seperti Rational Rose© (lihat www.rational.com) atau MagicDraw (www.magicdraw.com). Sekarang beberapa alat UML CASE gratis juga tersedia di web. Sebagian besar alat CASE yang tersedia membantu menyempurnakan model objek awal ke desain akhir, dan ini juga secara otomatis menghasilkan templat kode dalam berbagai bahasa, setelah model UML dibuat.

Evolusi UML

Sejak rilis UML 1.0 pada tahun 1997, UML terus berkembang (lihat Gambar 7.13) dengan umpan balik dari praktisi dan akademisi untuk membuatnya berlaku untuk situasi pengembangan sistem yang berbeda. Hampir setiap tahun beberapa rilis baru (ditunjukkan sebagai UML 1.X pada Gambar 7.13) diumumkan. Tonggak utama dalam evolusi UML adalah rilis UML 2.0 pada tahun 2007. Karena penggunaan aplikasi tertanam meningkat pesat, ada permintaan populer untuk memperluas UML untuk mendukung konsep dan notasi khusus yang diperlukan untuk mengembangkan aplikasi tertanam. UML 2.0 adalah upaya untuk membuat UML dapat diterapkan pada pengembangan sistem konkuren dan embedded. Untuk ini, banyak fitur baru seperti acara, port, dan bingkai dalam diagram urutan diperkenalkan.



Gambar 7.13 Evolusi UML.

Apa itu model?

Sebelum kita membahas fitur-fitur UML secara mendetail, penting untuk memahami apa sebenarnya yang dimaksud dengan model, dan mengapa perlu membuat model. Model adalah abstraksi dari masalah (atau situasi) nyata, dan dibangun dengan meninggalkan detail

yang tidak perlu. Ini mengurangi kompleksitas masalah dan membuatnya mudah untuk memahami masalah (atau situasi).

Model adalah versi sederhana dari sistem nyata. Adalah berguna untuk menganggap model sebagai menangkap aspek penting untuk beberapa aplikasi sementara menghilangkan (atau mengabstraksikan) sisanya. Ketika ukuran masalah meningkat, kompleksitas yang dirasakan meningkat secara eksponensial karena keterbatasan kognitif manusia. Oleh karena itu, untuk mengembangkan pemahaman yang baik tentang masalah apa pun, perlu untuk membangun model masalah. Pemodelan ternyata menjadi alat yang sangat penting dalam desain perangkat lunak dan membantu menangani kompleksitas masalah secara efektif. Model-model ini yang pertama dibangun adalah model masalah. Metodologi desain pada dasarnya mengubah model analisis ini menjadi model desain melalui penyempurnaan berulang.

Berbagai jenis model diperoleh berdasarkan aspek spesifik dari sistem aktual yang diabaikan saat membangun model. Untuk memahami hal ini, mari kita perhatikan model-model yang dibangun oleh seorang arsitek dari sebuah bangunan besar. Saat membangun tampak depan sebuah bangunan besar (rencana elevasi), arsitek mengabaikan aspek-aspek seperti denah lantai, kekuatan dinding, detail arsitektur bagian dalam, dll. Saat membangun denah lantai, ia sama sekali mengabaikan tampilan depan (ketinggian). denah), denah lokasi, karakteristik termal dan pencahayaan, dll. dari bangunan.

Sebuah model dalam konteks pengembangan perangkat lunak dapat berupa grafis, tekstual, matematika, atau berbasis kode program. Model grafis sangat populer karena mudah dipahami dan dibangun. UML pada dasarnya adalah alat pemodelan grafis. Namun, ada situasi pemodelan tertentu (dibahas nanti dalam Bab ini), di mana selain model UML grafis, penjelasan tekstual terpisah diperlukan untuk menyertai model grafis.

Mengapa membangun model?

Alasan penting di balik pembuatan model adalah membantu mengelola kompleksitas masalah dan memfasilitasi penyelesaian yang baik dan pada saat yang sama membantu mengurangi biaya desain. Model awal dari suatu masalah disebut model analisis. Model analisis suatu masalah dapat disempurnakan menjadi model desain dengan menggunakan metodologi desain. Setelah model sistem telah dibangun, ini dapat digunakan untuk berbagai tujuan selama pengembangan perangkat lunak, termasuk yang berikut:

- Analisis
- Spesifikasi
- Rancangan
- Pengkodean
- Visualisasi dan pemahaman dari sebuah implementasi.
- Pengujian, dll.

Karena model dapat digunakan untuk berbagai tujuan, masuk akal untuk mengharapkan bahwa model akan bervariasi secara detail tergantung pada tujuan pembuatannya. Misalnya, model yang dikembangkan untuk analisis dan spesifikasi awal harus sangat berbeda dari model yang digunakan untuk desain. Sebuah model yang dibangun untuk analisis dan spesifikasi tidak akan menunjukkan keputusan desain yang akan dibuat nanti selama tahap desain. Di sisi lain, model yang dibangun untuk tujuan desain harus menangkap semua keputusan desain. Oleh karena itu, adalah ide yang baik untuk secara eksplisit menyebutkan tujuan dari model yang telah dikembangkan.

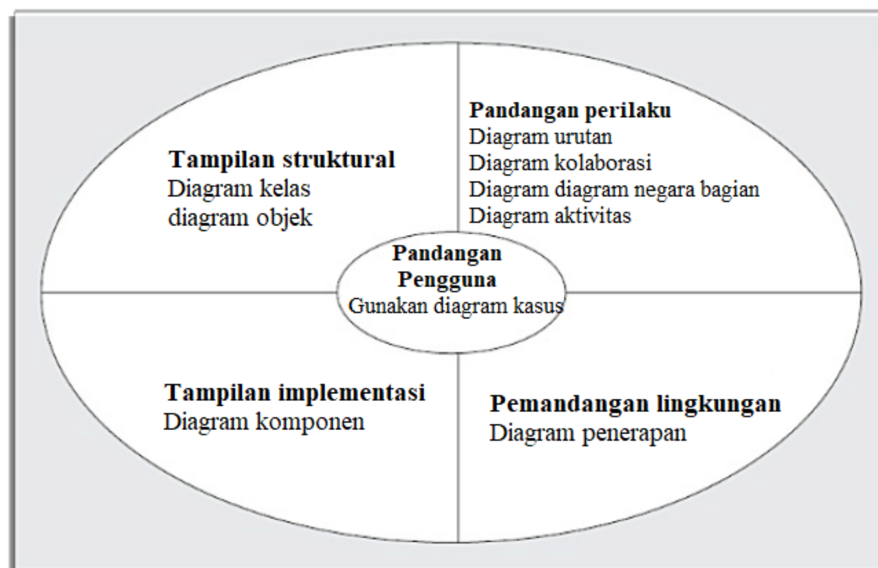
7.3 DIAGRAM UML

Pada bagian ini, kita membahas diagram yang didukung oleh UML 1.0. Kemudian di bagian selanjutnya kita akan membahas perubahan pada UML 1.0 yang dibawa oleh UML 2.0. UML 1.0 dapat digunakan untuk membangun sembilan jenis diagram yang berbeda untuk menangkap lima pandangan yang berbeda dari suatu sistem. Sama seperti sebuah bangunan dapat dimodelkan dari beberapa pandangan (atau perspektif) seperti perspektif ventilasi, perspektif listrik, perspektif pencahayaan, perspektif pemanasan, dll; diagram UML yang berbeda memberikan perspektif yang berbeda dari sistem perangkat lunak yang akan dikembangkan dan memfasilitasi pemahaman yang komprehensif dari sistem. Setiap perspektif berfokus pada beberapa aspek tertentu dan mengabaikan sisanya. Beberapa orang mungkin bertanya, mengapa membangun beberapa model dari perspektif yang berbeda—mengapa tidak membangun satu model yang mencakup semua perspektif? Jawabannya adalah sebagai berikut: Jika satu model dibuat untuk menangkap semua perspektif yang diperlukan, maka itu akan serumit masalah aslinya, dan tidak akan banyak berguna.

Setelah sistem dimodelkan dari semua perspektif yang diperlukan, model yang dibangun dapat disempurnakan untuk mendapatkan implementasi sistem yang sebenarnya. Diagram UML dapat menangkap tampilan (model) sistem berikut:

- Pandangan pengguna
- Tampilan struktural
- Pandangan perilaku
- Tampilan implementasi
- Pemandangan lingkungan

Gambar 7.14 menunjukkan pandangan berbeda yang dapat didokumentasikan oleh diagram UML. Perhatikan bahwa tampilan pengguna ditampilkan sebagai tampilan utama. Ini karena berdasarkan tampilan pengguna, semua tampilan lain dikembangkan dan semua tampilan harus sesuai dengan tampilan pengguna. Sebagian besar metodologi analisis dan desain berorientasi objek, termasuk yang akan kita bahas di Bab 8 mengharuskan kita untuk beralih di antara pandangan yang berbeda beberapa kali untuk sampai pada desain akhir. Pertama-tama memberikan gambaran singkat tentang pandangan yang berbeda dari suatu sistem yang dapat didokumentasikan menggunakan UML.



Gambar 7.14 Berbagai jenis diagram dan tampilan yang didukung di UML.

Pandangan pengguna

Tampilan ini mendefinisikan fungsionalitas yang disediakan oleh sistem untuk penggunanya. Tampilan pengguna menangkap pandangan sistem dalam hal fungsionalitas yang ditawarkan oleh sistem kepada penggunanya. Tampilan pengguna adalah tampilan kotak hitam dari sistem di mana struktur internal, perilaku dinamis dari komponen sistem yang berbeda, implementasi, dll. tidak ditangkap. Pandangan pengguna sangat berbeda dari semua pandangan lain dalam arti bahwa itu adalah model fungsional1 dibandingkan dengan semua pandangan lain yang pada dasarnya adalah model objek.2

Tampilan pengguna dapat dianggap sebagai tampilan utama dan semua tampilan lain diperlukan untuk menyesuaikan dengan tampilan ini. Pemikiran ini sebenarnya adalah inti dari setiap gaya pengembangan yang berpusat pada pengguna. Sungguh luar biasa bahwa bahkan untuk pengembangan berorientasi objek, kita membutuhkan tampilan fungsional. Itu karena, bagaimanapun, pengguna menganggap sistem sebagai menyediakan seperangkat fungsi.

Tampilan struktural

Pandangan struktural mendefinisikan struktur masalah (atau solusi) dalam hal jenis objek (kelas) yang penting untuk memahami kerja sistem dan implementasinya. Ini juga menangkap hubungan antar kelas (objek). Model struktural juga disebut model statis, karena struktur suatu sistem tidak berubah seiring waktu.

Pandangan perilaku

Tampilan perilaku menangkap bagaimana objek berinteraksi satu sama lain pada waktunya untuk mewujudkan perilaku sistem. Perilaku sistem menangkap perilaku bergantung waktu (dinamis) dari sistem. Oleh karena itu merupakan model dinamis dari sistem.

Tampilan implementasi

Pandangan ini menangkap komponen penting dari sistem dan saling ketergantungannya. Misalnya, tampilan implementasi mungkin menunjukkan bagian GUI, middleware, dan bagian database sebagai bagian yang berbeda dan juga akan menangkap saling ketergantungannya.

Pemandangan lingkungan

Tampilan ini memodelkan bagaimana komponen yang berbeda diimplementasikan pada perangkat keras yang berbeda. Untuk setiap masalah yang diberikan, haruskah seseorang membangun semua tampilan menggunakan semua diagram yang disediakan oleh UML? Jawabannya Tidak. Untuk sistem yang sederhana, model use case, diagram kelas, dan salah satu diagram interaksi mungkin sudah cukup. Untuk sistem di mana objek mengalami banyak perubahan keadaan, diagram diagram keadaan mungkin diperlukan. Untuk sistem, yang diimplementasikan pada sejumlah besar komponen perangkat keras, diagram penyebaran mungkin diperlukan. Jadi, jenis model yang akan dibangun tergantung pada masalah yang dihadapi. Rosenberg memberikan analogi [Ros 2000] yang mengatakan bahwa "Sama seperti Anda tidak menggunakan semua kata yang tercantum dalam kamus saat menulis prosa, Anda tidak menggunakan semua diagram UML dan elemen pemodelan saat membuat model sistem."

7.4 GUNAKAN MODEL KASUS

Model use case untuk sistem apapun terdiri dari sekumpulan use case. Secara intuitif, use case mewakili berbagai cara di mana suatu sistem dapat digunakan oleh pengguna. Cara sederhana untuk menemukan semua kasus penggunaan suatu sistem adalah dengan mengajukan pertanyaan — "Apa yang dapat dilakukan oleh semua kategori pengguna yang berbeda dengan menggunakan sistem?" Jadi, untuk sistem informasi perpustakaan (LIS), use case dapat berupa:

- buku edisi
- buku kueri
- buku kembali
- buat-anggota
- buku tambahan, dll.

Secara kasar, use case sesuai dengan persyaratan fungsional tingkat tinggi yang kita bahas di Bab 4. Kita juga dapat mengatakan bahwa use case mempartisi perilaku sistem menjadi transaksi, sehingga setiap transaksi melakukan beberapa tindakan yang berguna dari sudut pandang pengguna. Setiap transaksi, untuk diselesaikan, mungkin melibatkan beberapa pertukaran pesan antara pengguna dan sistem. Tujuan dari use case adalah untuk mendefinisikan bagian dari perilaku yang koheren tanpa mengungkapkan struktur internal sistem. Kasus penggunaan tidak menyebutkan algoritma spesifik apa pun yang akan digunakan atau representasi data internal, struktur internal perangkat lunak. Sebuah use case biasanya melibatkan urutan interaksi antara pengguna dan sistem.

Bahkan untuk kasus penggunaan yang sama, bisa ada beberapa urutan interaksi yang berbeda. Sebuah use case terdiri dari satu urutan jalur utama dan beberapa urutan alternatif. Urutan jalur utama mewakili interaksi antara pengguna dan sistem yang biasanya terjadi. Urutan arus utama adalah urutan interaksi yang paling sering terjadi. Misalnya, dalam urutan utama dari kasus penggunaan penarikan tunai yang didukung oleh ATM bank adalah— pengguna memasukkan kartu ATM, memasukkan kata sandi, memilih opsi penarikan jumlah, memasukkan jumlah yang akan ditarik, menyelesaikan transaksi, dan mengumpulkan jumlah. Beberapa variasi urutan jalur utama (disebut urutan alternatif) mungkin juga ada. Biasanya, variasi dari urutan arus utama terjadi ketika beberapa kondisi tertentu berlaku. Untuk contoh ATM bank, perhatikan variasi atau urutan alternatif berikut:

- Kata sandi tidak valid.
- Jumlah yang akan ditarik melebihi saldo rekening.

Urutan jalur utama dan masing-masing urutan alternatif yang sesuai dengan pemanggilan use case disebut skenario use case. Sebuah use case dapat dilihat sebagai satu set skenario terkait yang diikat bersama oleh tujuan bersama. Urutan jalur utama dan setiap variasinya disebut skenario atau instance dari use case. Setiap skenario adalah jalur tunggal peristiwa pengguna dan aktivitas sistem.

Biasanya, setiap kasus penggunaan tidak tergantung pada kasus penggunaan lainnya. Namun, dependensi implisit di antara kasus penggunaan mungkin ada karena dependensi yang mungkin ada di antara kasus penggunaan di tingkat implementasi karena faktor-faktor seperti sumber daya, objek, atau fungsi bersama. Misalnya, dalam contoh Sistem Otomasi Perpustakaan, perbarui buku dan buku cadangan adalah dua kasus penggunaan independen. Namun, dalam implementasi aktual dari pembaruan buku, pemeriksaan harus dilakukan untuk melihat apakah ada buku yang telah dipesan oleh eksekusi sebelumnya dari kasus penggunaan buku cadangan. Contoh lain dari ketergantungan antar use case adalah sebagai berikut. Dalam Perangkat Lunak Otomasi Toko Buku, perbarui inventaris dan buku penjualan adalah dua kasus penggunaan independen. Namun, selama pelaksanaan sale-book ada ketergantungan implisit pada updateinventory. Karena ketika jumlah yang cukup tidak tersedia dalam persediaan, buku penjualan tidak dapat beroperasi sampai persediaan diisi ulang menggunakan updateinventory. Model use case merupakan artefak analisis dan desain yang penting. Seperti yang telah disebutkan, model UML lainnya harus sesuai dengan model ini dalam setiap pendekatan analisis dan pengembangan berbasis kasus penggunaan (juga disebut sebagai pengguna-sentris). Harus diingat bahwa "model use case" sebenarnya bukan model berorientasi objek menurut definisi istilah yang ketat. Berbeda dengan semua jenis

diagram UML lainnya, model use case mewakili model fungsional atau proses dari suatu sistem.

Representasi Kasus Penggunaan

Sebuah model use case dapat didokumentasikan dengan menggambar diagram use case dan menulis teks yang menyertainya yang menguraikan gambar tersebut. Dalam diagram use case, setiap use case diwakili oleh elips dengan nama use case ditulis di dalam elips. Semua elips (yaitu kasus penggunaan) dari suatu sistem tertutup dalam persegi panjang yang mewakili batas sistem. Nama sistem yang dimodelkan (misalnya, sistem informasi perpustakaan) muncul di dalam persegi panjang.

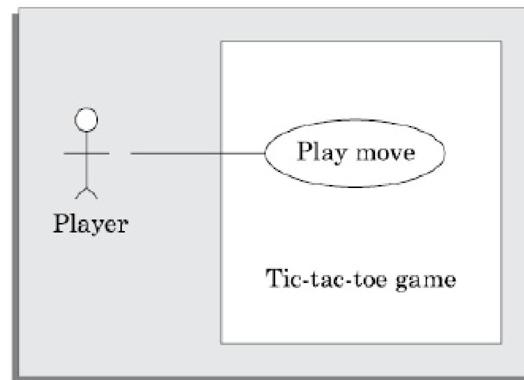
Pengguna sistem yang berbeda diwakili dengan menggunakan ikon stick person. Setiap ikon orang tongkat disebut sebagai aktor. 3 Aktor adalah peran yang dimainkan oleh pengguna sehubungan dengan penggunaan sistem. Ada kemungkinan bahwa pengguna yang sama dapat memainkan peran banyak aktor. Seorang aktor dapat berpartisipasi dalam satu atau lebih use case. Garis yang menghubungkan aktor dan use case disebut hubungan komunikasi. Ini menunjukkan bahwa aktor memanfaatkan fungsionalitas yang disediakan oleh use case. Baik pengguna manusia maupun sistem eksternal dapat diwakili oleh ikon stick person. Ketika ikon orang tongkat mewakili sistem eksternal, itu dijelaskan oleh stereotip <<sistem eksternal>>.

Pada titik ini, perlu dijelaskan konsep stereotip dalam UML. Salah satu tujuan utama dari pencipta UML adalah untuk membatasi jumlah simbol primitif dalam bahasa. Jelas bagi mereka bahwa ketika suatu bahasa memiliki sejumlah besar simbol primitif, menjadi sangat sulit untuk dipelajari penggunaannya. Untuk meyakinkan diri sendiri, pertimbangkan bahwa bahasa Inggris dengan 26 huruf jauh lebih mudah dipelajari dan digunakan dibandingkan dengan bahasa Cina yang memiliki ribuan simbol. Dalam konteks ini, tujuan utama stereotip adalah untuk mengurangi jumlah jenis simbol yang berbeda yang perlu dipelajari seseorang. Konstruk stereotip ketika digunakan untuk membubuhi keterangan pada simbol dasar, dapat memberikan makna yang sedikit berbeda pada simbol dasar—sehingga menghilangkan kebutuhan untuk memiliki beberapa simbol yang maknanya sedikit berbeda satu sama lain.

Sama seperti Anda membuat stereotip teman Anda sebagai orang yang rajin belajar, periang, serius, dll. stereotip dapat digunakan untuk memberi arti khusus pada konstruksi UML dasar apa pun. Kita akan, nanti, melihat bagaimana konstruksi UML lainnya dapat distereotipkan. Kita dapat membuat stereotip simbol ikon orang tongkat untuk menunjukkan sistem eksternal. Jika developer UML telah menetapkan simbol terpisah untuk menunjukkan hal-hal seperti sistem eksternal, maka jumlah simbol dasar yang harus dipelajari dan diingat saat menggunakan UML akan meningkat secara signifikan. Hal ini tentunya akan membuat belajar dan menggunakan UML menjadi jauh lebih sulit.

Anda dapat menggambar persegi panjang di sekitar kasus penggunaan, yang disebut kotak batas sistem, untuk menunjukkan ruang lingkup sistem Anda. Apa pun di dalam kotak mewakili fungsionalitas yang ada dalam ruang lingkup dan apa pun di luar kotak tidak. Namun, menggambar batas sistem adalah opsional. Berikut ini adalah beberapa contoh untuk menggambarkan bagaimana kasus penggunaan sistem dapat didokumentasikan.

Contoh 7.2 Model use case untuk software game Tic-tac-toe ditunjukkan pada Gambar 7.15. Software ini hanya memiliki satu use case, yaitu “play move”. Perhatikan bahwa saya tidak menamai use case "get-user-move", karena "getuser-move" tidak tepat karena ini akan mewakili perspektif developer dari use case. Kasus penggunaan harus diberi nama dari sudut pandang pengguna.



Gambar 7.15 Gunakan model kasus untuk Contoh 7.2.

Deskripsi teks

Setiap elips dalam diagram use case, dengan sendirinya menyampaikan informasi yang sangat sedikit, selain memberikan gambaran kabur tentang use case. Oleh karena itu, setiap use case diagram harus disertai dengan deskripsi teks. Deskripsi teks harus mendefinisikan detail interaksi antara pengguna dan komputer serta aspek lain yang relevan dari use case. Ini harus mencakup semua perilaku yang terkait dengan kasus penggunaan dalam hal urutan jalur utama, berbagai urutan alternatif, respons sistem yang terkait dengan kasus penggunaan, kondisi luar biasa yang mungkin terjadi dalam perilaku, dll. Deskripsi perilaku sering ditulis dalam gaya percakapan yang menggambarkan interaksi antara aktor dan sistem. Deskripsi teks mungkin informal, tetapi beberapa penataan sangat membantu. Berikut ini adalah beberapa informasi yang mungkin disertakan dalam deskripsi teks kasus penggunaan selain urutan jalur utama, dan skenario alternatif.

Kontak person: Bagian ini berisi daftar personel organisasi klien yang dengannya use case didiskusikan, tanggal dan waktu pertemuan, dll.

Aktor: Selain mengidentifikasi aktor, beberapa informasi tentang aktor yang menggunakan use **case** yang dapat membantu implementasi use case dapat direkam.

Prakondisi: Prakondisi akan menggambarkan keadaan sistem sebelum eksekusi use case dimulai.

Post-condition: Ini menangkap status sistem setelah use case berhasil diselesaikan.

Persyaratan non-fungsional : Ini dapat berisi batasan penting untuk desain dan implementasi, seperti kondisi platform dan lingkungan, pernyataan kualitatif, persyaratan waktu respons, dll.

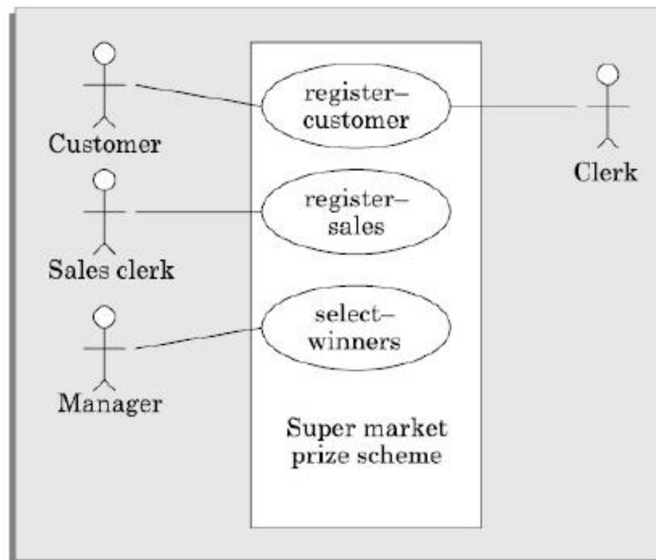
Pengecualian, situasi kesalahan: Ini hanya berisi kesalahan terkait domain seperti kurangnya hak akses pengguna, entri yang tidak valid di bidang input, dll. Jelas, kesalahan yang tidak terkait domain, seperti kesalahan perangkat lunak, tidak perlu dibahas di sini.

Contoh dialog: Ini berfungsi sebagai contoh yang menggambarkan kasus penggunaan.

Persyaratan antarmuka pengguna khusus : Ini berisi persyaratan khusus untuk antarmuka pengguna dari kasus penggunaan. Misalnya, mungkin berisi formulir yang akan digunakan, tangkapan layar, gaya interaksi, dll.

Referensi dokumen: Bagian ini berisi referensi ke dokumen terkait domain tertentu yang mungkin berguna untuk memahami operasi sistem.

Contoh 7.3 Diagram kasus penggunaan skema hadiah pasar Super yang dijelaskan dalam contoh 6.3 ditunjukkan pada Gambar 7.16.



Gambar 7.16 Model kasus penggunaan untuk Contoh 7.3.

Deskripsi teks

U1: daftar-pelanggan: Menggunakan kasus penggunaan ini, pelanggan

Skenario 1: Urutan arus utama

1. Pelanggan: pilih opsi daftar pelanggan
2. Sistem: menampilkan prompt untuk memasukkan nama, alamat, dan nomor telepon.
3. Pelanggan: masukkan nilai yang diperlukan
4. Sistem: menampilkan id yang dihasilkan dan pesan bahwa pelanggan telah berhasil terdaftar.

Skenario 2: Pada langkah 4 dari urutan arus utama

4 : Sistem : menampilkan pesan bahwa pelanggan sudah terdaftar.

Skenario 3: Pada langkah 4 dari urutan arus utama

4 : Sistem: menampilkan pesan bahwa beberapa informasi input belum dimasukkan. Sistem menampilkan prompt untuk memasukkan nilai yang hilang.

U2: register-sales: Dengan use case ini, petugas dapat mendaftarkan detail pembelian yang dilakukan oleh pelanggan.

Skenario 1: Urutan arus utama

1. Petugas: memilih opsi daftar penjualan.
2. Sistem: menampilkan prompt untuk memasukkan detail pembelian dan id pelanggan.
3. Petugas: memasukkan rincian yang diperlukan.
4. Sistem: menampilkan pesan telah berhasil mendaftarkan penjualan.

U3: pemenang terpilih. Dengan menggunakan kasus penggunaan ini, manajer dapat membuat daftar pemenang.

Skenario 2: Urutan arus utama

1. Manajer: memilih opsi pilih pemenang.
2. Sistem: menampilkan koin emas dan daftar pemenang hadiah kejutan.

ngapa Mengembangkan Diagram Kasus Penggunaan?

Jika Anda memeriksa diagram use case, kegunaan dari use case yang diwakili oleh elips akan menjadi jelas. Mereka bersama dengan deskripsi teks yang menyertainya berfungsi sebagai jenis spesifikasi persyaratan sistem dan model yang menjadi dasar pengembangan

semua model lainnya. Dengan kata lain, model use case membentuk model inti yang harus dipatuhi oleh semua model lainnya. Tapi, bagaimana dengan para aktor (ikon orang tongkat)? Apa cara mereka berguna untuk pengembangan sistem? Salah satu kemungkinan penggunaan untuk mengidentifikasi berbagai jenis pengguna (aktor) adalah dalam menerapkan mekanisme keamanan melalui sistem login, sehingga setiap aktor hanya dapat memanggil fungsionalitas yang menjadi haknya. Penggunaan penting lainnya adalah dalam mendesain antarmuka pengguna dalam implementasi use case yang ditargetkan untuk setiap kategori pengguna tertentu yang akan menggunakan use case. Penggunaan lain yang mungkin adalah dalam mempersiapkan dokumentasi (misalnya manual pengguna) yang ditargetkan pada setiap kategori pengguna. Selanjutnya, aktor membantu dalam mengidentifikasi kasus penggunaan dan memahami fungsi sistem yang tepat.

Bagaimana Mengidentifikasi Kasus Penggunaan Sistem?

Identifikasi kasus penggunaan melibatkan brain storming dan meninjau dokumen SRS. Biasanya, persyaratan tingkat tinggi yang ditentukan dalam dokumen SRS sesuai dengan kasus penggunaan. Dengan tidak adanya dokumen SRS yang terformulasi dengan baik, metode populer untuk mengidentifikasi use case adalah berbasis aktor. Ini melibatkan pertama-tama mengidentifikasi berbagai jenis aktor dan penggunaan sistem mereka. Selanjutnya, untuk setiap aktor, fungsi yang berbeda yang mungkin mereka mulai atau berpartisipasi diidentifikasi. Misalnya, untuk Sistem Otomasi Perpustakaan, kategori pengguna dapat berupa anggota, pustakawan, dan akuntan. Setiap pengguna biasanya berfokus pada serangkaian fungsi. Contoh musuh, anggota biasanya mementingkan dirinya sendiri dengan aspek penerbitan buku, pengembalian, dan pembaruan. Pustakawan memperhatikan dirinya sendiri dengan pembuatan dan penghapusan catatan anggota dan buku. Akuntan memperhatikan jumlah yang dikumpulkan dari iuran keanggotaan dan aspek pengeluaran.

Kasus Penggunaan Esensial versus Kasus Penggunaan Nyata

Kasus penggunaan penting dibuat selama elisitasi persyaratan awal. Ini juga merupakan artefak analisis masalah awal. Mereka tidak tergantung pada keputusan desain dan cenderung benar dalam jangka waktu yang lama. Kasus penggunaan nyata menggambarkan fungsionalitas sistem dalam hal desain aktualnya saat ini yang berkomitmen pada teknologi input/output tertentu. Oleh karena itu, kasus penggunaan nyata dapat dikembangkan hanya setelah keputusan desain dibuat. Kasus penggunaan nyata adalah artefak desain. Namun, terkadang organisasi berkomitmen pada kontrak pengembangan yang menyertakan spesifikasi antarmuka pengguna yang terperinci. Dalam kasus seperti itu, tidak ada perbedaan antara use case esensial dan use case nyata.

Pemfaktoran Kesamaan di antara Kasus Penggunaan

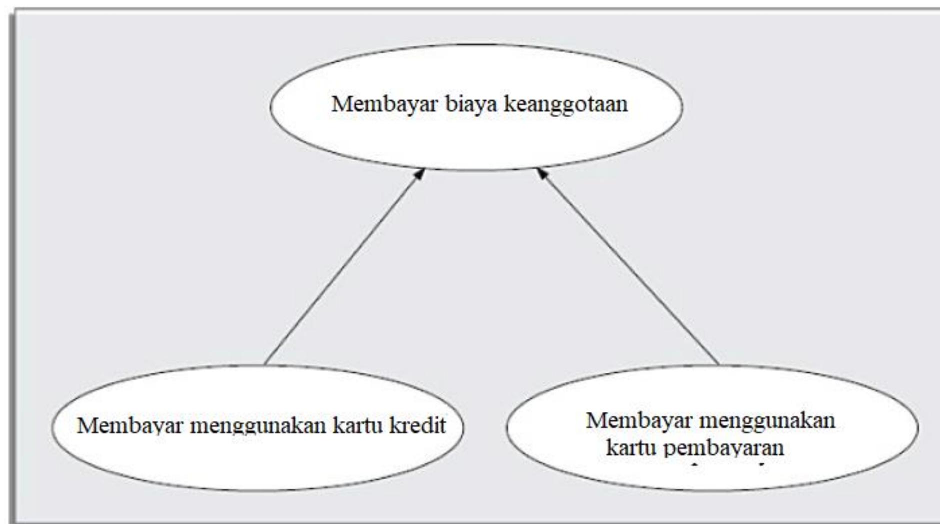
Seringkali diinginkan untuk memfaktorkan use case ke dalam use case komponen. Semua kasus penggunaan tidak perlu diperhitungkan. Faktanya, pemfaktoran kasus penggunaan diperlukan dalam dua situasi sebagai berikut:

- Kasus penggunaan yang kompleks perlu diperhitungkan ke dalam kasus penggunaan yang lebih sederhana. Ini tidak hanya akan membuat perilaku yang terkait dengan use case jauh lebih mudah dipahami, tetapi juga membuat diagram interaksi yang sesuai menjadi lebih mudah dipahami. Tanpa dekomposisi, diagram interaksi untuk kasus penggunaan yang kompleks dapat menjadi terlalu besar untuk ditampung pada kertas berukuran standar tunggal (A4).
- Kasus penggunaan perlu diperhitungkan setiap kali ada perilaku umum di berbagai kasus penggunaan. Anjak akan memungkinkan untuk mendefinisikan perilaku seperti itu hanya sekali dan menggunakannya kembali di mana pun diperlukan.

Diinginkan untuk memfaktorkan penggunaan umum seperti penanganan kesalahan dari serangkaian kasus penggunaan. Ini membuat analisis desain kelas menjadi lebih sederhana dan elegan. Namun, kata peringatan di sini. Anjak piutang use case tidak boleh dilakukan kecuali untuk mencapai dua tujuan di atas. Dari sudut pandang desain, tidak menguntungkan untuk memecah use case menjadi banyak bagian yang lebih kecil hanya untuk kepentingan itu. UML menawarkan tiga mekanisme anjak piutang seperti yang dibahas lebih lanjut.

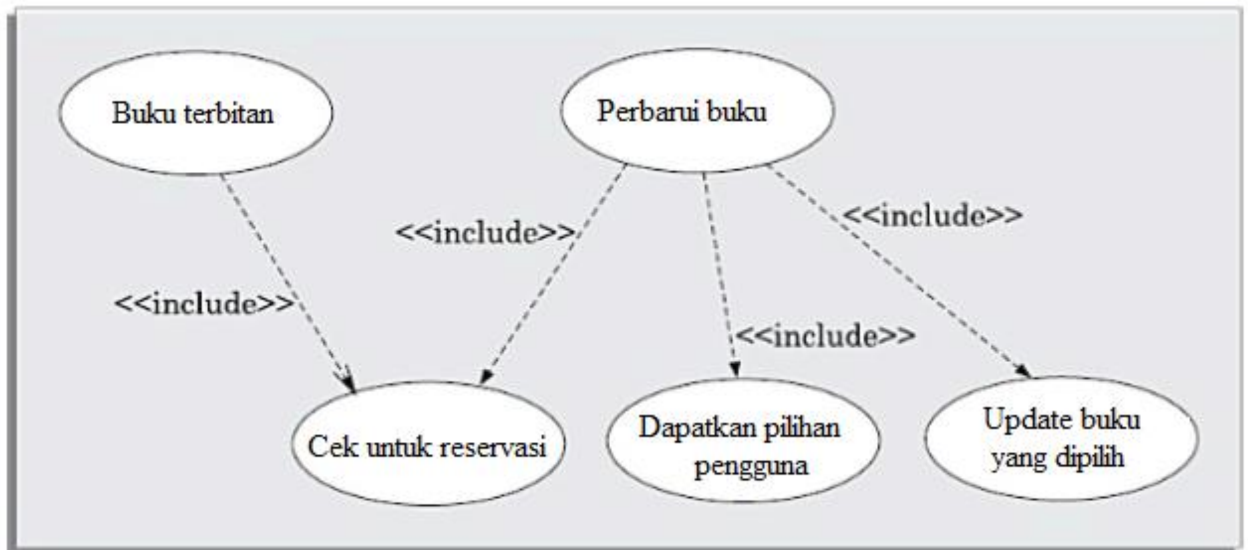
Generalisasi

Generalisasi use case dapat digunakan ketika Anda memiliki satu use case yang mirip dengan yang lain, tetapi melakukan sesuatu yang sedikit berbeda atau sesuatu yang lebih. Generalisasi bekerja dengan cara yang sama dengan kasus penggunaan seperti halnya dengan kelas. Kasus penggunaan anak mewarisi perilaku dan makna dari kasus penggunaan saat ini. Notasinya juga sama (Lihat Gambar 7.17). Penting untuk diingat bahwa basis dan kasus penggunaan turunan adalah kasus penggunaan yang terpisah dan harus memiliki deskripsi teks yang terpisah.



Gambar 7.17 Representasi dari generalisasi use case.

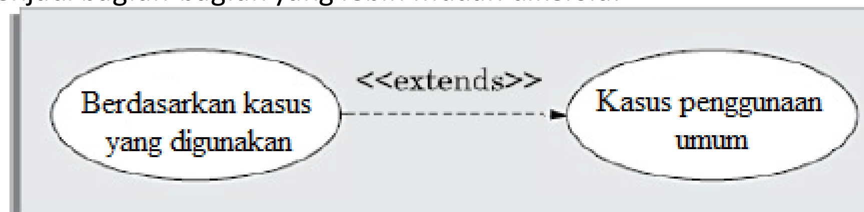
Gambar 7.18 Representasi inklusi use case.



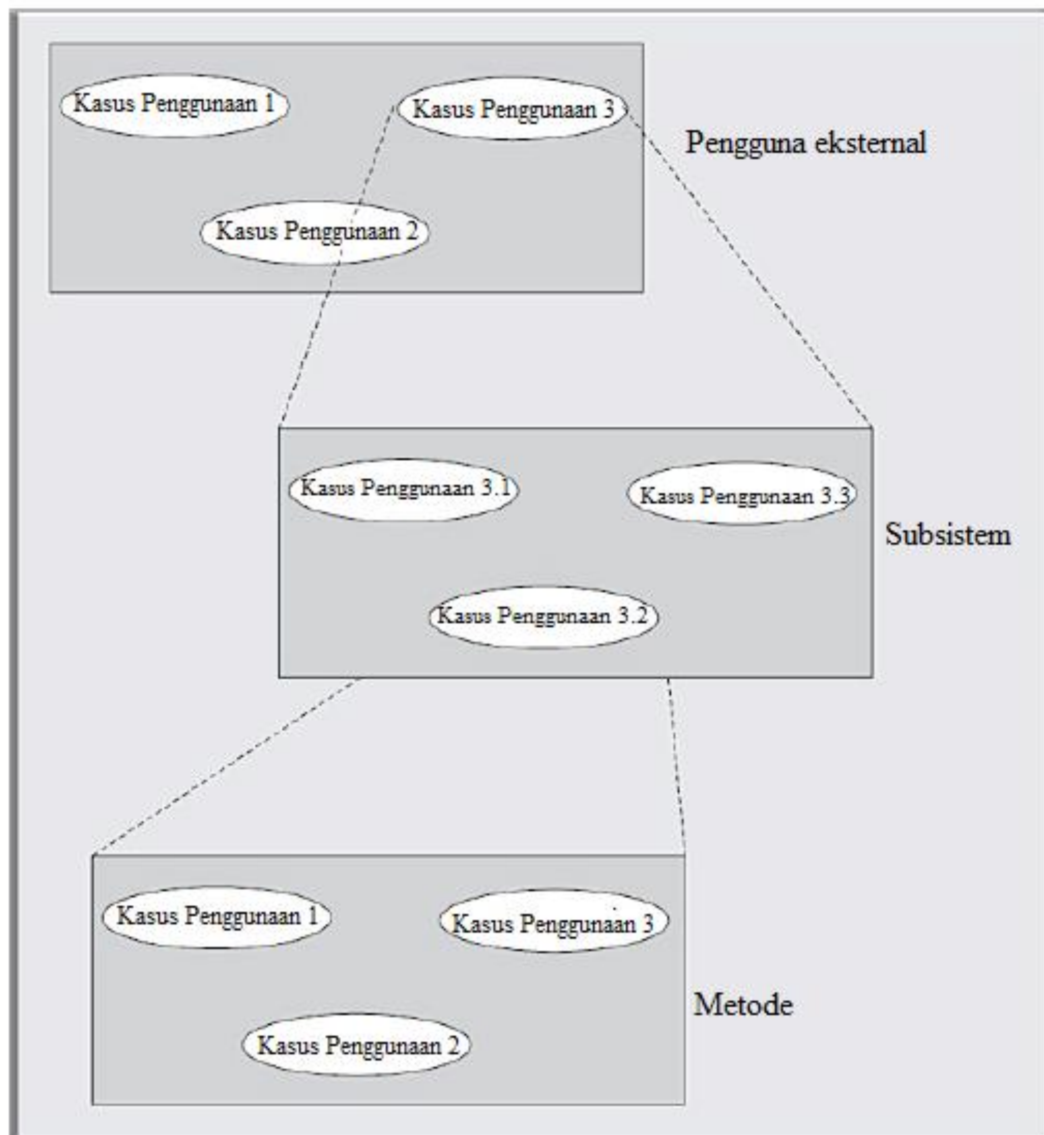
Gambar 7.19 Contoh inklusi use case.

Termasuk

Hubungan include dalam versi UML yang lebih lama (sebelum UML 1.1) dikenal sebagai hubungan penggunaan. Hubungan include menyiratkan satu use case mencakup perilaku use case lain dalam urutan kejadian dan tindakannya. Hubungan penyertaan sesuai ketika Anda memiliki sejumlah perilaku yang serupa di sejumlah kasus penggunaan. Anjak perilaku seperti itu akan membantu dalam tidak mengulangi spesifikasi dan implementasi di berbagai kasus penggunaan. Dengan demikian, hubungan include mengeksplorasi masalah penggunaan kembali dengan memfaktorkan kesamaan di seluruh kasus penggunaan. Ini juga dapat digunakan secara menguntungkan untuk menguraikan kasus penggunaan yang besar dan kompleks menjadi bagian-bagian yang lebih mudah dikelola.



Gambar 7.20 Contoh ekstensi use case.



Gambar 7.21 Organisasi hierarki kasus penggunaan.

Seperti yang ditunjukkan pada Gambar 7.18, hubungan include direpresentasikan menggunakan stereotip yang telah ditentukan sebelumnya <<include>>. Dalam hubungan include, sebuah kasus penggunaan dasar secara wajib dan otomatis menyertakan perilaku kasus penggunaan umum. Seperti yang ditunjukkan pada contoh Gambar 7.19, kasus penggunaan buku terbitan dan buku pembaruan keduanya mencakup kasus penggunaan cek-reservasi. Kasus penggunaan dasar dapat mencakup beberapa kasus penggunaan. Dalam kasus seperti itu, mungkin menyisipkan kasus penggunaan umum yang terkait bersama-sama. Kasus penggunaan umum menjadi kasus penggunaan yang terpisah dan deskripsi teks independen harus disediakan untuk itu.

Memperpanjang

Gagasan utama di balik hubungan yang diperluas di antara kasus penggunaan adalah memungkinkan Anda menunjukkan perilaku sistem opsional. Perilaku sistem opsional dijalankan hanya jika kondisi tertentu terpenuhi, jika tidak, perilaku opsional tidak dijalankan. Hubungan antara use case ini juga telah ditentukan sebelumnya sebagai stereotip seperti yang ditunjukkan pada Gambar 7.20.

Hubungan yang diperluas mirip dengan generalisasi. Tetapi tidak seperti generalisasi, use case yang diperluas dapat menambahkan perilaku tambahan hanya pada titik ekstensi

hanya ketika kondisi tertentu terpenuhi. Titik-titik ekstensi adalah titik-titik dalam kasus penggunaan di mana variasi ke urutan tindakan (normal) arus utama dapat terjadi. Hubungan yang diperluas biasanya digunakan untuk menangkap jalur atau skenario alternatif.

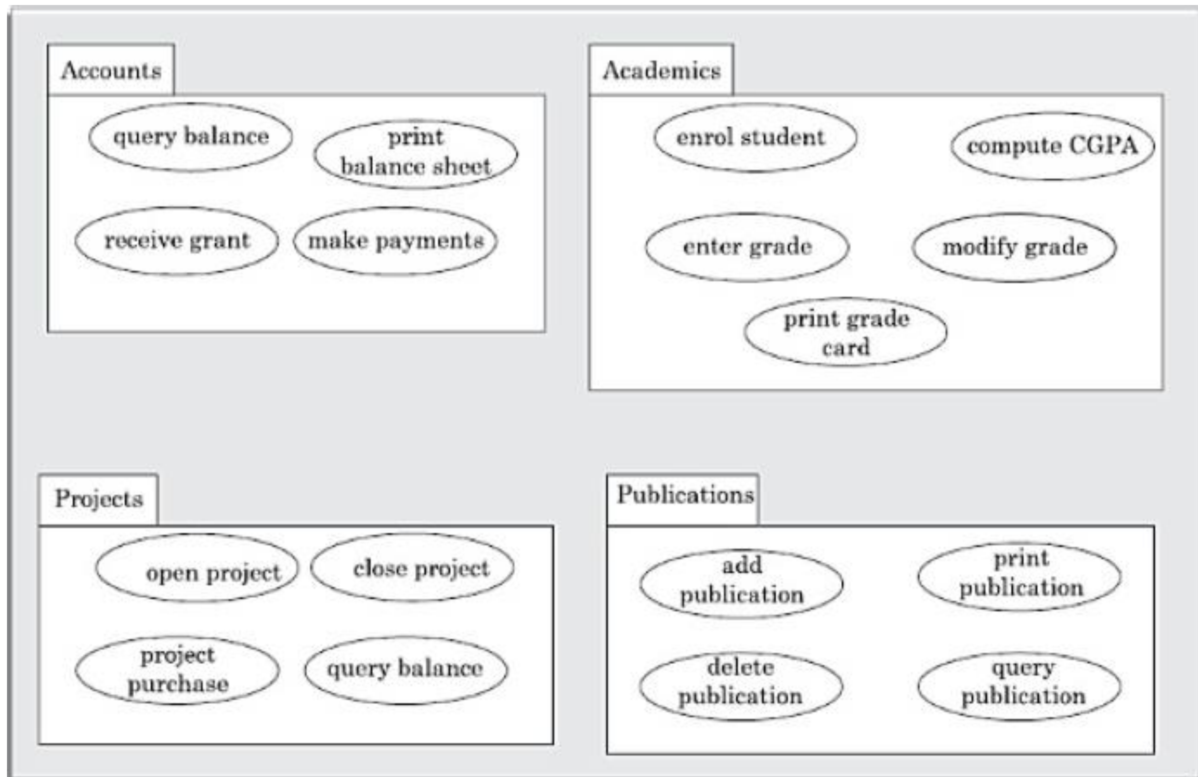
Organisasi

Ketika kasus penggunaan difaktorkan, mereka diatur secara hierarkis. Kasus penggunaan tingkat tinggi disempurnakan menjadi satu set kasus penggunaan yang lebih kecil dan lebih halus seperti yang ditunjukkan pada Gambar 7.21. Kasus penggunaan tingkat atas berada di atas kasus penggunaan yang disempurnakan. Kasus penggunaan yang disempurnakan adalah sub-ordinat dari kasus penggunaan tingkat atas. Perhatikan bahwa hanya kasus penggunaan kompleks yang harus didekomposisi dan diatur dalam hierarki. Tidak perlu menguraikan kasus penggunaan sederhana.

Fungsionalitas *use case super-ordinat* dapat dilacak ke use case bawahannya. Dengan demikian, fungsionalitas yang disediakan oleh use case super-ordinat adalah gabungan dari fungsionalitas use case sub-ordinat. Pada level tertinggi dari model use case, hanya use case fundamental yang ditampilkan. Fokusnya adalah pada konteks aplikasi. Oleh karena itu, level ini juga disebut sebagai diagram konteks. Dalam diagram konteks, batas sistem ditekankan. Dalam diagram tingkat atas, hanya kasus penggunaan yang berinteraksi dengan pengguna eksternal yang ditampilkan. Kasus penggunaan teratas menentukan layanan lengkap yang ditawarkan oleh sistem kepada pengguna eksternal sistem. Kasus penggunaan tingkat subsistem menentukan layanan yang ditawarkan oleh subsistem. Sejumlah level yang melibatkan subsistem dapat digunakan. Di tingkat terendah dari hierarki use case, use case tingkat kelas menentukan fragmen fungsional atau operasi yang ditawarkan oleh kelas.

7.5 GUNAKAN KEMASAN KASUS

Pengemasan adalah mekanisme yang disediakan oleh UML untuk menangani kompleksitas. Ketika kita memiliki terlalu banyak kasus penggunaan dalam diagram tingkat atas, kita dapat mengemas kasus penggunaan terkait sehingga paling banyak 6 atau 7 paket hadir di diagram tingkat atas. Setiap elemen pemodelan yang menjadi besar dan kompleks dapat dipecah menjadi paket. Harap dicatat bahwa Anda dapat menempatkan elemen UML (termasuk paket lain) dalam diagram paket. Simbol untuk sebuah paket adalah folder. Sama seperti Anda mengatur banyak koleksi dokumen dalam folder, Anda mengatur elemen UML ke dalam paket. Contoh kasus penggunaan kemasan ditunjukkan pada Gambar 7.22.



Gambar 7.22 Kemasan kasus penggunaan

7.6 DIAGRAM KELAS

Diagram kelas menggambarkan struktur statis suatu sistem. Ini menunjukkan bagaimana suatu sistem terstruktur daripada bagaimana berperilaku. Struktur statis suatu sistem terdiri dari sejumlah diagram kelas dan dependensinya. Konstituen utama dari diagram kelas adalah kelas dan hubungannya—generalisasi, agregasi, asosiasi, dan berbagai jenis dependensi.

Kelas

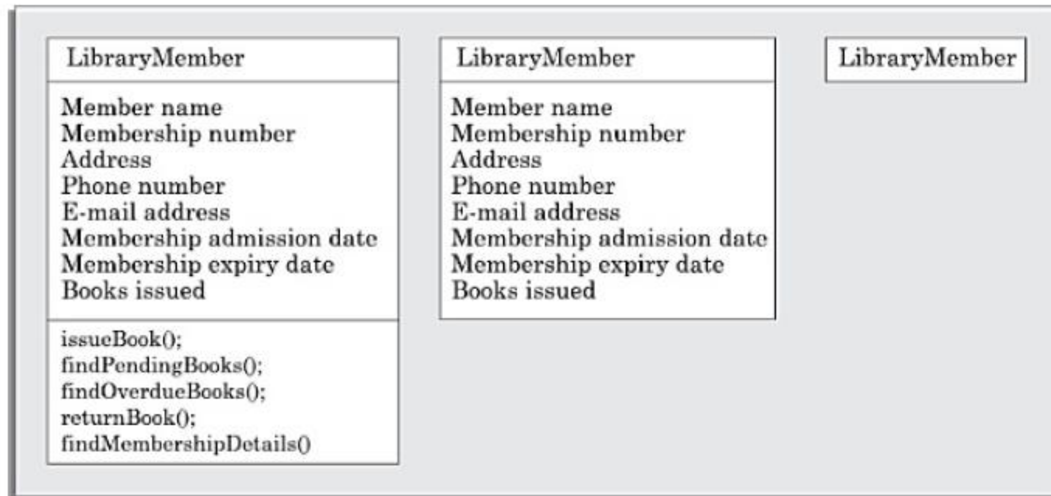
Kelas mewakili entitas dengan fitur umum, yaitu atribut dan operasi. Kelas direpresentasikan sebagai persegi panjang garis padat dengan kompartemen. Kelas memiliki kompartemen nama wajib di mana nama ditulis di tengah dengan huruf tebal. Nama kelas biasanya ditulis menggunakan konvensi kasus campuran dan dimulai dengan huruf besar (misalnya LibraryMember). Nama objek di sisi lain, ditulis menggunakan konvensi kasus campuran, tetapi dimulai dengan huruf kecil (misalnya, studentMember). Nama kelas biasanya dipilih menjadi kata benda tunggal. Contoh berbagai representasi kelas ditunjukkan pada Gambar 7.23.

Kelas memiliki atribut opsional dan kompartemen operasi. Sebuah kelas dapat muncul pada beberapa diagram. Atribut dan operasinya ditekan pada semua kecuali satu diagram. Tapi, orang mungkin bertanya-tanya mengapa ada begitu banyak representasi untuk sebuah kelas! Jawabannya adalah bahwa notasi yang berbeda ini digunakan tergantung pada jumlah informasi tentang kelas yang tersedia. Pada awal proses desain, hanya nama-nama kelas yang diidentifikasi. Ini adalah representasi paling abstrak untuk kelas. Kemudian dalam proses desain metode untuk kelas dan atribut diidentifikasi dan notasi lain yang lebih konkret digunakan.

Atribut

Atribut adalah properti bernama dari sebuah kelas. Ini mewakili jenis data yang mungkin berisi objek. Atribut terdaftar dengan namanya, dan secara opsional dapat berisi

spesifikasi tipenya (yaitu, kelasnya, misalnya, Int, Book, Employee, dll.), nilai awal, dan batasan. Nama atribut ditulis rata kiri menggunakan huruf biasa, dan nama harus dimulai dengan huruf kecil.



Gambar 7.23 Representasi yang berbeda dari kelas LibraryMember.

Nama atribut dapat diikuti dengan tanda kurung siku yang berisi ekspresi multiplisitas, mis. status sensor[10]. Ekspresi multiplisitas menunjukkan jumlah atribut per instance kelas. Atribut tanpa tanda kurung siku harus memiliki tepat satu nilai. Jenis atribut ditulis dengan mengikuti nama atribut dengan titik dua dan nama jenis, (misalnya, sensorStatus[1]:Int).

Nama atribut dapat diikuti dengan ekspresi inisialisasi. Ekspresi inisialisasi dapat terdiri dari tanda sama dengan dan nilai awal yang digunakan untuk menginisialisasi atribut objek yang baru dibuat, mis. sensorStatus[1]:Int=0.

Operasi: Nama operasi biasanya dibiarkan rata, dalam tipe biasa, dan selalu dimulai dengan huruf kecil. Operasi abstrak ditulis dalam *italic* (Ingat bahwa operasi abstrak adalah operasi yang implementasinya tidak disediakan selama definisi kelas.) Parameter fungsi mungkin memiliki jenis yang ditentukan. Jenisnya mungkin "dalam" yang menunjukkan bahwa parameter dilewatkan ke dalam operasi; atau "keluar" yang menunjukkan bahwa parameter hanya dikembalikan dari operasi; atau “inout” yang menunjukkan bahwa parameter tersebut digunakan untuk melewatkan data ke dalam operasi dan mendapatkan hasil dari operasi tersebut. Standarnya adalah "dalam".

Operasi mungkin memiliki tipe pengembalian yang terdiri dari ekspresi tipe pengembalian tunggal, mis., issueBook(in bookName):Boolean. Sebuah operasi mungkin memiliki ruang lingkup kelas (yaitu, dibagi di antara semua objek kelas) dan dilambangkan dengan menggarisbawahi nama operasi. Seringkali perbedaan dibuat antara istilah operasi dan metode. Operasi adalah sesuatu yang didukung oleh sebuah kelas dan dipanggil oleh objek dari kelas lain. Mungkin ada beberapa metode yang mengimplementasikan operasi yang sama, ini disebut polimorfisme statis. Nama metode bisa sama; namun, harus dimungkinkan untuk membedakan antara metode dengan memeriksa parameternya. Dengan demikian, istilah operasi dan metode hanya dapat dibedakan jika terdapat polimorfisme. Ketika hanya ada satu metode yang mengimplementasikan suatu operasi, istilah metode dan operasi tidak dapat dibedakan dan dapat digunakan secara bergantian.

Asosiasi

Asosiasi antara dua kelas diwakili dengan menggambar garis lurus antara kelas yang bersangkutan. Gambar 7.24 mengilustrasikan representasi grafis dari hubungan asosiasi. Nama asosiasi ditulis di sepanjang garis asosiasi. Sebuah panah dapat ditempatkan pada garis

asosiasi untuk menunjukkan arah membaca asosiasi. Kepala panah tidak boleh disalahpahami untuk menunjukkan arah pointer yang mengimplementasikan asosiasi. Di setiap sisi hubungan asosiasi, multiplisitas dicatat sebagai angka individu atau sebagai rentang nilai. Multiplicity menunjukkan berapa banyak instance dari satu kelas yang terkait dengan yang lain. Rentang nilai multiplisitas dicatat dengan menentukan nilai minimum dan maksimum, dipisahkan oleh dua titik, mis. 1.5. Tanda bintang digunakan sebagai kartu liar dan berarti banyak (nol atau lebih). Asosiasi Gambar 7.24 harus dibaca sebagai "Banyak buku dapat dipinjam oleh Anggota Perpustakaan". Biasanya, asosiasi (dan tautan) muncul sebagai kata kerja dalam pernyataan masalah.

Gambar 7.24 Asosiasi antara dua kelas.

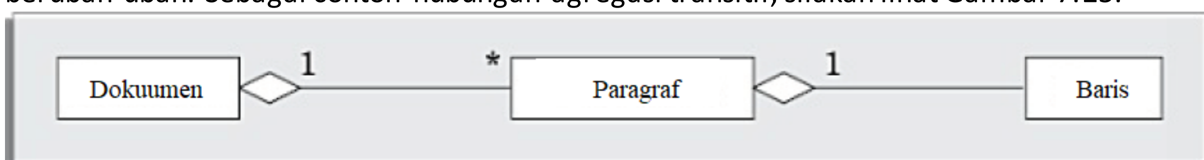
Asosiasi biasanya diwujudkan dengan menetapkan atribut referensi yang sesuai ke kelas yang terlibat. Dengan demikian, asosiasi dapat diimplementasikan dengan menggunakan pointer dari satu kelas objek ke kelas lainnya. Tautan dan asosiasi juga dapat diimplementasikan dengan menggunakan kelas terpisah yang menyimpan objek kelas mana yang ditautkan ke objek kelas lain. Beberapa alat CASE menggunakan nama peran dari relasi asosiasi untuk atribut yang dihasilkan secara otomatis terkait.

Pengumpulan

Agregasi adalah jenis khusus dari hubungan asosiasi di mana kelas yang terlibat tidak hanya terkait satu sama lain, tetapi hubungan seluruh bagian ada di antara mereka. Artinya, objek agregat tidak hanya mengetahui alamat bagian-bagiannya dan karena itu memanggil metode bagian-bagiannya, tetapi juga mengambil tanggung jawab untuk membuat dan menghancurkan bagian-bagiannya. Contoh agregasi, register buku adalah agregasi objek buku. Buku dapat ditambahkan ke register dan dihapus jika diperlukan.

Agregasi diwakili oleh simbol berlian kosong di akhir agregat dari suatu hubungan. Contoh hubungan agregasi telah ditunjukkan pada Gambar 7.25. Angka tersebut mewakili fakta bahwa dokumen dapat dianggap sebagai agregasi paragraf. Setiap paragraf pada gilirannya dapat dianggap sebagai agregasi baris. Perhatikan bahwa angka 1 dianotasi di ujung berlian, dan a * dianotasi di ujung lainnya. Artinya, satu dokumen bisa memiliki banyak paragraf. Di sisi lain, jika kita ingin menunjukkan bahwa sebuah dokumen terdiri dari tepat 10 paragraf, maka kita akan menulis angka 10 sebagai pengganti (*).

Hubungan agregasi tidak bisa refleksif (yaitu rekursif). Artinya, suatu objek tidak dapat berisi objek dari kelas yang sama dengan dirinya sendiri. Juga, hubungan agregasi tidak simetris. Artinya, dua kelas A dan B tidak dapat berisi instance satu sama lain. Namun, hubungan agregasi bisa transitif. Dalam hal ini, agregasi dapat terdiri dari sejumlah level yang berubah-ubah. Sebagai contoh hubungan agregasi transitif, silakan lihat Gambar 7.25.

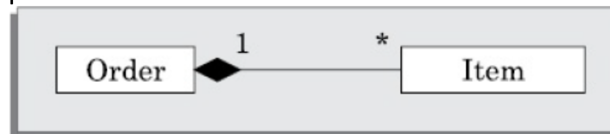


Gambar 7.25 Representasi agregasi.

Komposisi

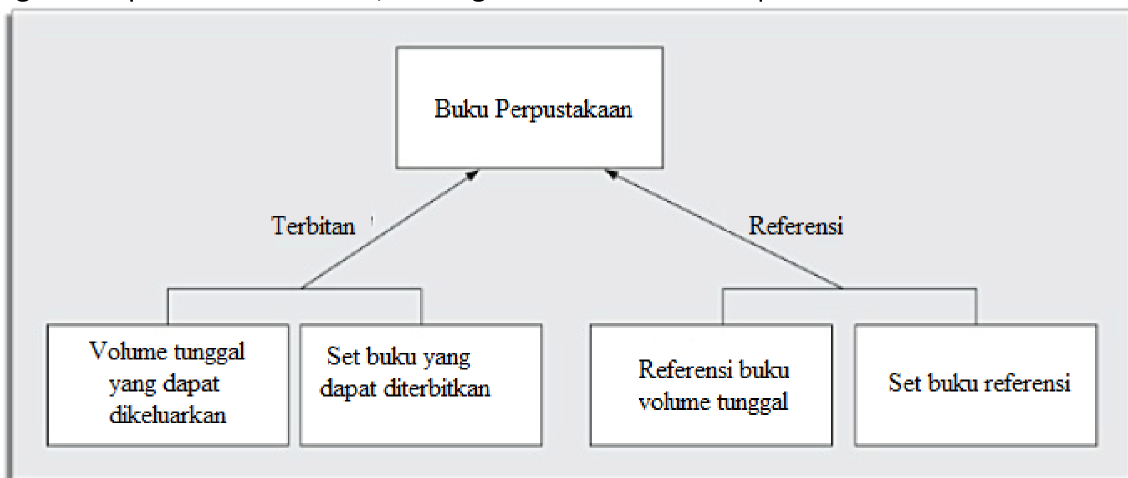
Komposisi adalah bentuk agregasi yang lebih ketat, di mana bagian-bagiannya bergantung pada keberadaan keseluruhan. Ini berarti bahwa kehidupan bagian-bagian tidak dapat ada di luar keseluruhan. Dengan kata lain, garis hidup keseluruhan dan bagian adalah identik. Ketika keseluruhan diciptakan, bagian-bagian dibuat dan ketika keseluruhan dihancurkan, bagian-bagian dihancurkan.

Contoh khas komposisi adalah objek pesanan di mana setelah melakukan pemesanan, tidak ada item dalam pesanan yang tidak dapat diubah. Jika ada perubahan pada salah satu item pesanan yang diperlukan setelah pesanan dilakukan, maka seluruh pesanan harus dibatalkan dan pesanan baru harus dilakukan dengan item yang diubah. Dalam hal ini, segera setelah objek pesanan dibuat, semua item pesanan di dalamnya dibuat dan segera setelah objek pesanan dihancurkan, semua item pesanan di dalamnya juga dihancurkan. Artinya, umur komponen (order item) sama dengan agregat (order). Hubungan komposisi direpresentasikan sebagai berlian terisi yang digambar di ujung komposit. Contoh hubungan komposisi ditunjukkan pada Gambar 7.26.



Gambar 7.26 Representasi komposisi

Agregasi versus Komposisi: Baik agregasi dan komposisi mewakili hubungan bagian/keseluruhan. Ketika komponen secara dinamis dapat ditambahkan dan dihapus dari agregat, maka hubungannya adalah agregasi. Jika komponen tidak dapat ditambahkan/dihapus secara dinamis maka komponen tersebut memiliki life time yang sama dengan komposit. Dalam hal ini, hubungan diwakili oleh komposisi.



Gambar 7.27 Representasi hubungan pewarisan.

Sebagai contoh, perhatikan contoh pesanan yang terdiri dari banyak item pesanan. Jika pesanan sekali ditempatkan, item tidak dapat diubah sama sekali. Dalam hal ini, pesanan merupakan susunan dari barang pesanan. Namun, jika item pesanan dapat diubah (ditambah, dihapus, dan diubah) setelah pesanan ditempatkan, maka relasi agregasi dapat digunakan untuk memodelkannya.

Warisan

Hubungan pewarisan diwakili oleh panah kosong yang menunjuk dari subclass ke superclass. Panah dapat langsung ditarik dari subclass ke superclass. Sebagai alternatif, ketika ada banyak subkelas dari kelas dasar, panah pewarisan dari subkelas dapat digabungkan

menjadi satu baris (lihat Gambar 7.27) dan diberi label dengan aspek kelas yang diabstraksikan. Panah langsung memungkinkan fleksibilitas dalam menyusun diagram dan dapat dengan mudah digambar dengan tangan. Panah gabungan menekankan kolektivitas subclass, ketika spesialisasi telah dilakukan atas dasar beberapa diskriminator. Pada contoh Gambar 7.27, issuable dan reference merupakan diskriminator. Berbagai subclass dari superclass kemudian dapat dibedakan dengan menggunakan diskriminator. Himpunan subclass dari kelas yang memiliki diskriminator yang sama disebut partisi. Seringkali membantu untuk menyebutkan diskriminator selama pemodelan, karena ini menjadi keputusan desain yang terdokumentasi.

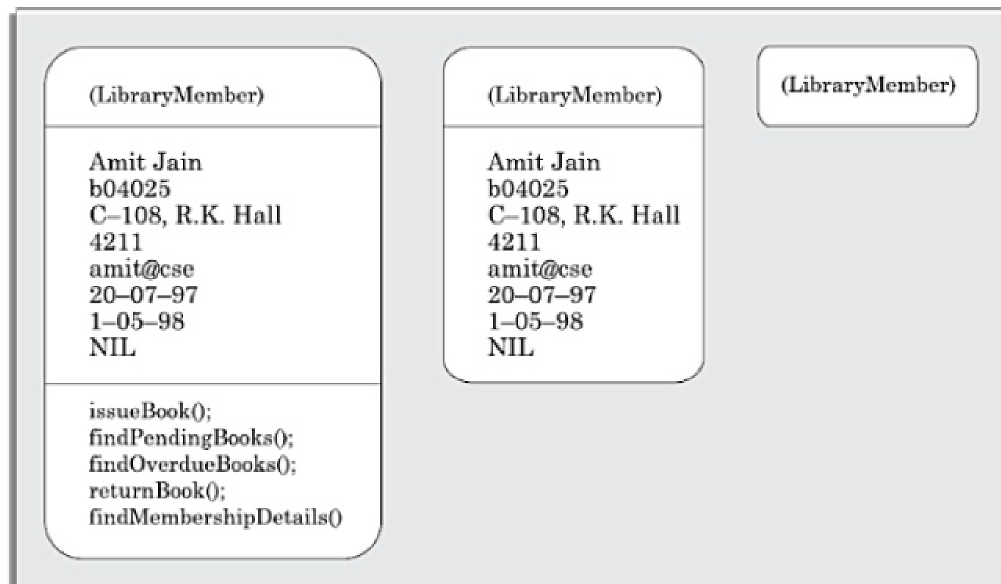
Ketergantungan

Hubungan ketergantungan ditunjukkan sebagai panah putus-putus (lihat Gambar 7.28) yang ditarik dari kelas dependen ke kelas independen.

Gambar 7.28 Representasi ketergantungan antar kelas

Kendala

Batasan menggambarkan kondisi atau aturan integritas. Batasan biasanya digunakan untuk menggambarkan kumpulan nilai atribut yang diizinkan, untuk menentukan kondisi sebelum dan sesudah operasi, untuk menentukan urutan item tertentu, dll. Misalnya, untuk menunjukkan bahwa buku di perpustakaan diurutkan Nomor ISBN kita dapat membubuhi keterangan kelas buku dengan batasan {sorted}.



Gambar 7.29 Representasi berbeda dari objek LibraryMember.

UML memungkinkan Anda menggunakan ekspresi bentuk bebas apa pun untuk menjelaskan batasan. Satu-satunya aturan adalah bahwa mereka harus diapit dalam kurung. Kendala dapat diekspresikan dengan menggunakan bahasa Inggris informal. Namun, UML juga menyediakan bahasa kendala objek (OCL) untuk menentukan batasan. Dalam OCL batasan ditetapkan sebagai bahasa semi-formal, dan oleh karena itu lebih dapat diterima untuk

pemrosesan otomatis dibandingkan dengan batasan informal yang disertakan dalam {}. Pembaca yang tertarik dirujuk ke [Rumbaugh1999].

Diagram objek

Diagram objek menunjukkan snapshot dari objek dalam suatu sistem pada suatu titik waktu. Karena menunjukkan contoh kelas, bukan kelas itu sendiri, sering disebut sebagai diagram contoh. Objek digambar menggunakan persegi panjang bulat (lihat Gambar 7.29). Diagram objek dapat mengalami perubahan terus menerus saat eksekusi berlangsung. Misalnya, tautan mungkin terbentuk di antara objek dan rusak. Objek dapat dibuat dan dihancurkan, dan seterusnya. Diagram objek berguna untuk menjelaskan cara kerja suatu sistem.

7.7 DIAGRAM INTERAKSI

Ketika pengguna memanggil salah satu fungsi yang didukung oleh sistem, perilaku yang diperlukan diwujudkan melalui interaksi beberapa objek dalam sistem. Diagram interaksi, seperti namanya sendiri, adalah model yang menggambarkan bagaimana kelompok objek berinteraksi di antara mereka sendiri melalui pesan lewat untuk mewujudkan beberapa perilaku. Biasanya, setiap diagram interaksi menyadari perilaku kasus penggunaan tunggal.

Terkadang, terutama untuk kasus penggunaan yang kompleks, lebih dari satu diagram interaksi mungkin diperlukan untuk menangkap perilaku. Diagram interaksi menunjukkan sejumlah objek contoh dan pesan yang dikirimkan antara objek dalam use case. Ada dua jenis diagram interaksi — diagram urutan dan diagram kolaborasi. Kedua diagram ini setara dalam arti bahwa salah satu diagram dapat diturunkan secara otomatis dari yang lain. Namun, keduanya berguna. Keduanya benar-benar menggambarkan perspektif yang berbeda dari perilaku suatu sistem dan berbagai jenis kesimpulan dapat diambil dari mereka. Diagram interaksi memainkan peran utama dalam berorientasi proses desain objek yang efektif.

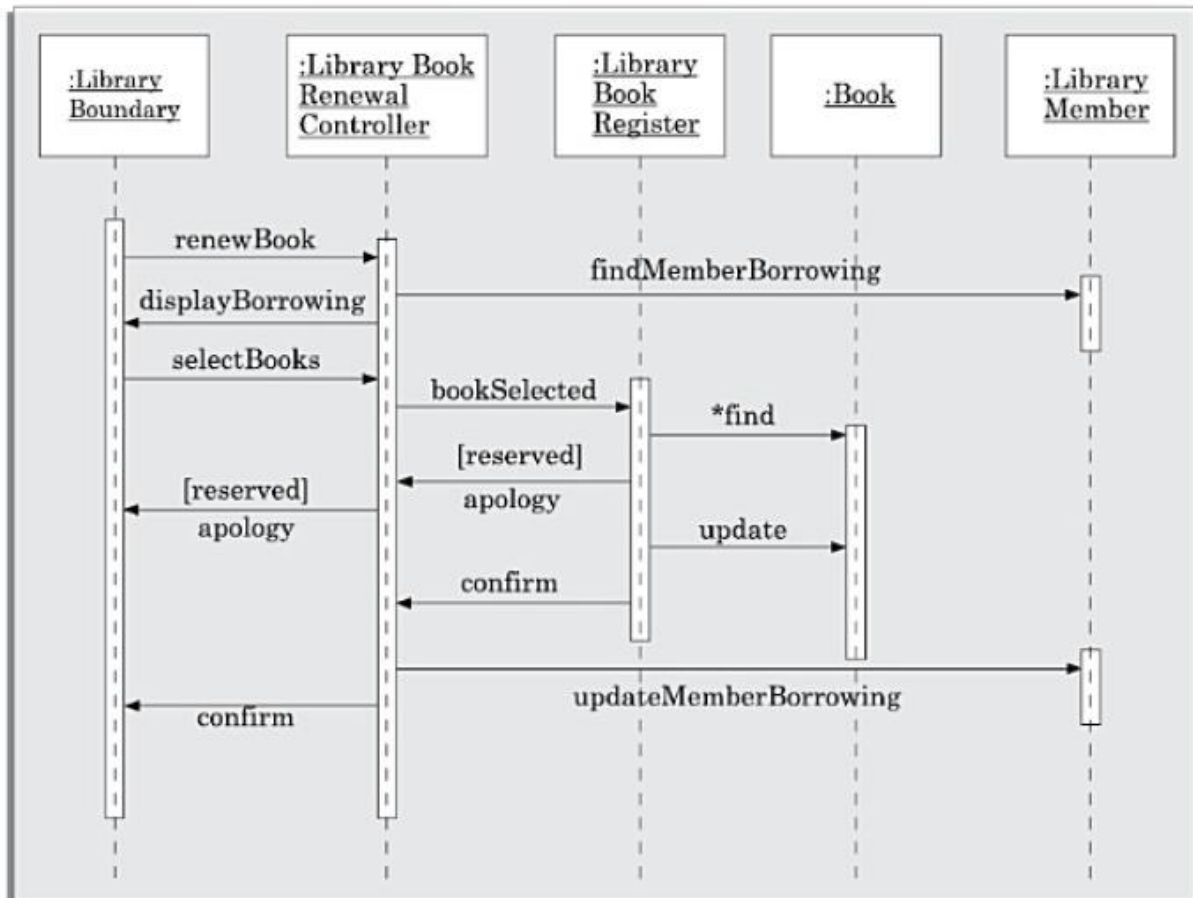
Diagram urutan

Sebuah diagram urutan menunjukkan interaksi antara objek sebagai grafik dua dimensi. Grafik dibaca dari atas ke bawah. Objek yang berpartisipasi dalam interaksi ditampilkan di bagian atas bagan sebagai kotak yang dilampirkan ke garis putus-putus vertikal. Di dalam kotak nama objek ditulis dengan tanda titik dua yang memisahkannya dari nama kelas dan nama objek dan kelas digarisbawahi. Ini menandakan bahwa kita merujuk setiap instance kelas yang sewenang-wenang. Sebagai contoh, pada Gambar 7.30 :Book merepresentasikan sembarang instance dari kelas Book.

Objek yang muncul di bagian atas diagram urutan menandakan bahwa objek itu ada bahkan sebelum eksekusi use case dimulai. Namun, jika beberapa objek dibuat selama eksekusi use case dan berpartisipasi dalam interaksi (misalnya, pemanggilan metode), maka objek tersebut harus ditampilkan di tempat yang sesuai pada diagram tempat objek tersebut dibuat. Garis putus-putus vertikal disebut garis hidup objek. Setiap titik pada garis hidup menyiratkan bahwa objek ada pada titik itu. Tidak adanya garis hidup setelah beberapa titik menunjukkan bahwa objek tidak lagi ada setelah titik waktu itu, titik waktu tertentu. Biasanya, pada titik jika suatu objek dihancurkan, garis hidup objek dilintasi pada titik itu dan gari s hidup untuk objek tersebut tidak ditarik melampaui titik itu. Sebuah persegi panjang yang disebut simbol aktivasi digambar pada garis hidup suatu objek untuk menunjukkan titik waktu di mana objek tersebut aktif. Jadi simbol aktivasi menunjukkan bahwa suatu objek aktif selama simbol (persegi panjang) ada di garis hidup. Setiap pesan ditunjukkan sebagai panah di antara garis hidup dua objek. Pesan ditampilkan dalam urutan kronologis dari atas ke bawah. Artinya, membaca diagram dari atas ke bawah akan menunjukkan urutan terjadinya pesan. Setiap

pesan diberi label dengan nama pesan. Beberapa informasi kontrol juga dapat disertakan. Dua jenis informasi kontrol yang penting adalah:

- Sebuah kondisi (misalnya, [tidak valid]) menunjukkan bahwa pesan dikirim, hanya jika kondisinya benar.
- Penanda iterasi menunjukkan bahwa pesan dikirim berkali-kali ke beberapa objek penerima seperti yang akan terjadi saat Anda mengulangi koleksi atau elemen larik. Anda juga dapat menunjukkan dasar iterasi, mis., [untuk setiap objek buku].



Gambar 7.30 Diagram urutan untuk kasus penggunaan buku pembaruan

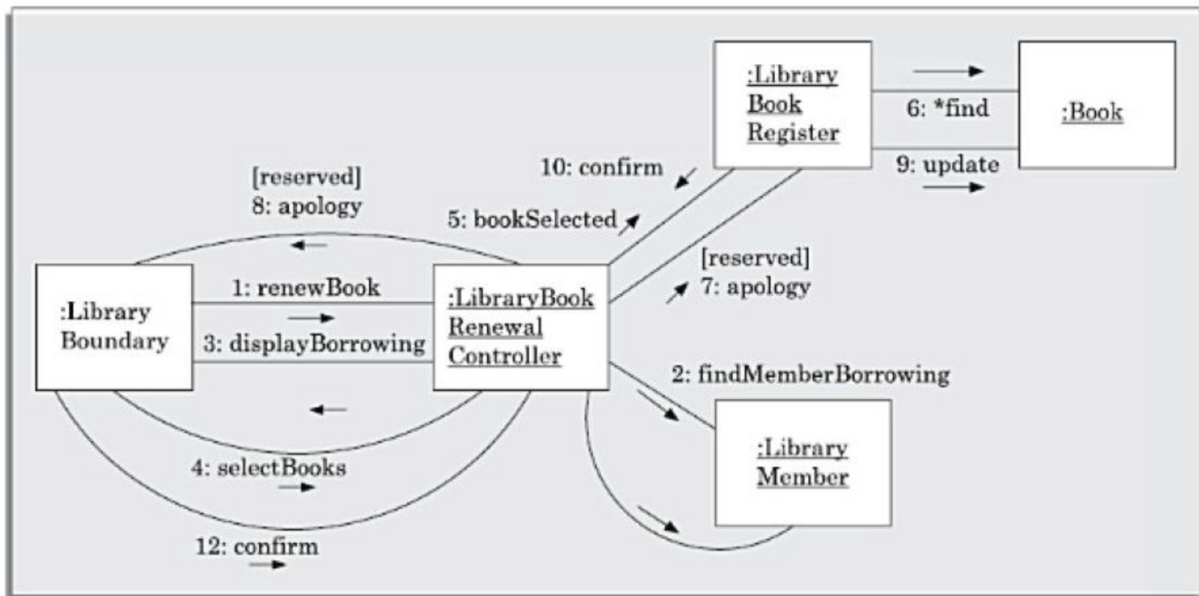
Diagram urutan use case pembaruan buku untuk Software Otomasi Perpustakaan ditunjukkan pada Gambar 7.30. Amati bahwa objek-objek eksak yang berpartisipasi untuk mewujudkan perilaku pembaruan buku dan urutan interaksinya dapat disimpulkan dengan jelas dari diagram urutan. Pengembangan diagram urutan dalam metodologi pengembangan (dibahas dalam Bab 8) akan membantu kita untuk menentukan tanggung jawab yang harus diberikan ke kelas yang berbeda; yaitu, metode apa yang harus didukung oleh setiap kelas.

Diagram kolaborasi

Diagram kolaborasi menunjukkan aspek struktural dan perilaku secara eksplisit. Ini tidak seperti diagram urutan yang hanya menunjukkan aspek perilaku. Aspek struktural diagram kolaborasi terdiri dari objek dan tautan di antaranya yang menunjukkan asosiasi. Dalam diagram ini, setiap objek juga disebut kolaborator. Aspek perilaku dijelaskan oleh serangkaian pesan yang dipertukarkan di antara kolaborator yang berbeda.

Tautan antar objek ditampilkan sebagai garis padat dan dapat digunakan untuk mengirim pesan antara dua objek. Pesan ditampilkan sebagai panah berlabel yang ditempatkan di dekat tautan. Pesan diawali dengan nomor urut karena merupakan satu-satunya cara untuk menggambarkan urutan relatif pesan dalam diagram ini. Diagram

kolaborasi untuk contoh Gambar 7.30 ditunjukkan pada Gambar 7.31. Penggunaan diagram kolaborasi dalam proses pengembangan akan membantu dalam menentukan kelas mana yang terkait dengan kelas lain.



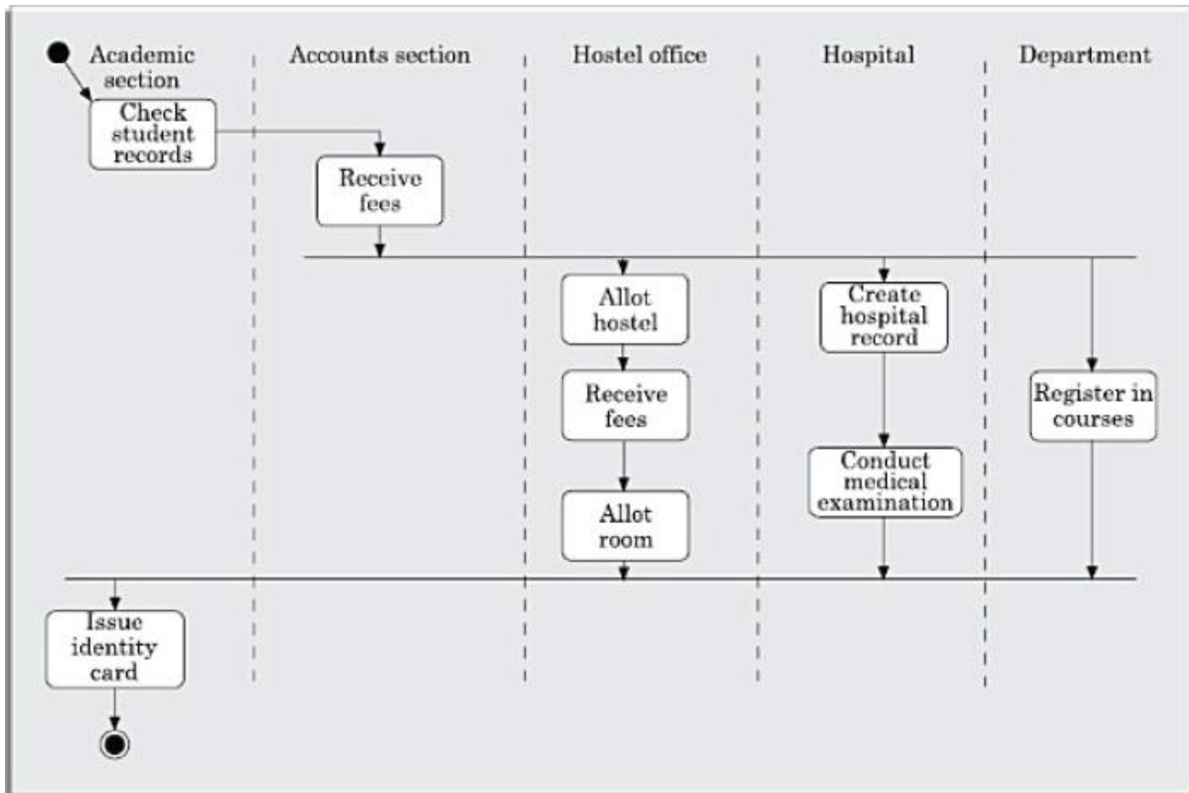
Gambar 7.31 Diagram kolaborasi untuk use case pembaruan buku.

7.8 DIAGRAM AKTIVITAS

Diagram aktivitas mungkin merupakan salah satu elemen pemodelan yang tidak ada di salah satu pendahulu UML. Tidak ada diagram seperti itu baik dalam karya Booch, Jacobson, atau Rumbaugh. Ini mungkin didasarkan pada diagram peristiwa Odell [1992] meskipun notasinya sangat berbeda dari yang digunakan oleh Odell.

Diagram aktivitas berfokus pada representasi berbagai aktivitas atau potongan pemrosesan dan urutan aktivasinya. Kegiatan secara umum mungkin tidak sesuai dengan metode kelas. Aktivitas adalah keadaan dengan tindakan internal dan satu atau lebih transisi keluar yang secara otomatis mengikuti penghentian aktivitas internal. Jika suatu aktivitas memiliki lebih dari satu transisi keluar, maka situasi yang tepat di mana masing-masing dijalankan harus diidentifikasi melalui kondisi yang sesuai. Diagram aktivitas mirip dengan diagram alur prosedural. Perbedaan utama adalah bahwa diagram aktivitas mendukung deskripsi aktivitas paralel dan aspek sinkronisasi yang terlibat dalam aktivitas yang berbeda.

Aktivitas paralel direpresentasikan pada diagram aktivitas dengan menggunakan jalur renang. Jalur renang memungkinkan Anda mengelompokkan aktivitas berdasarkan siapa yang melakukannya, misalnya departemen akademik vs. kantor asrama. Jadi jalur renang membagi kegiatan berdasarkan tanggung jawab beberapa komponen. Aktivitas di jalur renang dapat ditetapkan ke beberapa elemen model, mis. kelas atau beberapa komponen, dll. Misalnya, pada Gambar 7.32 jalur renang yang sesuai dengan bagian akademik, kegiatan yang dilakukan oleh bagian akademik dan situasi spesifik di mana ini dilakukan.



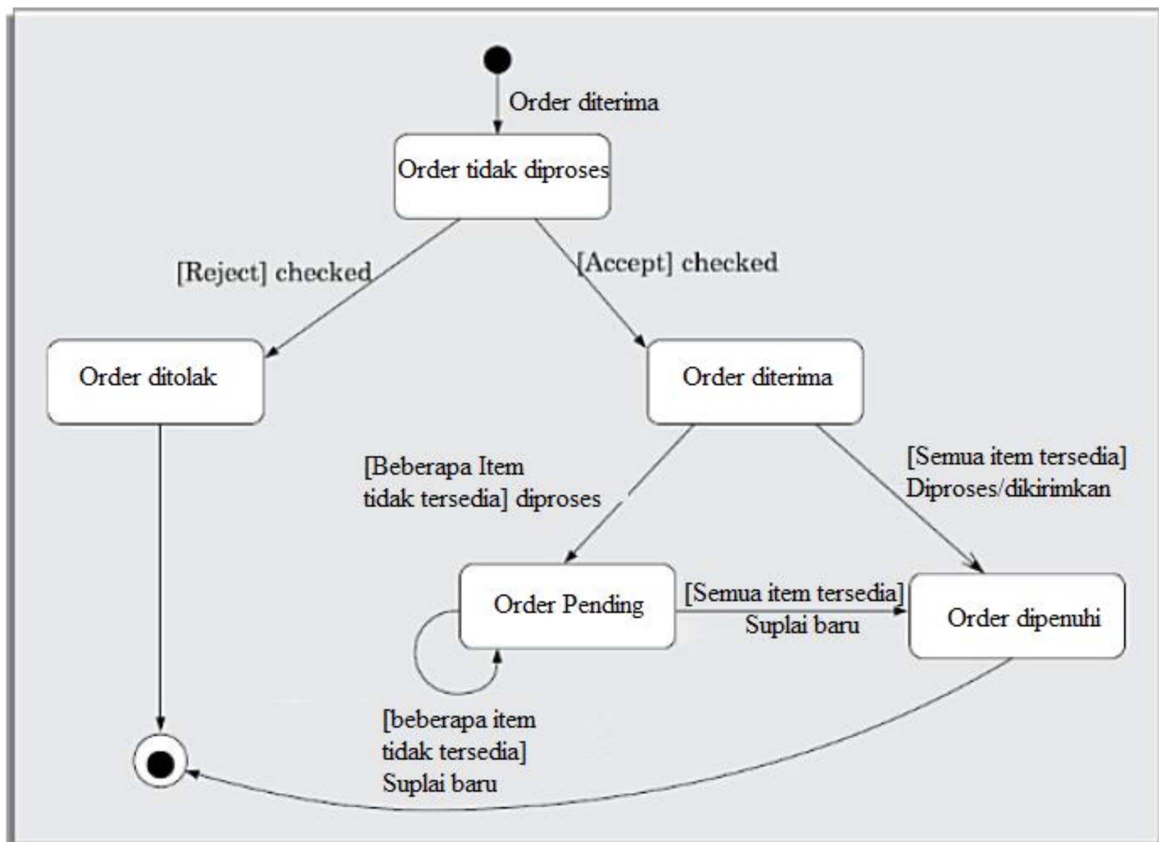
Gambar 7.32 Activity diagram prosedur penerimaan mahasiswa di IIT.

Diagram aktivitas biasanya digunakan dalam pemodelan proses bisnis. Ini dilakukan selama tahap awal analisis kebutuhan dan spesifikasi. Diagram aktivitas dapat sangat berguna untuk memahami aktivitas pemrosesan kompleks yang melibatkan peran yang dimainkan oleh banyak komponen. Selain membantu developer untuk memahami aktivitas pemrosesan yang kompleks, diagram ini juga dapat digunakan untuk mengembangkan diagram interaksi yang membantu mengalokasikan aktivitas (tanggung jawab) ke kelas.

Proses penerimaan mahasiswa di IIT ditunjukkan dengan diagram aktivitas pada Gambar 7.32. Ini menunjukkan peran yang dimainkan oleh berbagai komponen Institut dalam prosedur penerimaan. Setelah biaya diterima di bagian rekening, kegiatan paralel dimulai di kantor asrama, rumah sakit, dan Departemen. Setelah semua kegiatan ini selesai (ini adalah masalah sinkronisasi dan direpresentasikan sebagai garis horizontal), kartu identitas dapat dikeluarkan untuk mahasiswa oleh bagian Akademik.

7.9 DIAGRAM BAGAN NEGARA

Diagram bagan keadaan biasanya digunakan untuk memodelkan bagaimana keadaan suatu objek berubah dalam waktu hidupnya. Diagram bagan keadaan bagus dalam menggambarkan bagaimana perilaku suatu objek berubah di beberapa eksekusi kasus penggunaan. Namun, jika kita tertarik untuk memodelkan beberapa perilaku yang melibatkan beberapa objek yang saling berkolaborasi, diagram state chart tidak tepat. Kita telah melihat bahwa perilaku seperti itu lebih baik dimodelkan menggunakan diagram urutan atau kolaborasi. Diagram bagan negara didasarkan pada formalisme mesin negara hingga (FSM). FSM terdiri dari sejumlah status terbatas yang sesuai dengan objek yang dimodelkan. Objek mengalami perubahan keadaan ketika peristiwa tertentu terjadi. Formalisme FSM ada jauh sebelum teknologi berorientasi objek dan telah digunakan untuk berbagai macam aplikasi. Selain pemodelan, bahkan telah digunakan dalam ilmu komputer teoretis sebagai generator untuk bahasa reguler.



Gambar 7.33 Diagram diagram keadaan untuk objek pesanan

Mengapa grafik negara bagian?

Kerugian utama dari formalisme FSM adalah masalah ledakan negara. Jumlah state menjadi terlalu banyak dan model menjadi terlalu kompleks ketika digunakan untuk memodelkan sistem praktis. Masalah ini adalah diatasi di UML dengan menggunakan grafik status. Formalisme bagan negara diusulkan oleh David Harel [1990]. Bagan keadaan adalah model hierarkis dari suatu sistem dan memperkenalkan konsep keadaan gabungan (juga disebut keadaan bersarang). Tindakan dikaitkan dengan transisi dan dianggap sebagai proses yang terjadi dengan cepat dan tidak dapat diinterupsi. Aktivitas terkait dengan status dan dapat memakan waktu lebih lama. Suatu aktivitas dapat diinterupsi oleh suatu peristiwa.

Elemen dasar bagan negara bagian

Elemen dasar diagram bagan negara adalah sebagai berikut:

Keadaan awal: Ini direpresentasikan sebagai lingkaran penuh.

Keadaan akhir: Ini diwakili oleh lingkaran penuh di dalam lingkaran yang lebih besar.

Negara: Ini diwakili oleh persegi panjang dengan sudut membulat.

Transisi: Sebuah transisi ditampilkan sebagai panah antara dua negara. Biasanya, nama peristiwa yang menyebabkan transisi ditempatkan di sepanjang sisi panah. Anda juga dapat menetapkan penjaga untuk transisi. Guard adalah kondisi logika Boolean. Transisi hanya dapat terjadi jika guard bernilai true. Sintaks untuk label transisi ditampilkan dalam 3 bagian—[guard]event/action. Contoh bagan status untuk objek pesanan perangkat lunak Trade House Automation ditunjukkan pada Gambar 7.33. Perhatikan bahwa dari status pesanan Ditolak, ada transisi otomatis dan implisit ke status akhir. Transisi seperti ini disebut transisi semu.

7.10 NOTA BENE

UML telah mendapatkan penerimaan yang cepat di kalangan praktisi dan akademisi dalam waktu singkat dan telah membuktikan kegunaannya dalam mencapai solusi desain yang baik untuk masalah pengembangan perangkat lunak.

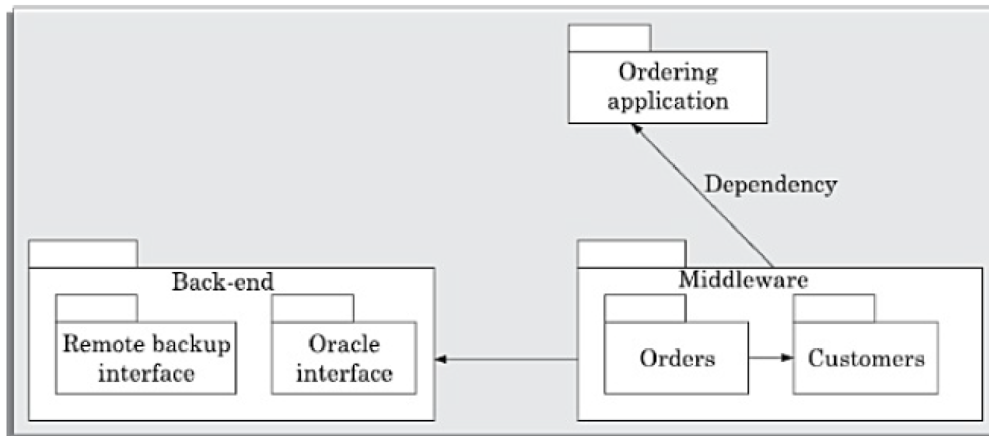
Diskusi tentang UML disini hanya berkonsentrasi pada aspek-aspek yang diperlukan untuk memecahkan masalah desain perangkat lunak tradisional berukuran sedang. Sebelum mengakhiri bab ini, gambaran umum tentang beberapa aspek yang telah dipilih untuk ditinggalkan akan diberikan disini. Pertama membahas diagram paket dan penerapan. Karena UML telah mengalami perubahan yang signifikan dengan dirilisnya UML 2.0 pada tahun 2003. Secara singkat disebutkan bahwa peningkatan yang dibawa UML 2.0 atas UML 1.X yang menjadi fokus kita sejauh ini. Revisi signifikan ini diperlukan untuk membuat UML dapat diterapkan pada pengembangan perangkat lunak untuk domain tertanam dan telekomunikasi yang baru muncul.

Paket, Komponen, dan Diagram Penerapan

Dalam subbagian berikut, Saya berikan gambaran singkat tentang paket, komponen, dan diagram penerapan:

Diagram paket

Paket adalah pengelompokan beberapa kelas. Bahkan, diagram paket dapat digunakan untuk mengelompokkan artefak UML apa pun. Paket adalah cara populer untuk mengatur file kode sumber. Paket Java adalah contoh yang baik yang dapat dimodelkan menggunakan diagram paket. Diagram paket tersebut menunjukkan kelompok kelas yang berbeda (paket) dan saling ketergantungannya. Ini sangat berguna untuk mendokumentasikan organisasi file sumber untuk proyek besar yang memiliki banyak file program. Contoh diagram paket ditunjukkan pada Gambar 7.34.



Gambar 7.34 Contoh diagram paket.

Perhatikan, bahwa sebuah paket mungkin berisi paket lebih lanjut.

Diagram komponen

Komponen mewakili bagian dari perangkat lunak yang dapat dibeli, ditingkatkan, dan diintegrasikan secara independen ke dalam perangkat lunak yang ada. Diagram komponen dapat digunakan untuk mewakili struktur fisik dari suatu implementasi dalam hal berbagai komponen sistem. Diagram komponen biasanya digunakan untuk mencapai tujuan berikut:

- Atur kode sumber agar dapat membuat rilis yang dapat dieksekusi.
- Tentukan dependensi di antara komponen yang berbeda.

Diagram paket dapat digunakan untuk memberikan tampilan tingkat tinggi dari setiap komponen dalam hal kelas berbeda yang dikandungnya.

Diagram penerapan

Diagram penyebaran menunjukkan pandangan lingkungan dari suatu sistem. Artinya, ia menangkap lingkungan di mana solusi perangkat lunak diimplementasikan. Dengan kata lain, diagram penyebaran menunjukkan bagaimana sistem perangkat lunak akan digunakan secara fisik di lingkungan perangkat keras. Yaitu, komponen mana yang akan mengeksekusi komponen perangkat keras mana dan bagaimana mereka akan berkomunikasi satu sama lain. Karena diagram memodelkan arsitektur waktu berjalan dari suatu aplikasi, diagram ini dapat sangat berguna bagi staf operasi sistem.

Pandangan lingkungan yang disediakan oleh diagram penyebaran penting untuk solusi perangkat lunak yang kompleks dan besar yang berjalan pada sistem perangkat keras yang terdiri dari beberapa komponen. Dalam hal ini, diagram penyebaran memberikan gambaran tentang bagaimana komponen yang berbeda didistribusikan di antara komponen perangkat keras yang berbeda dari sistem.

7.11 UML 2.0

UML 1.X tidak memiliki beberapa kemampuan khusus yang membuatnya sulit untuk digunakan di beberapa domain non-tradisional. Beberapa fitur yang sangat kurang di UML 1.X termasuk kurangnya dukungan untuk representasi berikut—eksekusi metode secara bersamaan, domain pengembangan, pesan asinkron, peristiwa, port, dan objek aktif. Dalam banyak aplikasi, termasuk pengembangan perangkat lunak tertanam dan telekomunikasi, kemampuan untuk memodelkan persyaratan waktu menggunakan diagram waktu sangat diperlukan untuk membuat UML dapat diterapkan di segmen penting pengembangan perangkat lunak ini. Selanjutnya, perubahan tertentu diperlukan untuk mendukung interoperabilitas di antara alat CASE berbasis UML menggunakan pertukaran metadata XML (XMI). UML 2.0 mendefinisikan tiga belas jenis diagram, dibagi menjadi tiga kategori sebagai berikut:

Diagram struktur: Ini termasuk diagram kelas, diagram objek, diagram komponen, diagram struktur komposit, diagram paket, dan diagram penyebaran.

Diagram perilaku: Diagram ini mencakup diagram kasus penggunaan, diagram aktivitas, dan diagram mesin keadaan.

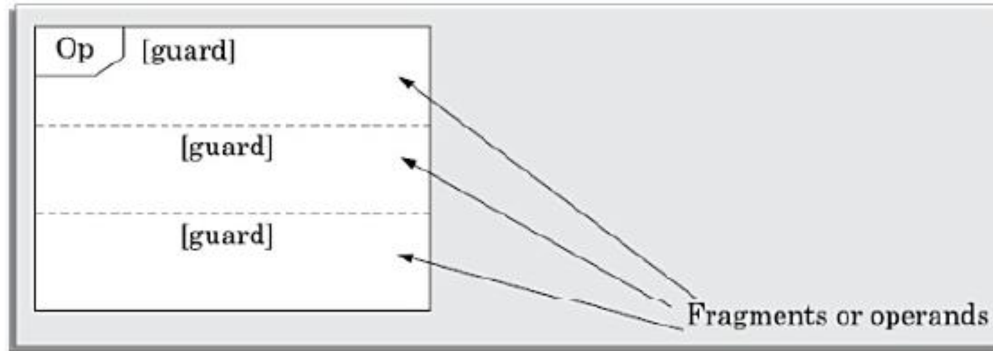
Diagram interaksi: Diagram ini termasuk diagram urutan, diagram komunikasi, diagram waktu, dan diagram gambaran interaksi. Diagram kolaborasi UML 1.X telah diganti namanya di UML 2.0 sebagai diagram komunikasi. Penggantian nama ini diperlukan karena nama sebelumnya agak menyesatkan, ini menunjukkan komunikasi antar kelas selama pelaksanaan use case daripada menunjukkan pemecahan masalah kolaboratif. Meskipun sejumlah besar fitur baru telah diperkenalkan di UML 2.0 dibandingkan dengan 1.X, disini saya hanya akan membahas dua peningkatan di UML2.0 melalui fragmen gabungan dan diagram struktur komposit.

Fragmen gabungan dalam diagram urutan

Fragmen gabungan adalah konstruksi yang telah diperkenalkan di UML 2.0 untuk memungkinkan deskripsi berbagai kontrol dan struktur logika dengan cara yang lebih jelas dan ringkas secara visual. Hal ini juga memungkinkan representasi perilaku eksekusi bersamaan seperti yang terjadi dalam situasi eksekusi mutithreaded. Sekarang, mari kita memahami anatomi fragmen gabungan dan penggunaannya. Fragmen gabungan membagi diagram urutan menjadi sejumlah area atau fragmen yang memiliki perilaku berbeda (lihat Gambar 7.35). Fragmen gabungan muncul di atas area diagram urutan untuk membuat aspek kontrol dan logika tertentu menjadi jelas secara visual. Seperti yang ditunjukkan pada Gambar 7.35, fragmen gabungan terdiri dari banyak fragmen dan operator yang ditunjukkan di sudut kiri

atas. Setiap fragmen dapat dikaitkan dengan penjaga (ekspresi Boolean). Komponen-komponen ini dari fragmen gabungan adalah sebagai berikut:

Fragmen: Sebuah fragmen dalam diagram urutan diwakili oleh sebuah kotak, dan membungkus sebagian dari interaksi dalam diagram urutan. Setiap fragmen juga dikenal sebagai operan interaksi. Operand interaksi mungkin berisi kondisi penjaga opsional, yang juga disebut batasan interaksi. Perilaku yang ditentukan dalam operan interaksi dijalankan hanya jika kondisi penjaganya bernilai benar.



Gambar 7.35 Anatomi fragmen gabungan di UML 2.0.

Operator: Fragmen gabungan dikaitkan dengan satu operator yang disebut operator interaksi yang ditampilkan di sudut kiri atas fragmen. Operator menunjukkan jenis fragmen. Jenis operator logika bersama dengan penjaga di fragmen mendefinisikan perilaku fragmen gabungan. Fragmen gabungan juga dapat berisi fragmen gabungan bersarang atau penggunaan interaksi yang berisi struktur bersyarat tambahan yang mewakili struktur yang lebih kompleks yang memengaruhi aliran pesan. Beberapa operator penting dari fragmen gabungan adalah sebagai berikut:

alt: Operator ini menunjukkan bahwa di antara beberapa fragmen, hanya satu yang penjaganya benar yang akan dieksekusi.

opt: Fragmen opsional yang akan dijalankan hanya jika penjaga benar.

par: Operator ini menunjukkan bahwa berbagai fragmen dapat dieksekusi secara bersamaan.

loop: Operator loop menunjukkan bahwa berbagai fragmen dapat dieksekusi beberapa kali dan penjaga menunjukkan dasar iterasi, yang berarti bahwa eksekusi akan berlanjut hingga penjaga menjadi salah.

region: Ini mendefinisikan wilayah kritis di mana hanya satu utas yang dapat dieksekusi.

Contoh fragmen gabungan telah ditunjukkan pada Gambar 7.36.

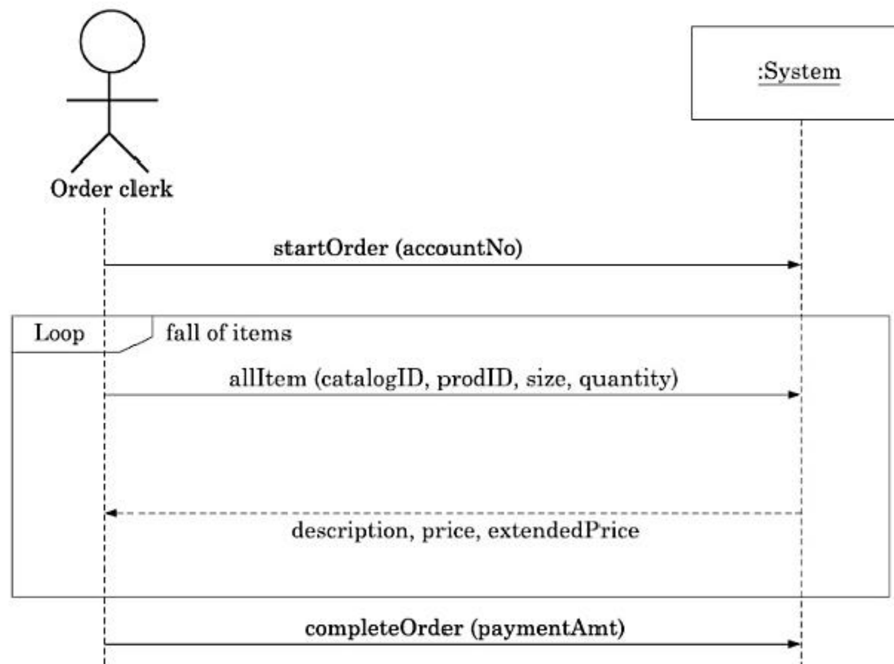
Diagram struktur komposit

Diagram struktur komposit memungkinkan Anda menentukan bagaimana kelas didefinisikan oleh struktur kelas lebih lanjut dan jalur komunikasi antara bagian-bagian ini. Beberapa konstruksi inti baru seperti bagian, port dan konektor diperkenalkan.

Part : Konsep bagian memungkinkan deskripsi struktur internal kelas.

Port: Konsep port memungkinkan untuk menggambarkan titik koneksi secara formal. Ini dapat dialamatkan, yang berarti bahwa sinyal dapat dikirim kepada mereka.

Konektor: Konektor dapat digunakan untuk menentukan hubungan komunikasi antara dua atau lebih bagian.



Gambar 7.36 Contoh diagram urutan yang menunjukkan fragmen gabungan di UML 2.0.

7.12 RINGKASAN

- Salah satu keuntungan utama dari orientasi objek adalah peningkatan produktivitas tim pengembangan perangkat lunak. Alasan mengapa proyek berorientasi objek mencapai tingkat produktivitas yang lebih tinggi secara dramatis dapat dikaitkan terutama karena penggunaan kembali kelas yang telah ditentukan dan sebagian penggunaan kembali dicapai karena pewarisan, dan kesederhanaan konseptual yang dibawa oleh pendekatan objek.
- Pemodelan objek sangat penting dalam menganalisis, merancang, dan memahami sistem. UML dengan cepat mendapatkan popularitas dan siap menjadi standar dalam pemodelan objek.
- UML dapat digunakan untuk membangun lima tampilan berbeda dari suatu sistem menggunakan sembilan jenis diagram yang berbeda. Namun, tidak wajib untuk membangun semua tampilan sistem menggunakan semua jenis diagram dalam upaya pemodelan. Jenis model yang akan dibangun tergantung pada masalah yang dihadapi.
- Sintaks dan semantik dari beberapa jenis diagram penting yang dapat dibangun menggunakan UML.

7.13 LATIHAN

1. Pilih opsi yang benar:
   a. Pengemasan data dan fungsi menjadi satu kesatuan dalam suatu program dikenal sebagai:
      i. Polimorfisme
      ii. Abstraksi
      iii. Enkapsulasi
      iv. Warisan

b. Pertimbangkan pernyataan—“Seorang karyawan adalah pekerja atau manajer.” Dengan asumsi bahwa Karyawan dan Manajer menjadi dua kelas, apa yang dapat dikatakan tentang hubungan antara dua kelas ini?
    i. Asosiasi
    ii. Generalisasi-spesialisasi
    iii. Penahanan
    iv. Polimorfisme
c. Manakah dari berikut ini yang dapat dikatakan tentang tipe data abstrak (ADT):
    i. Sama seperti kelas abstrak
    ii. Tipe data yang tidak dapat diinstansiasi
    iii. Tipe data yang hanya dapat digunakan melalui operasi yang ditentukan di atasnya
    iv. Sama seperti kumpulan item data
d. Manakah dari berikut ini yang dapat dikatakan mewakili hubungan antara kelas dan kelas induk publiknya?
    i. "...adalah..."
    ii. "...punya..."
    iii. ”...diimplementasikan sebagai...”
    iv. ”...menggunakan...”
e. Manakah dari kalimat berikut yang paling tepat menggambarkan "warisan berganda"?
    i. Di mana dua kelas mewarisi satu sama lain
    ii. Ketika kelas dasar memiliki dua atau lebih kelas turunan
    iii. Ketika kelas anak memiliki dua atau lebih kelas induk
    iv. Ketika kelas anak memiliki hubungan "adalah" dan "memiliki" kelas induknya
f. Manakah dari pewarisan sifat terbaik berikut ini?
    i. Sama dengan enkapsulasi.
    ii. Agregasi informasi.
    iii. Generalisasi dan spesialisasi.
    iv. Polimorfisme
g. Bagaimana manfaat warisan?
    i. Mencegah hilangnya properti yang diwarisi
    ii. Ini meminimalkan jumlah kode yang harus ditulis
    iii. Ini menciptakan struktur pohon yang elegan
    iv. Ini membagi objek menjadi kelas-kelas yang berguna
h. Perhatikan kalimat: Sebuah buku memiliki satu halaman atau lebih. Manakah dari konsep berikut yang paling mencirikannya?
    i. Warisan
    ii. Spesialisasi
    iii. Asosiasi
    iv. Komposisi
i. Manakah dari berikut ini yang merupakan jenis hubungan?
    i. Agregasi
    ii. Asosiasi
    iii. Ketergantungan
    iv. Warisan
j. Diagram urutan adalah:

i. Garis waktu yang menggambarkan urutan panggilan yang khas antara anggota fungsi objek
ii. Pohon panggilan yang menggambarkan semua kemungkinan urutan panggilan antara anggota fungsi kelas
iii. Garis waktu yang menggambarkan perubahan dalam pewarisan dan hubungan instantiasi antara kelas dan objek dari waktu ke waktu
iv. Pohon yang menggambarkan pewarisan dan hubungan antar kelas
v. Graf asiklik berarah yang menggambarkan hubungan pewarisan dan instantiasi antara kelas dan objek

k. Diagram UML mana yang harus Anda gunakan saat mengalokasikan perilaku use case ke kelas?
i. diagram urutan dan komunikasi
ii. use case dan diagram aktivitas
iii. diagram urutan dan aktivitas
iv. diagram kelas dan struktur komposit

l. Manakah dari berikut ini yang merupakan karakteristik dari desain berorientasi objek yang baik:
i. Hirarki kelas dalam
ii. Sejumlah besar metode per kelas
iii. Sejumlah besar pertukaran pesan per kasus penggunaan
iv. Jumlah metode sedang per kelas

m. Bagaimana perbedaan antara kelas abstrak dan kelas konkret?
i. Kelas abstrak mewakili konsep tidak berwujud dalam domain aplikasi, sedangkan kelas konkret mewakili hal-hal fisik.
ii. Kelas abstrak adalah superclass, sedangkan kelas konkret adalah subclass.
iii. Kelas abstrak tidak memiliki instance, sedangkan kelas konkret memiliki instance.
iv. Kelas abstrak adalah tipe khusus dari kelas beton.

n. Manakah dari pernyataan berikut yang salah tentang enkapsulasi?
i. Enkapsulasi membantu dalam penggunaan kembali karena developer lain tidak perlu mengetahui bagaimana komponen perangkat lunak bekerja secara internal.
ii. Enkapsulasi berarti bahwa komponen perangkat lunak dapat bekerja lebih efisien.
iii. Enkapsulasi berarti developer perangkat lunak tidak perlu mendokumentasikan pekerjaan mereka.
iv. Enkapsulasi menghalangi penggunaan kembali.

2. Dengan bantuan contoh yang sesuai, jelaskan bagaimana fitur pewarisan dari paradigma berorientasi objek membantu dalam penggunaan kembali kode?
3. Bisakah hubungan asosiasi antar kelas menjadi unary? Jika jawaban Anda "ya", berikan contoh asosiasi unary antar kelas. Jika jawaban Anda adalah "tidak", jelaskan mengapa hubungan seperti itu tidak bisa ada.
4. 4Apakah kelas merupakan tipe data abstrak (ADT)? Justifikasi jawaban Anda.
5. Berikan contoh yang berarti dari masing-masing jenis hubungan antar kelas berikut. Hanya diagram kelas dan garis penjelasan yang diperlukan dalam setiap kasus.
a. Warisan tunggal
b. Warisan berganda

   c. Asosiasi
   d. Agregasi
   e. Ketergantungan
6. Perhatikan kalimat-kalimat berikut yang diambil dari berbagai deskripsi informasi hingga masalah pengembangan perangkat lunak. Dari analisis kalimat, mengidentifikasi kelas dan hubungan di antara mereka yang dapat disimpulkan dari kalimat. Mewakili jawaban Anda menggunakan diagram kelas UML:
   a. Persegi adalah poligon
   b. Shyam adalah seorang siswa
   c. Setiap mahasiswa memiliki nama
   d. Siswa tinggal di asrama
   e. Setiap mahasiswa adalah anggota perpustakaan
   f. Seorang mahasiswa dapat memperbaharui buku yang dipinjamnya
   g. Sebuah perguruan tinggi memiliki banyak siswa
   h. Sebuah linked list terdiri dari banyak node sehingga setiap node adalah penerus dari beberapa node dan merupakan pendahulu dari beberapa node.
7. Dengan bantuan contoh yang sesuai, jelaskan bagaimana polimorfisme membantu dalam mengembangkan kode yang mudah dipelihara dan menarik secara intuitif.
8. Apa yang Anda pahami dengan metode overloading dalam konteks pemrograman berorientasi objek? Bagaimana metode overloading berguna?
9. Bisakah C++ dan Java dianggap sebagai bahasa pemrograman berorientasi objek murni? Justifikasi jawaban Anda.
10. Jelaskan konsep pengikatan dinamis seperti yang digunakan dalam bahasa berorientasi objek menggunakan contoh ilustrasi sederhana. Bagaimana pengembangan program yang mengikat dinamis bermanfaat?
11. Apa alasan di balik peningkatan produktivitas yang dicatat ketika tim pengembangan mengadopsi paradigma berorientasi objek dibandingkan dengan mengadopsi paradigma prosedural?
12. Diskusikan keuntungan dan kerugian mengadopsi gaya pengembangan perangkat lunak berorientasi objek.
13. Dalam konteks orientasi objek, bedakan antara operasi dan metode. Benarkah setiap operasi harus diimplementasikan dengan metode yang unik?
14. Apa perbedaan antara kelebihan metode dan penggantian metode? Jelaskan jawaban Anda dengan menggunakan contoh yang sesuai.
15. Warisan dan komposisi (penyematan objek) dapat dianggap serupa dalam arti bahwa keduanya memerlukan salinan komponen (dasar) untuk disematkan (ditautkan) dalam objek gabungan (turunan). Apakah mungkin menggunakan penyematan objek (yaitu, objek komposit) untuk mewujudkan fitur pewarisan dan sebaliknya? Buktikan jawaban Anda dengan menggunakan contoh-contoh yang sesuai.
16. Apa yang Anda pahami dengan enkapsulasi dan abstraksi dalam konteks orientasi objek? Bagaimana enkapsulasi berbeda dibandingkan dengan abstraksi?
17. Apa perbedaan antara kelebihan metode dan penggantian metode? Jelaskan mekanisme metode overloading dan metode overriding menggunakan contoh yang sesuai.
18. Apa saja tampilan sistem yang berbeda yang dapat dimodelkan menggunakan UML? Apa saja diagram UML yang berbeda yang dapat digunakan untuk menangkap setiap tampilan? Apakah Anda perlu mengembangkan semua tampilan sistem menggunakan semua diagram pemodelan yang didukung oleh UML? Justifikasi jawaban Anda.

19. Apa perbedaan antara use case dan skenario? Identifikasi setidaknya tiga skenario kasus penggunaan penarikan tunai di ATM bank.
20. Menurut Anda mengapa UML membutuhkan beberapa model dari perspektif yang berbeda untuk dibangun—bukankah ide yang baik untuk hanya memiliki satu model yang menangkap semua perspektif yang diperlukan?
21. Nyatakan TR UE atau FALSE dari berikut ini. Dukung jawaban Anda dengan alasan yang tepat:
    a. Desain berorientasi objek tidak dapat diimplementasikan menggunakan bahasa pemrograman prosedural.
    b. Bahasa apa pun yang secara langsung mendukung tipe data abstrak (ADT) dapat disebut sebagai bahasa berorientasi objek.
    c. Berbeda dengan kelas abstrak, kelas konkret harus mendefinisikan semua data dan metode dalam definisi kelas itu sendiri tanpa mewarisi salah satu dari mereka.
    d. Bahasa berorientasi objek dapat digunakan untuk mengimplementasikan desain erorientasi fungsi.
    e. Hubungan pewarisan menggambarkan memiliki hubungan antar kelas.
    f. Fitur warisan dari paradigma berorientasi objek membantu dalam penggunaan kembali kode.
    g. Hubungan pewarisan antara dua kelas dapat dianggap sebagai hubungan generalisasi-spesialisasi.
    h. Penyematan objek (yaitu, objek komposit) dapat digunakan untuk mewujudkan hubungan pewarisan.
    i. Hubungan agregasi antar kelas bersifat antisimetris.
    j. Hubungan agregasi dapat didefinisikan secara rekursif, yaitu suatu objek dapat berisi instance dari dirinya sendiri.
    k. State chart diagram dalam UML biasanya digunakan untuk memodelkan bagaimana beberapa perilaku dari suatu sistem diwujudkan melalui tindakan kooperatif dari beberapa objek.
    l. Multiple inheritance adalah fitur dimana banyak subclass dapat mewarisi fitur dari satu kelas dasar.
    m. Diagram kelas yang dikembangkan menggunakan UML dapat berfungsi sebagai fungsional spesifikasi suatu sistem.
    n. Keuntungan penting dari polimorfisme adalah fasilitasi penggunaan kembali.
    o. Model berorientasi objek statis harus menangkap atribut dan metode kelas dan dalam urutan apa metode yang berbeda memanggil satu sama lain.
    p. Dalam diagram kelas UML, hubungan agregasi mendefinisikan hubungan ekivalensi suatu objek.
    q. Kelas abstrak dan kelas Antarmuka (seperti yang digunakan dalam UML, Java, dll.) adalah konsep yang setara.
    r. Hubungan agregasi dapat dianggap sebagai jenis hubungan asosiasi khusus.
    s. Hubungan agregasi dapat bersifat refleksif.
    t. Hubungan agregasi tidak bisa refleksif tetapi transitif.
    u. Biasanya, Anda menggunakan satu diagram interaksi per kelas untuk mewakili bagaimana perilaku suatu objek kelas berubah selama masa hidupnya.
    v. Ada kemungkinan bahwa lebih dari satu metode dalam satu kelas mengimplementasikan operasi yang sama.

   w. Urutan kronologis pesan dalam diagram interaksi tidak dapat ditentukan dari pemeriksaan diagram.
   x. Diagram interaksi dapat digunakan secara efektif untuk menggambarkan bagaimana perilaku suatu objek berubah di seluruh eksekusi beberapa kasus penggunaan.
   y. Dari diagram urutan UML untuk use case, dimungkinkan untuk menyimpulkan berbagai skenario dalam use case.
22. Nyatakan BENAR atau SALAH dari berikut ini. Dukung jawaban Anda dengan alasan yang tepat:
    a. Program berorientasi objek yang tidak menggunakan mekanisme pewarisan dalam definisi kelas, tidak dapat menampilkan pengikatan dinamis.
    b. Mekanisme pewarisan dapat dianggap menyediakan abstraksi fitur.
    c. Diagram bagan keadaan berguna untuk menggambarkan perilaku yang melibatkan banyak objek yang bekerja sama satu sama lain untuk mencapai beberapa perilaku.
    d. Implementasi use case dalam hal pemanggilan metode tertentu digambarkan dalam diagram urutan.
    e. Istilah metode dan operasi adalah konsep yang setara dan dapat digunakan secara bergantian.
    f. Upaya untuk menguji dan men-debug program berorientasi objek dapat dikurangi dengan mengurangi jumlah pertukaran pesan antar objek.
23. Apa yang Anda pahami dengan istilah enkapsulasi dalam konteks desain perangkat lunak? Apa keuntungan dari enkapsulasi?
24. Apa yang Anda pahami tentang abstraksi data? Bagaimana abstraksi data membantu mengurangi kopling dalam solusi desain?
25. Apa jenis hubungan berbeda yang mungkin ada di antara kelas-kelas dalam desain berorientasi objek? Berikan contoh masing-masing.
26. Apa yang Anda pahami tentang hubungan asosiasi antar kelas. Berikan contoh kehidupan nyata atau pemrograman dari asosiasi unary, biner, dan ternary antar kelas.
27. Apakah yang Anda maksud: kelas abstrak Berikan contoh kelas abstrak. Kelas abstrak tidak dapat memiliki instance. Lalu apa gunanya mendefinisikan kelas abstrak?
28. Apa saja jenis model masalah yang berbeda yang dapat dibangun menggunakan UML? Mengapa perlu membangun lebih dari satu jenis model masalah?
    a. Tunjukkan perbedaan utama antara bahasa berorientasi objek (misalnya, C++) dan bahasa prosedural (misalnya, C).
    b. Dapatkah desain berorientasi objek diimplementasikan menggunakan bahasa prosedural? Bisakah desain berorientasi fungsi tradisional diimplementasikan menggunakan bahasa berorientasi objek? Tulis alasan di balik jawaban Anda.
29. Fitur dasar apa yang perlu didukung oleh bahasa pemrograman untuk disebut sebagai bahasa berorientasi objek? Bagaimana bahasa pemrograman berorientasi objek berbeda dari bahasa pemrograman prosedural tradisional seperti C atau PASCAL?
30. Apa perbedaan antara use case dan skenario? Identifikasi semua skenario kasus penggunaan penarikan tunai dari ATM bank standar.
31. Apa perbedaan antara diagram urutan dan diagram kolaborasi? Dalam konteks apa Anda akan menggunakan masing-masing?
32. Apa itu stereotip dalam UML? Jelaskan dengan beberapa contoh situasi di mana ini dapat digunakan?

33. Bagaimana Anda bisa menentukan batasan yang berbeda pada elemen pemodelan di UML? Misalnya, bagaimana Anda bisa menentukan bahwa semua buku disimpan menurut abjad di perpustakaan?
34. Apa perbedaan antara model statis dan dinamis dalam konteks pemodelan sistem berorientasi objek? Identifikasi diagram UML yang masing-masing menyediakan kedua model ini.
35. Dalam pemodelan sistem menggunakan UML, bagaimana klasifikasi pengguna sistem ke dalam berbagai jenis aktor dan representasi mereka dalam diagram use case membantu dalam pengembangan sistem?
36. Gambarlah diagram kelas menggunakan sintaks UML untuk mewakili fakta bahwa suatu orderRegister terdiri dari banyak pesanan. Setiap pesanan terdiri dari hingga sepuluh item pesanan. Setiap item pesanan berisi nama item, jumlah dan tanggal yang dibutuhkan. Setiap item pesanan dijelaskan oleh objek spesifikasi pesanan item yang memiliki rincian item pesanan seperti harga satuan, nama dan alamat produsen, serta masa garansi dan syarat garansi.
37. Gambarlah diagram kelas menggunakan sintaks UML untuk mewakili aspek-aspek berikut mengenai perpustakaan. Isu dapat berupa buku atau CD. Buku dapat berupa buku referensi atau buku teks. Rincian berbagai penerbitan disimpan dalam register yang disebut register yang dapat diterbitkan. Perpustakaan memiliki banyak anggota yang rinciannya disimpan dalam daftar anggota. Seorang anggota dapat menerbitkan hingga 10 buku teks selama sebulan. Seorang anggota juga dapat mengeluarkan dua CD selama seminggu.
38. Gambarlah diagram kelas menggunakan sintaks UML untuk merepresentasikan fakta bahwa armada kendaraan di biro perjalanan terdiri dari kendaraan jenis Tata Indica, Maruti van, dan Mahindra Xylo. Pelanggan tetap agen perjalanan dapat menyewa kendaraan apa pun yang mereka inginkan. Rincian pelanggan seperti nama, alamat, dan nomor telepon dikelola oleh agensi.
39. Gambarlah diagram kelas menggunakan sintaks UML untuk mewakili fakta bahwa register buku perpustakaan berisi rincian semua buku di perpustakaan. Rincian untuk setiap buku meliputi judul, penulis, nomor ISBN, harga, tanggal pengadaan, harga, dan tanggal pinjaman terakhir, orang yang dipinjamkan. Sebuah buku dapat berupa buku referensi atau jenis terbitan. Buku-buku referensi harus dirujuk di dalam perpustakaan dan tidak dapat dipinjamkan, sedangkan buku-buku terbitan dapat diambil dengan pinjaman oleh anggota. Daftar anggota berisi rincian semua anggota perpustakaan. Rincian yang disimpan untuk anggota termasuk nama anggota, alamat, nomor telepon, tanggal bergabung dengan perpustakaan dan buku-buku yang beredar. Setiap anggota perpustakaan dapat meminjam paling banyak lima buku terbitan.
40. Gambarlah diagram kelas menggunakan sintaks UML untuk mewakili berikut ini. Sebuah perguruan tinggi teknik menawarkan gelar B.Tech di tiga cabang — Elektronik, Listrik, dan Ilmu dan teknik komputer. Setiap cabang dapat menerima 30 mahasiswa setiap tahun. Bagi seorang mahasiswa untuk menyelesaikan, gelar B.Tech dia harus menyelesaikan semua 30 mata kuliah inti dan setidaknya 10 mata kuliah pilihan.
41. Jelaskan secara singkat bagaimana prinsip-prinsip dekomposisi dan abstraksi digunakan dalam paradigma berorientasi objek.
42. Bagaimana diagram aktivitas berguna selama pengembangan sistem? Apa cara penting di mana diagram aktivitas berbeda dari diagram alir?
43. Sebutkan kekurangan penting dari UML 1.X. Bagaimana UML 2.0 mengatasi ini?

44. Mengembangkan model kasus penggunaan untuk perangkat lunak pengolah kata seperti MSWORD.
45. Mengembangkan model use case untuk ATM bank standar.

BAB 8
PENGEMBANGAN PERANGKAT LUNAK BERORIENTASI OBYEK

Dalam Bab ini, kita akan membangun konsep pemodelan objek yang diperkenalkan pada Bab terakhir untuk membahas metodologi analisis dan desain berorientasi objek (OOAD). Kita akan tahu bahwa teknik analisis dan desain berorientasi objek (OOAD) menganjurkan pendekatan yang sangat berbeda dibandingkan dengan pendekatan desain berorientasi fungsi tradisional. Ingat diskusi di Bab 5 dan 6, di mana pendekatan desain berorientasi fungsi tradisional pada dasarnya menyarankan bahwa saat mengembangkan sistem, semua fungsi yang perlu didukung sistem harus diidentifikasi dan diimplementasikan. Sebaliknya, paradigma OOAD menyarankan bahwa objek (yaitu, entitas) yang terkait dengan masalah harus diidentifikasi dan diimplementasikan.

Dalam pendekatan desain berorientasi fungsi, cara sederhana untuk mengidentifikasi fungsi yang dilakukan oleh sistem adalah dengan memeriksa semua kata kerja yang muncul dalam deskripsi masalah—karena kata kerja (seperti create, edit, search, dll.) mewakili aktivitas (atau fungsi).) yang dilakukan oleh suatu sistem. Analisis kata kerja, pada kenyataannya, merupakan cara yang efektif untuk mengidentifikasi fungsi dari suatu sistem. Di sisi lain, entitas (atau objek) yang terjadi dalam suatu masalah (seperti buku, anggota, register, dll. dalam perangkat lunak otomasi perpustakaan) dapat diidentifikasi dengan memeriksa kata benda yang muncul dalam deskripsi masalah. Grady Booch menyimpulkan perbedaan mendasar antara pendekatan desain berorientasi fungsi dan berorientasi objek [1991] dengan mengatakan:

... baca spesifikasi perangkat lunak yang ingin Anda buat. Garis bawahi kata kerja jika Anda mencari kode prosedur, kata benda jika Anda bertujuan untuk program berorientasi objek.

Sejak awal tahun sembilan puluhan, sejumlah besar teknik analisis dan desain berorientasi objek (OOAD) telah diusulkan oleh para peneliti. Dari semua ini, yang paling menonjol mungkin adalah pendekatan Booch oleh Grady Booch [1991], teknik pemodelan objek (OMT) oleh Rumbaugh dan Blaha et al. [1991], Object-Oriented Analysis (OOA) oleh Coad dan Yourdon[1991], dan Objectory oleh Jacobson [1999]. Semua teknik OOAD ini, selain mengidentifikasi objek berbeda yang diperlukan untuk mengimplementasikan sistem, juga mendesain detail internal objek. Selanjutnya, hubungan yang ada di antara objek yang berbeda diidentifikasi dan direpresentasikan dalam model desain, sehingga menjadi mudah untuk mengkodekan desain menggunakan bahasa pemrograman. Semua metodologi OOAD pada dasarnya dimulai dengan melakukan analisis berorientasi objek (OOA) untuk mengembangkan model analisis terlebih dahulu dan kemudian menyempurnakannya menjadi model desain.

Analisis berorientasi objek (OOA) versus desain berorientasi objek (OOD)

Sebelum membahas detail OOA dan OOD, mari kita pahami perbedaan utama antara intent dan aktivitas yang dilakukan. Istilah analisis berorientasi objek (OOA) mengacu pada pengembangan model awal produk perangkat lunak dari analisis spesifikasi kebutuhannya. Analisis melibatkan membangun model (disebut model analisis) dengan menganalisis dan mengelaborasi kebutuhan pengguna yang didokumentasikan dalam dokumen SRS daripada menentukan bagaimana mendefinisikan solusi yang dapat dengan mudah diimplementasikan. Saat mengembangkan model analisis, keputusan spesifik implementasi (seperti dalam urutan apa kelas akan memanggil metode satu sama lain, perangkat keras tertentu yang digunakan, basis data yang digunakan, dll.) dihindari. Oleh karena itu, model analisis tetap valid, meskipun

aspek implementasinya berubah di kemudian hari. Namun, sangat sulit untuk secara langsung menerjemahkan model analisis ke dalam kode. Di sisi lain, model desain dapat dengan mudah diterjemahkan ke dalam kode. Saat ini, banyak alat rekayasa perangkat lunak berbantuan komputer (CASE) mendukung pembuatan templat kode secara otomatis dari model desain, sehingga sangat mengurangi pekerjaan pemrogram.

Perbedaan antara model analisis dan model desain disini akan dijelaskan melalui contoh sederhana. Pertimbangkan bahwa salah satu fungsi perangkat lunak otomatisasi trade-house adalah "menampilkan statistik penjualan", maka model analisis akan melibatkan representasi berbagai konsep yang dapat diidentifikasi dalam deskripsi fungsi ini seperti data input, data output, dan pemrosesan yang diperlukan. Di sisi lain, solusi desain harus membahas bagaimana tepatnya (dengan kelas dan metode interaksi mana) statistik penjualan akan dihitung.

Dalam pendekatan prosedural, perbedaan mencolok dengan mudah terlihat antara kegiatan analisis dan desain. Kegiatan analisis berkaitan dengan pengembangan model aliran data menggunakan notasi DFD, sedangkan kegiatan desain berkaitan dengan pengembangan desain menggunakan notasi bagan struktur. Bahkan notasi yang digunakan untuk mendokumentasikan masing-masing model1 berbeda. Sebaliknya, dalam pendekatan berorientasi objek, transisi dari aktivitas analisis ke aktivitas desain terjadi secara bertahap dan tidak ada batasan yang jelas antara kapan aktivitas analisis berakhir dan aktivitas desain dimulai. Juga, notasi identik digunakan untuk mendokumentasikan hasil analisis dan desain.

Metodologi OOAD

Dalam bab ini, kita akan membahas metodologi umum untuk mengembangkan desain berorientasi objek mulai dari deskripsi masalah awal. Metodologi ini terdiri dari pertama membangun model use case dari analisis masalah awal. Selanjutnya, model domain dibangun. Kedua model analisis ini secara iteratif disempurnakan menjadi model desain. Model desain dapat langsung diimplementasikan menggunakan bahasa pemrograman. Akan tetapi, harus diingat bahwa meskipun metodologi desain yang akan kita bahas mudah dikuasai, hal ini hanya berguna untuk memecahkan masalah sederhana. Namun, begitu kita dapat memahami metode desain sederhana ini, pendekatan untuk memecahkan masalah yang lebih kompleks dapat dipahami hanya dengan upaya tambahan. Bab ini disusun sebagai berikut. Pertama-tama kita akan membahas pola desain, kemudian mengenal analisis berorientasi objek dan metodologi desain yang sebagian besar didasarkan pada proses terpadu [Jacobson 1999] dan karya Rosenberg [2000]. Selanjutnya, mengilustrasikan cara kerja metodologi yang dibahas melalui beberapa contoh.

8.1 POLA

Konsep pola pertama kali berasal dari bidang Arsitektur, di mana bangunan besar dirancang dalam solusi pola tertentu. Konsep ini sekarang telah diserap di bidang desain berorientasi objek untuk menyediakan mekanisme yang sangat kuat untuk penggunaan kembali desain.

Pola desain dan perannya dalam OOAD

Saat mengerjakan solusi desain untuk suatu masalah, desainer berpengalaman secara sadar atau tidak sadar menggunakan kembali solusi yang mungkin telah mereka kerjakan di masa lalu. Penggunaan kembali solusi desain tersebut disistematisasikan oleh konsep pola. Faktanya, pola memungkinkan solusi yang diterima secara umum untuk digunakan kembali oleh semua orang yang akrab dengan pola. Kita akan segera melihat bahwa saat mengerjakan solusi desain untuk suatu masalah, pengetahuan tentang beberapa pola desain penting dapat

sangat membantu meningkatkan kualitas desain dan pada saat yang sama mengurangi upaya total. Tapi, apa itu pola desain?

Pola desain adalah solusi yang diterima secara umum untuk beberapa masalah yang berulang selama merancang aplikasi yang berbeda. Pola desain saat ini digunakan secara luas dan telah ditemukan untuk membuat proses desain menjadi efisien. Penggunaan pola mengurangi jumlah iterasi desain, dan pada saat yang sama meningkatkan kualitas solusi desain akhir. Hal ini membuat studi menyeluruh tentang pola berharga. Setelah kita terbiasa dengan beberapa pola penting, kita dapat menemukannya dalam desain yang kita coba kerjakan dan akan dapat menggunakan kembali solusi pola.

Konsep Pola Dasar

Setiap masalah non-trivial terdiri dari sejumlah besar submasalah. Memecahkan masalah karena itu melibatkan pemecahan semua submasalahnya. Bahkan, saat memecahkan dua masalah, beberapa submasalah yang umum di antara keduanya dapat diidentifikasi. Selanjutnya, beberapa submasalah berulang di sejumlah besar masalah. Ini membuka kemungkinan untuk mencapai solusi yang baik dengan upaya yang dikurangi dengan menguasai solusi yang diterima secara umum untuk beberapa submasalah penting yang berulang di berbagai masalah. Ini sebenarnya, menangkap ide sentral di balik pola.

Dalam konteks desain perangkat lunak, pola mendokumentasikan solusi untuk masalah tertentu yang dapat digunakan kembali selama desain aplikasi yang berbeda. Saat mengerjakan solusi untuk masalah desain, setelah mengidentifikasi pola, kita dapat langsung menggunakan solusi pola yang terdokumentasi, jika kita sudah memahaminya dengan baik. Kita sekarang dapat menyatakan ide dasar di balik pola sebagai berikut:

Ide dasar di balik pola adalah bahwa jika Anda dapat menguasai beberapa pola penting, Anda dapat dengan mudah menemukannya dalam masalah pengembangan aplikasi dan dengan mudah menggunakan solusi pola.

Solusi pola tidak benar-benar mendukung ide desain revolusioner, tetapi dibuat berdasarkan akal sehat dan penerapan prinsip-prinsip desain mendasar. Anda mungkin kemudian bertanya-tanya, “Jika pola benar-benar didasarkan pada akal sehat, lalu apa gunanya pola belajar? Bukankah seseorang secara mekanis akan sampai pada solusi serupa dengan cara apa pun?” Pada kenyataannya, diamati bahwa sementara bergulat dengan seluk beluk masalah desain yang kompleks, desainer sering melupakan akal sehat — dan cenderung membuat solusi desain mereka menjadi rumit, tidak fleksibel, dan tidak efisien. Di sisi lain, jika Anda terbiasa dengan beberapa pola penting, Anda dapat dengan mudah menemukannya

dan secara mekanis menggunakan solusi pola dalam desain Anda. Dengan demikian, pola berfungsi sebagai panduan untuk membuat desain yang “baik”. Juga, penggunaan pola secara signifikan meningkatkan produktivitas desainer, dengan mengurangi iterasi desain dan pada saat yang sama meningkatkan kualitas desain akhir. Mengingat kelebihan yang melekat, desainer berpengalaman biasanya membiasakan diri dengan ratusan pola. Pola dapat dilihat sebagai membantu desainer untuk membuat keputusan desain penting tertentu. Pada tingkat dasar, pola juga dapat dilihat sebagai blok bangunan yang terdokumentasi dengan baik dan dipikirkan dengan baik untuk desain perangkat lunak.

Tapi, siapa yang menciptakan pola? Pola dibuat dan didokumentasikan oleh orang-orang yang melihat tema berulang di seluruh desain. Setelah pola dibuat dan didokumentasikan, mereka dapat digunakan oleh desainer yang berbeda. Oleh karena itu, kita dapat mempertimbangkan pola sebagai sarana yang sangat efektif untuk menangkap dan mentransfer pengetahuan desain di berbagai masalah desain. Pola juga telah membentuk kosakata standar yang digunakan untuk mengkomunikasikan ide-ide desain dalam lingkungan profesional.

Selain memberikan solusi model, pola mendokumentasikan spesifikasi masalah yang jelas, dan juga menjelaskan keadaan di mana solusi akan berhasil dan tidak akan berhasil. Sebuah dokumentasi pola biasanya terdiri dari empat bagian penting:

- Masalah.
- Konteks di mana masalah itu terjadi.
- Solusinya.
- Konteks di mana solusi akan berhasil dan tidak akan berhasil.

Jenis Pola

Berbagai jenis pola telah diidentifikasi untuk digunakan dalam berbagai tahap desain. Mulai dari penggunaan dalam desain tingkat sangat tinggi yang disebut sebagai desain arsitektur, solusi pola telah didefinisikan untuk digunakan dalam desain beton dan bahkan dalam kode. Beberapa konsep dasar tentang ketiga jenis pola ini sebagai berikut.

Pola arsitektur: Pola arsitektur mengidentifikasi dan memberikan solusi untuk masalah yang dapat diidentifikasi saat melakukan desain arsitektur (atau tingkat sangat tinggi). Desain arsitektural menyangkut keseluruhan struktur sistem perangkat lunak. Desain arsitektur tidak dapat langsung diterjemahkan ke dalam kode, tetapi menjadi dasar untuk desain yang lebih detail. Setiap pola arsitektur menyarankan satu set subsistem yang telah ditentukan, menentukan tanggung jawab mereka, dan termasuk aturan dan pedoman untuk mengatur hubungan di antara mereka.

Desain arsitektur biasanya dibangun untuk masalah yang sangat besar. Oleh karena itu, tentu saja, pola arsitektur relevan saat mengerjakan solusi tingkat tinggi untuk masalah yang sangat besar. Pola desain: Pola desain biasanya menyarankan skema untuk menyusun kelas dalam solusi desain dan mendefinisikan interaksi yang diperlukan di antara kelas-kelas tersebut. Dengan kata lain, pola desain menggambarkan beberapa struktur kelas komunikasi yang berulang yang dapat digunakan untuk memecahkan beberapa masalah desain umum. Solusi pola desain biasanya dijelaskan dalam hal kelas, contoh mereka, peran dan kolaborasi mereka.

Idiom: Idiom adalah seperangkat pola tingkat rendah yang spesifik bahasa pemrograman. Sebuah idiom menjelaskan bagaimana menerapkan solusi untuk masalah tertentu menggunakan fitur dari bahasa pemrograman yang diberikan.

Dalam bahasa alami seperti bahasa Inggris, idiom berarti sekelompok kata yang bersama-sama memiliki arti yang berbeda dari yang diperoleh dengan penjajaran sederhana definisi kamus dari kata-kata individu. Contoh idiom bahasa Inggris adalah “hujan kucing dan anjing”, “selemparan batu”, dll. Idiom membantu menulis potongan prosa yang bagus dengan upaya yang jauh berkurang. Saat menulis prosa, para ahli bahasa dengan mudah memasukkan idiom ke dalam tulisan mereka untuk meningkatkan kualitas prosa dan pada saat yang sama mengurangi waktu dan upaya yang diperlukan untuk menulis prosa. Ketika seorang ahli bahasa ingin mengatakan bahwa hujan turun sangat deras, dia hampir secara mekanis menulis "Hujan kucing dan anjing." tanpa benar-benar harus memikirkan kombinasi kata tertentu untuk digunakan: seperti apakah akan mengatakan "Ini hujan tikus dan gajah", dll. Demikian pula untuk mengakses elemen yang berbeda dari array ukuran 100, programmer C akan menulis segmen kode : for(i=0;i<100;i++){...} tanpa harus berpikir dan memutuskan apakah akan menggunakan perulangan while, apakah akan memulai perulangan dengan variabel i yang diinisialisasi ke 1, dll. Programmer yang baik tahu beberapa idiom. Segera setelah mereka melihat persyaratan saat mengerjakan solusi pemrograman, idiom yang diperlukan muncul pada mereka secara instan dan membebaskan mereka dari keharusan berjuang untuk memilih konstruksi yang tepat untuk digunakan dalam program dan pada saat yang sama meningkatkan kualitas program. dan mengurangi deteksi bug dan iterasi koreksi.

Perbandingan berbagai jenis pola

Perbedaan utama di antara tiga jenis pola yang dibahas di atas terletak pada tingkat abstraksi dan detail yang mereka tangani. Pola arsitektur adalah strategi tingkat tinggi yang menyangkut solusi keseluruhan untuk masalah skala besar. Pola desain adalah solusi untuk bagian tertentu dari masalah skala menengah dan merekomendasikan struktur dan perilaku tertentu dari entitas yang berpartisipasi. Idiom adalah solusi pemrograman khusus paradigma dan bahasa yang merekomendasikan penggunaan segmen kode yang sesuai untuk memecahkan masalah pemrograman tingkat rendah.

Lebih Banyak Konsep Pola

Beberapa konsep pola penting lainnya adalah sebagai berikut:

Pola versus algoritma

Pemula sering bingung antara pola dan algoritme dan mengajukan pertanyaan seperti—"Apakah pola dan algoritme merupakan konsep yang identik? Bagaimanapun, keduanya menargetkan untuk memberikan solusi yang dapat digunakan kembali untuk masalah!" Faktanya, pola dan algoritma dalam beberapa hal serupa karena keduanya berusaha memberikan solusi yang dapat digunakan kembali. Namun, algoritme terutama berfokus pada pemecahan masalah dengan pengurangan ruang dan/atau persyaratan waktu, sedangkan pola fokus pada pemahaman dan pemeliharaan desain dan pengembangan yang lebih mudah. Berbeda dengan algoritma, pola lebih memperhatikan aspek-aspek seperti pemeliharaan dan kemudahan pengembangan daripada efisiensi ruang dan waktu.

Pro dan kontra dari pola desain

Sebelum menggunakan pola desain selama OOAD, disarankan untuk mengetahui kelebihan dan kekurangan pola desain.

Berikut ini adalah kelebihan utama dari pola desain:

- Pola desain memberikan kosakata umum yang membantu meningkatkan komunikasi di antara para pengembang.
- Pola desain membantu menangkap dan menyebarkan pengetahuan ahli.
- Penggunaan pola desain membantu desainer untuk menghasilkan desain yang fleksibel, efisien, dan mudah dirawat.
- Pola desain memandu developer untuk sampai pada keputusan desain yang benar dan membantu mereka meningkatkan kualitas desain mereka.
- Pola desain mengurangi jumlah iterasi desain, dan membantu meningkatkan produktivitas desainer.

Kekurangan penting dari pola desain adalah sebagai berikut:

- Pola desain tidak secara langsung mengarah pada penggunaan kembali kode. Karena pola desain disesuaikan untuk keadaan penggunaan kembali tertentu, dan oleh karena itu sulit untuk mengaitkan segmen kode tetap dengan suatu pola.
- Saat ini tidak ada metodologi yang tersedia yang dapat digunakan untuk memilih pola desain yang tepat pada titik yang tepat selama latihan desain.

Antipola

Jika suatu pola mewakili praktik terbaik, maka antipola mewakili pelajaran yang dipetik dari desain yang buruk. Berikut ini adalah dua jenis antipattern yang populer:

- Mereka yang menggambarkan solusi buruk untuk masalah, sehingga mengarah ke situasi yang buruk.
- Mereka yang menjelaskan bagaimana menghindari solusi buruk untuk masalah.

Antipattern sangat berharga karena membantu kita mengenali mengapa alternatif desain tertentu pada awalnya tampak seperti solusi yang menarik, tetapi kemudian mengarah pada kerumitan dan akhirnya berubah menjadi solusi yang buruk. Setelah kita terbiasa dengan

antipola yang penting, kita dapat secara sadar mencoba menghindarinya saat memecahkan masalah.

- **Input kludge**: Ini menyangkut kegagalan untuk menentukan dan menerapkan mekanisme untuk menangani input yang tidak valid.
- **Tombol tekan ajaib**: Ini menyangkut pengkodean logika implementasi langsung di dalam kode antarmuka pengguna, daripada melakukannya di kelas yang terpisah.
- **Bahaya ras**: Ini menyangkut kegagalan untuk melihat konsekuensi dari semua urutan kejadian yang berbeda yang mungkin terjadi dalam praktik.

8.2 BEBERAPA POLA DESAIN UMUM

Penting untuk mengenal setidaknya pola-pola penting, karena setelah Anda memahami pola-pola ini dengan baik, Anda harus dapat menemukan pola-pola ini saat memecahkan masalah dan kemudian Anda akan dapat dengan mudah menggunakan solusi yang diterima secara umum yang ditangkap oleh pola-pola tersebut. Selain itu, keakraban dengan terminologi pola akan membantu Anda menguasai kosakata yang diperlukan untuk mendiskusikan solusi pola alternatif dengan rekan kerja Anda saat mengerjakan solusi desain. Misalnya, kolega Anda mungkin saat meninjau desain Anda menyarankan cara untuk meningkatkan desain Anda dengan mengatakan "Mengapa tidak menggunakan pola MVC di bagian desain Anda yang berkaitan dengan penanganan input pengguna?"

Harus diingat bahwa pola menawarkan solusi umum yang biasanya perlu disesuaikan dalam konteks masalah tertentu. Kita beebas dalam mendiskusikan pola desain dalam konteks solusi untuk masalah desain tertentu untuk kemudahan pemahaman. Meskipun sejumlah besar pola desain telah diusulkan sejauh ini, yang oleh Larman dan geng empat atau GoF (diusulkan oleh Erich Gamma, Richard Helm, Ralph Johnson dan John Vlissides) sangat populer.

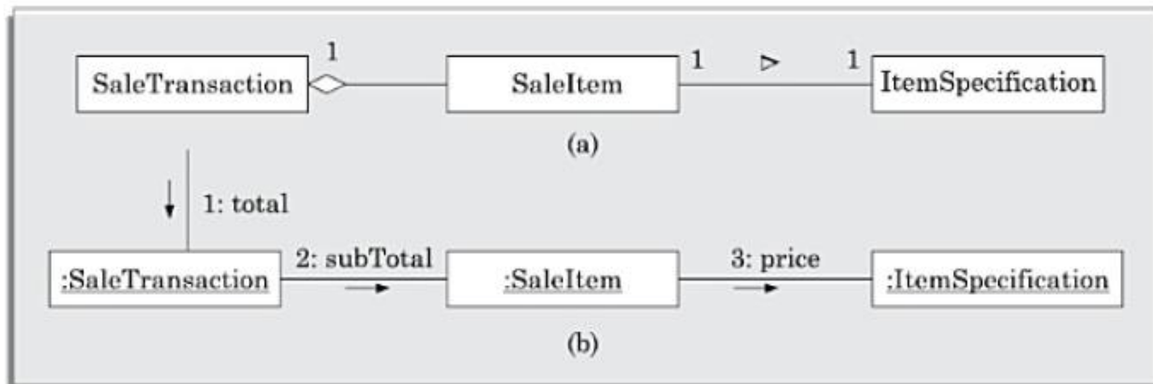
8.3 AHLI

Masalah: Ketika aktivitas tertentu perlu dilakukan, kelas mana yang harus bertanggung jawab untuk melakukannya?

Solusi: Tetapkan tanggung jawab kepada pakar informasi—kelas yang memiliki semua (atau sebagian besar) informasi yang diperlukan untuk memenuhi tanggung jawab yang diperlukan. Pola pakar mengungkapkan intuisi umum bahwa objek harus melakukan hal-hal yang berkaitan dengan informasi yang mereka simpan. Diagram kelas dan kolaborasi untuk solusi masalah kelas mana yang harus menghitung total biaya transaksi penjualan ditunjukkan pada Gambar 8.1.

Penjelasan: Pada contoh yang diberikan pada Gambar 8.1, transaksi penjualan terdiri dari banyak item penjualan. Setiap objek saleItem dikaitkan dengan objek itemSpecification. Objek itemSpecification antara lain menunjukkan harga satuan untuk item tersebut. Dalam situasi ini, objek mana yang harus menghitung harga total untuk transaksi penjualan? Mari kita pertimbangkan berbagai opsi yang tersedia. Haruskah objek *saleTransaction*, objek *saleItem*, atau objek itemSpecification diberi tanggung jawab untuk menghitung harga transaksi penjualan? Jika kita menetapkan tanggung jawab ke objek saleItem untuk menghitung harga transaksi, maka diperlukan beberapa informasi dari objek *saleTransaction* seperti jumlah item yang terjual untuk berbagai item dan bisa juga membutuhkan harga untuk item lain dari objek itemSpesifikasi yang sesuai untuk perhitungan ini. Ini akan membutuhkan sejumlah besar pertukaran data di antara objek untuk terjadi, dan mengarah pada solusi kualitas yang buruk. Demikian pula, itemSpecificatio akan menjadi pilihan yang buruk untuk menghitung total biaya yang harus dibayar untuk transaksi penjualan, karena objek tersebut

tidak memiliki sebagian besar informasi yang diperlukan untuk menghitung total biaya transaksi. Dapat dengan mudah dilihat bahwa objek saleTransaction memiliki sebagian besar informasi yang diperlukan untuk penghitungan, dan harus diberi tanggung jawab untuk menghitung total harga. Ini adalah ahli informasi, sejauh menyangkut item dan jumlah yang tepat yang dijual.



Gambar 8.1 Pola pakar: (a) Diagram kelas (b) Diagram kolaborasi

8.4 KREATOR

Masalah: Kelas mana yang harus bertanggung jawab untuk membuat instance baru dari kelas tertentu?

Solusi: Tetapkan tanggung jawab kelas C1 untuk membuat instance kelas C2, jika satu atau lebih hal berikut ini benar:

- C1 adalah agregator objek tipe C2.
- C1 berisi objek bertipe C2.
- C1 erat menggunakan objek tipe C2.
- C1 memiliki data yang diperlukan untuk menginisialisasi objek bertipe
- C2, ketika mereka dibuat.

Penjelasan: Objek agregator perlu membuat semua objek komponennya, karena agregator perlu mempertahankan alamat objek komponennya. Objek lain yang memerlukan beberapa informasi dari suatu komponen harus meminta agregator untuk mengambil informasi tersebut. Kadang-kadang, dalam masalah pemrograman berorientasi objek, objek mungkin perlu dibuat yang bukan merupakan bagian dari objek agregat apa pun. Dalam hal ini, objek harus dibuat oleh objek yang memiliki parameter inisialisasi yang diperlukan, atau objek yang paling cocok dengan objek tersebut.

Pola fasad

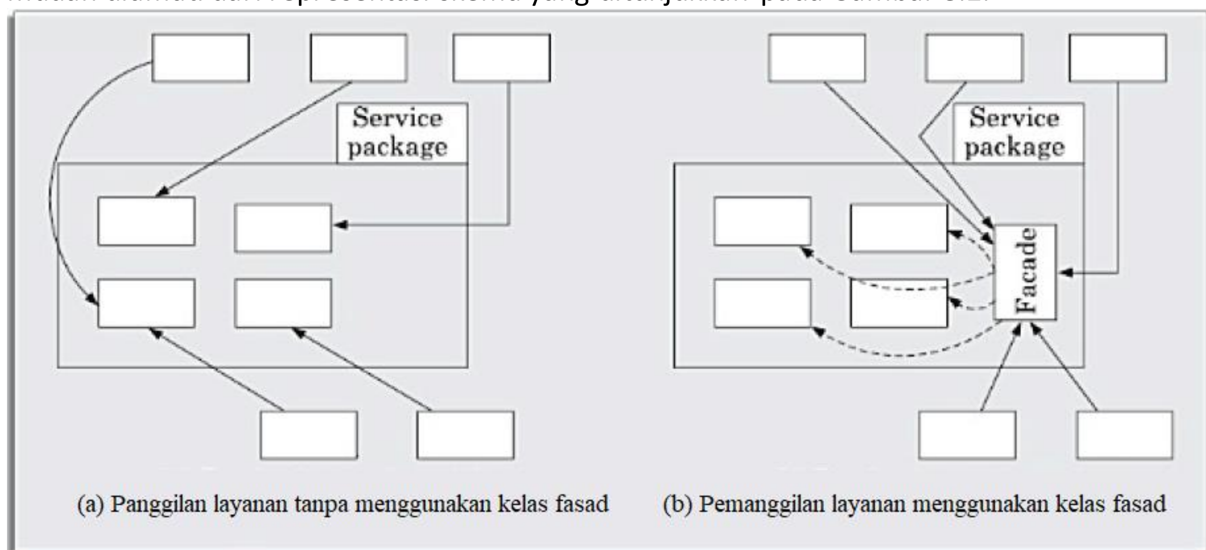
Masalah: Bagaimana seharusnya layanan diminta dari paket layanan oleh kelas klien (yaitu, kelas di luar paket)?

Konteks di mana masalah terjadi: Sebuah paket, seperti yang telah dibahas pada Bab 7, adalah sekumpulan kelas yang kohesif. Artinya, kelas-kelas dalam sebuah paket memiliki tanggung jawab yang sangat terkait. Misalnya, paket antarmuka RDBMS akan berisi kelas untuk menyediakan layanan yang dapat dipanggil oleh programmer aplikasi untuk melakukan berbagai operasi pada RDBMS.

Solusi: Kelas terpisah (seperti DBfacade) harus dibuat untuk menyediakan antarmuka umum ke layanan yang disediakan oleh kelas-kelas dalam paket.

Penjelasan: Mari kita memahami kebutuhan dan pentingnya kelas fasad menggunakan analogi sederhana. Pertimbangkan kompleks ruang kuliah. Misalkan semua mahasiswa diwajibkan masuk ke kompleks ruang kuliah melalui pintu masuk utama saja. Dalam hal ini,

ketika beberapa informasi umum seperti pembatalan beberapa kuliah pada tanggal tertentu harus diberitahukan kepada semua siswa, menampilkan satu pemberitahuan di pintu masuk sudah cukup. Namun, pertimbangkan situasi di mana mahasiswa memasuki kompleks ruang kuliah melalui beberapa pintu masuk, dan bahkan diizinkan untuk melompat ke ruang kelas melalui jendela yang berbeda dari berbagai ruang kelas. Dalam hal ini, pemberitahuan harus dipajang di semua titik masuk mahasiswa yang memungkinkan. Demikian pula, dalam konteks paket layanan, ketika (tipe atau jumlah) parameter dari beberapa metode kelas dalam paket berubah, semua kelas yang memanggil metode harus diubah. Namun, ketika objek luar (klien) memanggil layanan melalui kelas fasad, perubahan dapat diserap di kelas fasad dan perubahan pada kelas klien dapat dihindari. Juga, penggunaan kelas fasad mengurangi kompleksitas pemanggilan layanan, karena kelas klien tidak perlu mengetahui kelas khusus yang mengimplementasikan layanan. Keuntungan yang dibahas dari kelas fasad dapat dengan mudah diamati dari representasi skema yang ditunjukkan pada Gambar 8.2.



Gambar 8.2 Pemanggilan layanan dengan dan tanpa menggunakan kelas fasad

8.5 POLA PEMISAHAN TAMPILAN MODEL

Masalah: Bagaimana seharusnya kelas non-GUI berkomunikasi dengan kelas GUI dan sebaliknya?

Konteks di mana masalah terjadi: Ini adalah pola yang sangat umum terjadi yang dapat diidentifikasi di hampir setiap masalah desain. Di sini, view adalah sinonim untuk objek lapisan presentasi (atau objek GUI). dan model adalah sinonim untuk objek lapisan domain (non-GUI), Objek lapisan domain bertanggung jawab untuk benar-benar menyediakan layanan yang diperlukan, sedangkan objek lapisan presentasi hanya bertanggung jawab untuk menangani interaksi (input/output) dengan pengguna.

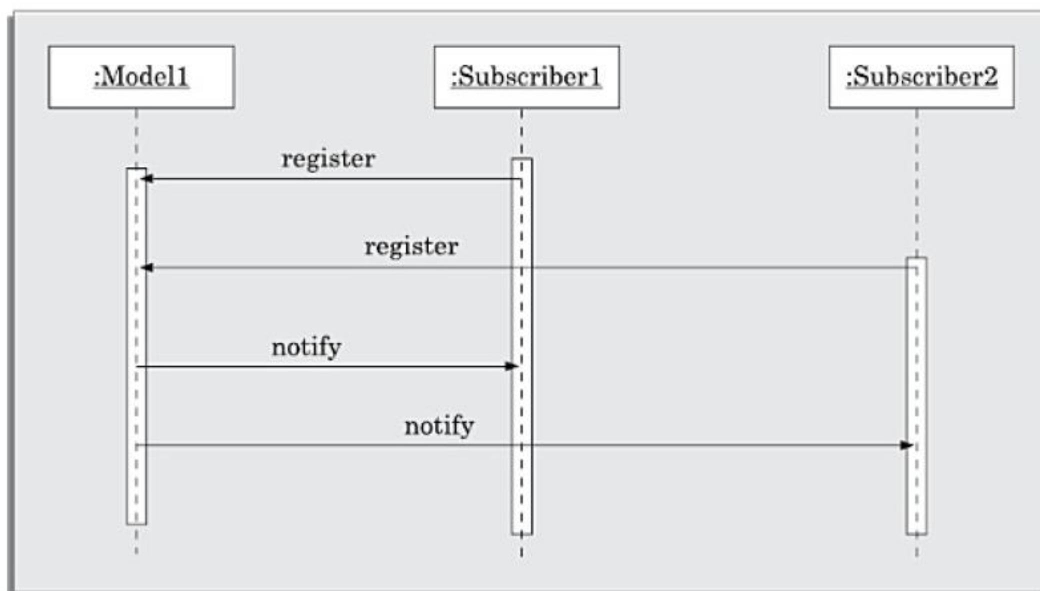
Solusi: Tergantung pada konteksnya, solusi yang diberikan oleh salah satu dari pola pengamat, pola model-view-controller(MVS), atau pola publish-subscribe (dibahas selanjutnya) dapat digunakan.

Penjelasan: Setiap kali interaksi antara objek model dan objek tampilan diperlukan, objek model harus digabungkan secara longgar ke objek tampilan. Ini akan membantu membuat penggunaan kembali objek model menjadi mudah. Juga, objek tampilan biasanya mengalami perubahan lebih sering dibandingkan dengan objek model. Setiap perubahan pada objek tampilan tidak memerlukan perubahan pada objek model.

Ketika informasi diperlukan untuk ditampilkan secara serempak, tarikan dari solusi di atas memuaskan. Namun, solusi ini tidak berfungsi dengan baik untuk tampilan informasi

asinkron. Ketika informasi diproduksi secara asinkron oleh objek model, objek tampilan tidak akan tahu kapan tepatnya informasi baru akan tersedia sehingga permintaan untuk itu dapat dibuat. Juga, polling (meminta informasi secara berkala untuk memeriksa apakah informasi baru telah tersedia) bukanlah solusi yang memuaskan untuk pertimbangan efisiensi.

Ada banyak situasi di mana informasi diperlukan untuk ditampilkan secara asinkron. Pertimbangkan perangkat lunak yang memantau kuotasi pasar saham secara terus menerus dan memperingatkan pengguna saat kuotasi saham yang dibutuhkan mencapai nilai ambang batas yang telah ditentukan sebelumnya. Perangkat lunak ini akan memunculkan pesan peringatan segera setelah salah satu kutipan saham mencapai nilai ambang batasnya. Dalam hal ini, pesan peringatan muncul secara tidak sinkron. GUI tidak akan dapat menarik informasi ini karena tidak akan mengetahui waktu yang tepat kapan ambang batas akan tercapai. Ada banyak contoh lain di mana tampilan asinkron seperti itu diperlukan, dan termasuk monitor intrusi jaringan, monitor kesalahan komputer, dll.



Gambar 8.3 Diagram interaksi untuk pola pengamat.

Ide utama di balik pola pemisahan model-view adalah untuk mencapai kopling longgar antara model dan objek tampilan, sehingga kompleksitas desain berkurang dan penggunaan kembali objek model ditingkatkan. Ada beberapa variasi pola pemisahan model-view yang berguna dalam situasi interaksi yang berbeda antara model dan objek view. Selanjutnya tiga pola pemisahan modelview pentingnya adalah sebagai berikut:

8.6 POLA PENGAMBIL

Masalah: Ketika objek model diakses oleh beberapa objek tampilan, bagaimana seharusnya interaksi antara model dan objek tampilan disusun?

Solusi: Pengamat harus mendaftarkan diri dengan objek model. Objek model akan mempertahankan daftar pengamat terdaftar. Ketika perubahan terjadi pada objek model, itu akan memberi tahu semua pengamat yang terdaftar. Setiap pengamat kemudian dapat menanyakan objek model untuk mendapatkan informasi spesifik tentang perubahan yang mungkin diperlukan. Oleh karena itu, pola ini menggunakan mode push dan pull. Diagram interaksi untuk pola ini ditunjukkan pada Gambar 8.3.

- Penjelasan: Solusi pola pengamat mencapai sambungan longgar antara model dan objek pengamat. Juga, mencapai decoupling di antara pengamat itu sendiri, mereka tidak perlu menyadari satu sama lain. Setelah pengamat diberi tahu

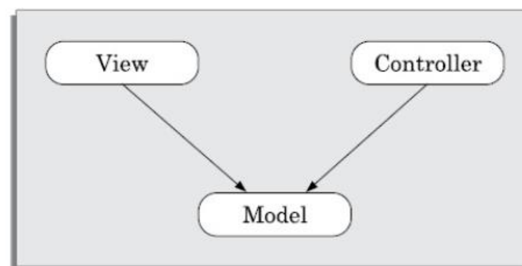
tentang perubahan, ia mungkin meminta model untuk beberapa informasi tertentu. Pola pengamat memiliki keterbatasan sebagai berikut:

- Objek model dikenakan biaya besar untuk mendukung pendaftaran pengamat, dan juga untuk menanggapi pertanyaan dari pengamat. Hal ini merugikan kinerja objek model, terutama ketika jumlah pengamat besar. Proses dua tahap (pemberitahuan dan kueri) mengurangi sambungan antara model dan objek pengamat. Namun, pemberitahuan selektif hanya kepada pengamat yang tertarik akan membuat solusi ini sangat tidak efisien, karena pertukaran pesan yang tidak perlu dapat terjadi ketika pengamat yang berbeda tertarik pada jenis kejadian yang berbeda.

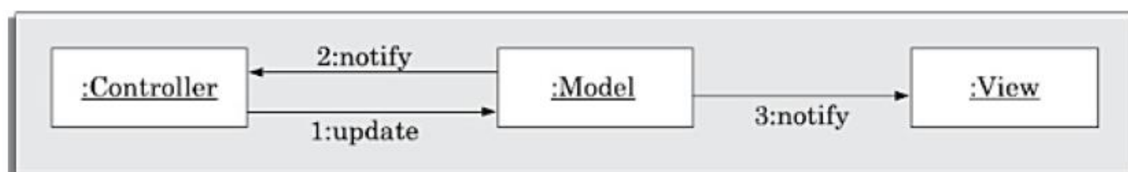
8.7 POLA *MODEL VIEW CONTROLLER* (MVC)

Masalah: Bagaimana seharusnya objek GUI berinteraksi dengan objek model?

Solusi: Objek GUI perlu dipisahkan menjadi tipe tampilan dan pengontrol. Objek pengontrol bertanggung jawab untuk mengumpulkan input data oleh pengguna. Pengontrol akan meneruskan informasi yang dikumpulkan ke objek model. Objek model akan memberi tahu objek tampilan dan pengontrol mengenai perubahan status model. Diagram kelas dan kolaborasi untuk pola MVC masing-masing telah ditunjukkan pada Gambar 8.4 dan 8.5.



Gambar 8.4 Struktur kelas untuk pola MVC.



Gambar 8.5 Model interaksi untuk pola MVC.

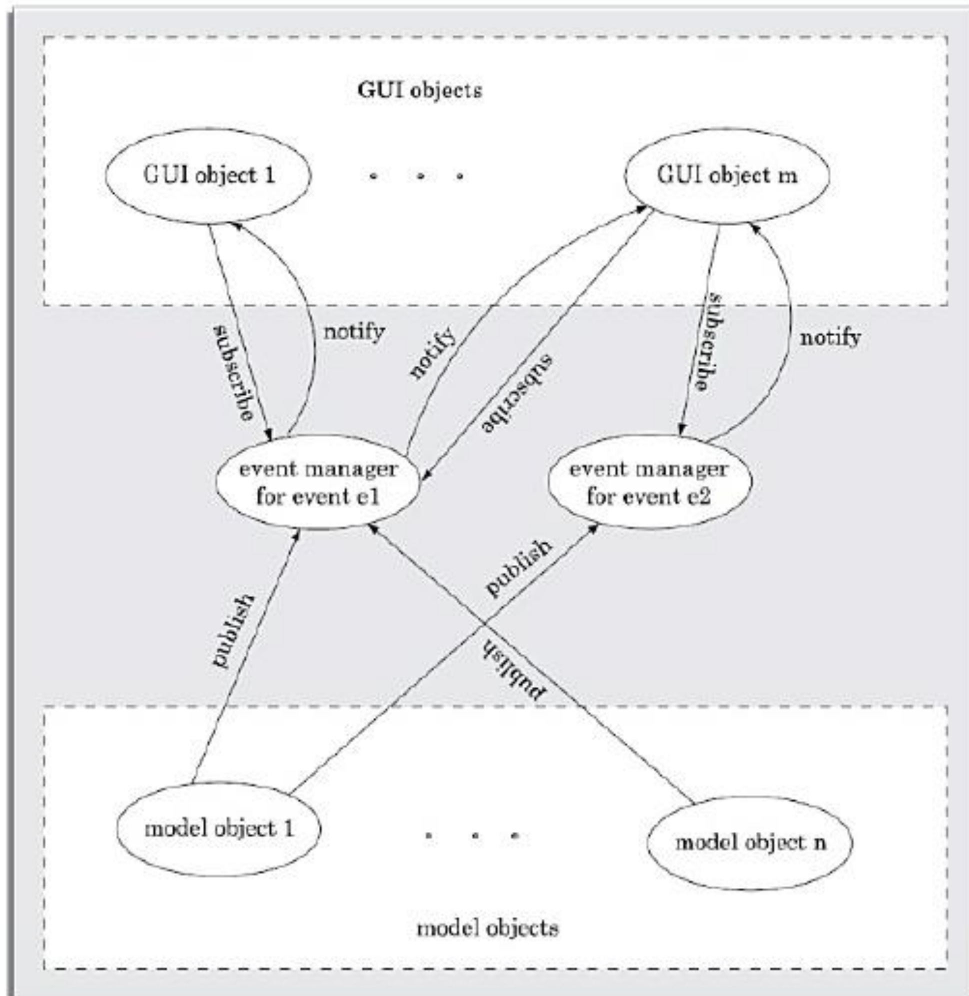
Penjelasan: Ketika objek pengontrol mengumpulkan data input dan meneruskannya ke objek model, status objek model dapat berubah. Saat status model berubah, model dapat mengirim pesan pembaruan ke semua objek tampilan dan pengontrol yang bergantung. Tujuan dari objek model yang memberi tahu objek tampilan sudah jelas (yaitu, untuk menyegarkan tampilan dengan informasi yang diperbarui). Objek pengontrol juga perlu diberi tahu, karena status objek model baru mungkin perlu meredupkan opsi menu tertentu, atau bahkan menutup objek kontrol tertentu. Manfaat dari pola ini termasuk penggabungan longgar antara tampilan dan objek model dan pemisahan yang jelas antara kode yang menangani input, yang menampilkan data, dan yang memproses data aplikasi.

Ini berguna dalam situasi di mana beberapa tampilan data yang sama perlu disajikan. Secara khusus, pola ini berguna dalam aplikasi di mana objek model dapat mengubah statusnya secara asinkron dan beberapa tampilan objek model yang konsisten perlu ditampilkan secara efektif. Sebagai contoh sederhana, pola MVC dapat digunakan ketika untuk input data yang sama beberapa representasi yang berbeda seperti plot garis, diagram batang,

dan diagram lingkaran perlu ditampilkan. Dalam hal ini, setiap kali model mengubah data atau propertinya, semua tampilan dependen diperbarui secara otomatis.

8.8 POLA PUBLIKASI-BERLANGGANAN

Pola publish-subscribe adalah bentuk yang lebih umum dari pola observer dan mengatasi banyak kekurangan dari pola observer.



Gambar 8.6 Model interaksi pola publish-subscribe

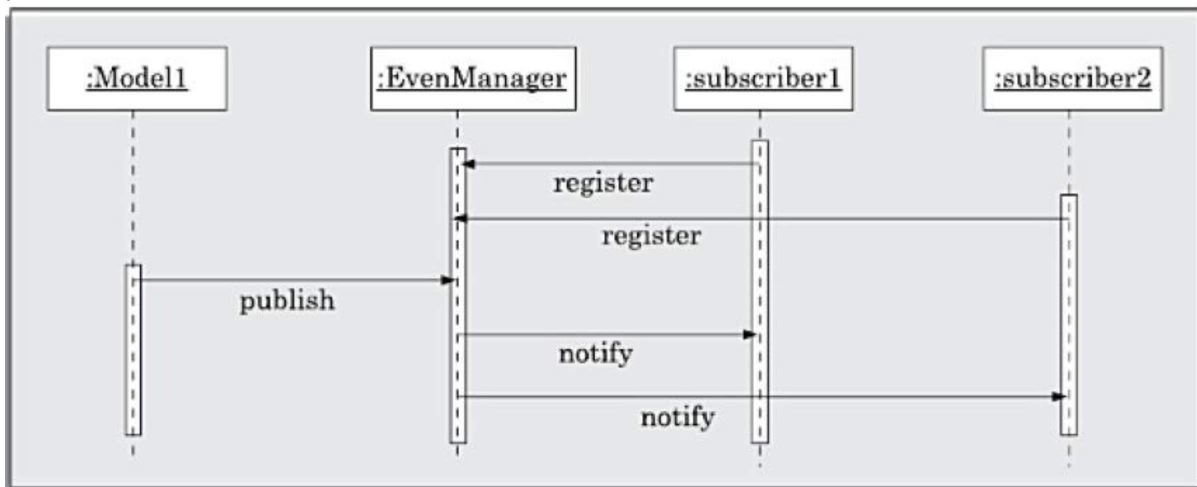
Masalah: Ketika objek model tertentu diakses oleh sejumlah besar objek tampilan, dan status model berubah secara tidak sinkron, bagaimana seharusnya interaksi terstruktur?

Solusi: Pola ini menyarankan bahwa sistem pemberitahuan peristiwa harus diterapkan di mana penerbit (objek model) dapat memberi tahu pelanggan secara tidak langsung segera setelah informasi yang diperlukan tersedia. Kelas manajer acara dapat ditentukan yang melacak pelanggan dan jenis acara yang mereka minati. Acara diterbitkan oleh penerbit dengan mengirimkan pesan ke objek pengelola acara. Manajer acara memberi tahu semua pelanggan terdaftar biasanya melalui pesan yang diparameterisasi (disebut callback).

Pola publish-subscribe secara skematis ditunjukkan pada Gambar 8.6. Amati bahwa objek GUI telah berlangganan ke acara e1 dengan pengelola acara yang sesuai. Segera setelah objek model menerbitkan acara e1, manajer acara terkait akan memberi tahu (panggilan balik) objek GUI yang telah berlangganan acara tersebut.

Penjelasan: Pola publish-subscribe memerlukan pembuatan kelas event manager untuk setiap jenis event yang mungkin dibuat oleh objek model. Pelanggan dapat

mendaftarkan minatnya untuk kelas acara dengan berlangganan kelas manajer acara yang sesuai. Acara yang dihasilkan oleh penerbit secara selektif diberitahukan kepada pelanggan. Seleksi dicapai dengan mendefinisikan kelas event manager. Segera setelah suatu acara terjadi, manajer acara menyampaikan informasi tersebut kepada semua orang yang telah berlangganan acara tersebut. Misalnya, saat penerbit membuat acara e1, pengelola acara yang sesuai akan diberi tahu tentang hal yang sama. Semua pelanggan yang sebelumnya berlangganan acara e1 akan menerima pemberitahuan asinkron tentang terjadinya e1. Pelanggan dapat menanggapi acara tersebut dengan mengasosiasikan penangan tertentu dengan setiap kelas acara. Manajer acara bertanggung jawab untuk mendaftarkan langganan, menerima acara dari model, memfilter acara, dan meneruskannya ke pelanggan yang tertarik (lihat Gambar 8.7). Bergantung pada sistem tertentu, pengelola acara dapat diimplementasikan dengan berbagai cara, misalnya, oleh server terpusat, oleh server kerja sama terdistribusi, atau mungkin secara kolektif oleh penerbit. Diagram interaksi ditunjukkan pada Gambar 8.7.



Gambar 8.7 Representasi skematis dari pola publish-subscribe.

Dibandingkan dengan pola pengamat, pola ini membebaskan objek model dari penanganan pendaftaran objek pengamat dan objek notifikasi. Dengan demikian, pola ini memiliki keunggulan yang jelas dibandingkan pola pengamat, terutama bila jumlah pengamat banyak. Bahasa berorientasi objek modern mendukung beberapa kelas pengelola acara. Misalnya, Java menyediakan antarmuka EventListener untuk tujuan tersebut.

8.9 POLA PERANTARA (ATAU PROKSI)

Masalah: Bagaimana seharusnya objek klien memanggil layanan objek server?

Konteks di mana masalah terjadi: Istilah klien dan server merujuk di sini ke objek yang ada di seluruh jaringan. Klien adalah konsumen layanan dan server adalah penyedia layanan.

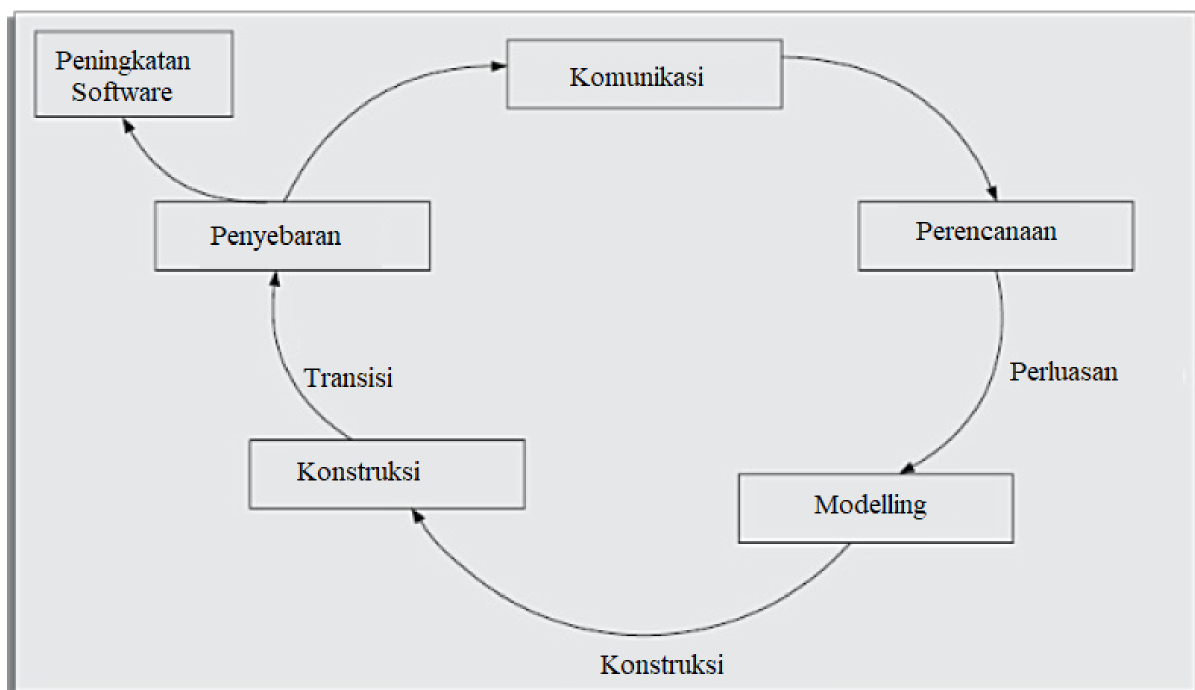
Solusi: Objek proxy harus dibuat di sisi klien. Objek proxy akan bertindak sebagai dudukan lokal untuk objek server jauh. Klien harus membuat semua permintaan layanannya ke proxy, proxy pada gilirannya akan mengirimkan permintaan layanan ke server yang sesuai, mendapatkan respons, dan mengirimkannya ke objek klien.

Penjelasan: Proxy menyembunyikan detail transmisi jaringan. Proxy bertanggung jawab untuk menentukan alamat server, mengomunikasikan permintaan klien ke server, memperoleh respons server dan dengan mulus meneruskannya ke klien. Proxy juga dapat menambah (atau menyaring) informasi yang dipertukarkan antara klien dan server. Misalnya, proxy dapat mengompresi atau mengenkripsi pesan saat mengirimnya ke server dan

membuka kompres atau mendekripsi pesan saat diterima. Proxy harus memiliki antarmuka yang sama dengan objek server jauh, sehingga klien dapat merasakan bahwa ia berinteraksi langsung dengan objek server jauh. Akibatnya, objek proxy menyembunyikan (mengabstraksi) kompleksitas transmisi jaringan.

8.10 METODOLOGI ANALISIS DAN DESAIN BERORIENTASI OBYEK (OOAD)

Proses desain yang akan kita bahas di sini dimulai dengan kegiatan analisis. Hasil dari kegiatan analisis tersebut disempurnakan menjadi model desain melalui beberapa iterasi. Mempertimbangkan bahwa proses terpadu sangat populer untuk pengembangan perangkat lunak berorientasi objek, pertama, menetapkan konteks mengenai fase proses terpadu di mana metodologi desain yang dibahas dapat diterapkan. Selanjutnya kit akan mendiskusikan proses analisis dan desain secara lebih rinci dan akhirnya mencari solusi untuk beberapa contoh masalah untuk menggambarkan penggunaannya.



Gambar 8.8 Model proses terpadu.

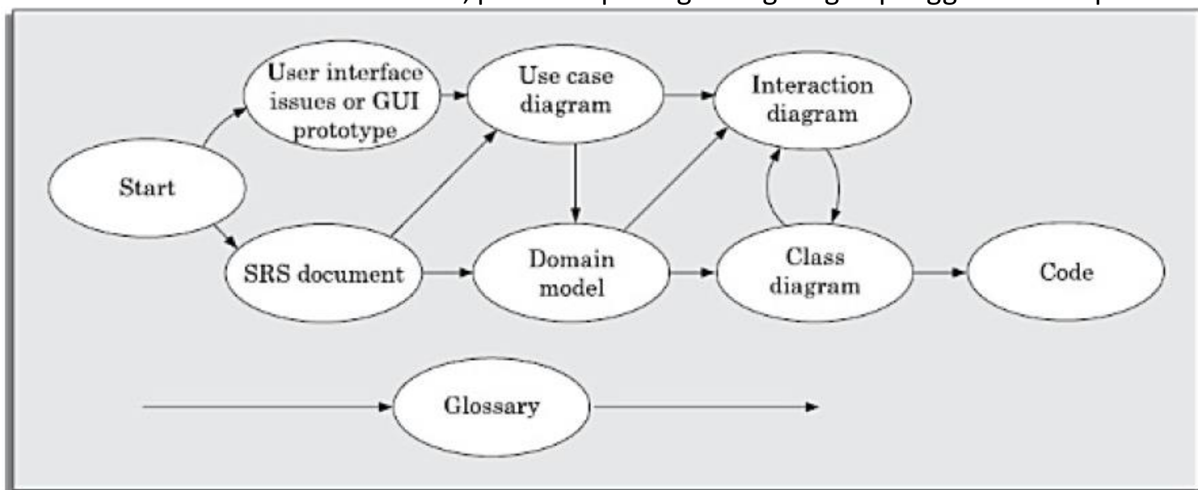
Proses Terpadu

Proses terpadu adalah tambahan dalam model proses iteratif untuk pengembangan perangkat lunak berorientasi objek yang telah diterima di kalangan praktisi dan akademisi. Buku pertama yang menggambarkan proses terpadu berjudul "Proses Pengembangan Perangkat Lunak Terpadu" dan diterbitkan pada tahun 1999 oleh Ivar Jacobson, Grady Booch dan James Rumbaugh. Sejak itu, berbagai penulis yang tidak berafiliasi dengan Rational Software Corporation sebelumnya telah menerbitkan buku dan artikel menggunakan nama proses terpadu, sedangkan penulis yang berafiliasi dengan Rational Software Corporation lebih menyukai nama proses terpadu rasional (RUP).

Proses terpadu adalah kerangka kerja yang dapat diperluas yang perlu disesuaikan untuk jenis proyek tertentu. Dua karakteristik utama dari proses terpadu adalah: use case-driven dan iterative. Istilah use case-driven menyiratkan bahwa use case (pandangan pelanggan) dari sistem dianggap sebagai pandangan sentral dan paling penting. Tampilan use case harus menjadi yang pertama dibangun dan harus disempurnakan secara iteratif menjadi implementasi. Model use case adalah model sentral. Semua model yang dibangun pada

kegiatan desain selanjutnya harus sesuai dengan model use case. Seperti yang ditunjukkan pada Gambar 8.8, proses terpadu melibatkan iterasi selama empat fase berbeda berikut sebagai berikut:

- **Inception**: Selama fase ini, ruang lingkup proyek didefinisikan dan prototipe dapat dikembangkan untuk membentuk gagasan yang jelas tentang proyek tersebut.
- **Elaborasi**: Pada fase ini, persyaratan fungsional dan non-fungsional ditangkap. Kasus penggunaan awal dan model domain dikembangkan selama fase ini.
- **Konstruksi**: Selama fase ini, kegiatan desain dan implementasi dilakukan. Deskripsi teks lengkap dari use case ditulis dan setiap use case diambil untuk memulai iterasi baru. Fitur sistem diimplementasikan dalam serangkaian iterasi pendek dan diuji. Setiap iterasi menghasilkan rilis perangkat lunak yang dapat dieksekusi.
- **Transisi**: Selama fase ini, produk dipasang di lingkungan pengguna dan dipelihara.



Gambar 8.9 Sebuah analisis berorientasi objek dan proses desain.

Ikhtisar Metodologi OOAD

Metodologi analisis dan desain berorientasi objek (OOAD) yang akan kita bahas secara skematis ditunjukkan pada Gambar 8.9. Seperti terlihat pada Gambar 8.9, model use case merupakan model pertama yang dikembangkan. Dalam setiap proses pengembangan yang berpusat pada pengguna seperti proses terpadu, semua model yang dikembangkan harus sesuai dengan model kasus penggunaan. Oleh karena itu, model use case memainkan peran penting dalam proses desain, dan perlu dikembangkan terlebih dahulu. Seperti yang ditunjukkan pada Gambar 8.9, model domain dibangun selanjutnya melalui analisis model use case dan dokumen SRS. Model domain disempurnakan menjadi diagram kelas melalui sejumlah iterasi yang melibatkan diagram interaksi. Setelah diagram kelas telah dibangun, dapat dengan mudah diterjemahkan ke kode. Banyak alat CASE mendukung pembuatan kerangka kode dari diagram kelas.

Sepanjang proses analisis dan desain, glosarium dibuat dan dipelihara secara terus menerus dan sadar. Glosarium adalah kamus istilah yang dapat membantu dalam memahami berbagai istilah (atau konsep) yang digunakan dalam model yang dibangun. Istilah yang tercantum dalam glosarium pada dasarnya adalah nama konsep. Glosarium (atau kamus model) mencantumkan dan mendefinisikan semua istilah yang memerlukan penjelasan untuk meningkatkan komunikasi dan mengurangi risiko kesalahpahaman. Mempertahankan glosarium adalah kegiatan yang berkelanjutan di seluruh proyek seperti yang ditunjukkan pada Gambar 8.9.

Gunakan Pengembangan Model Kasus

Model kasus penggunaan, pada dasarnya, menangkap persyaratan pengguna tingkat tinggi dari sistem yang akan dikembangkan. Untuk setiap kasus penggunaan, detail semua skenario interaksi pengguna dengan sistem ditangkap dalam deskripsi teks yang menyertainya. Di bagian ini kita membahas bagaimana seharusnya seseorang mengembangkan model use case untuk masalah yang diberikan? Kasus penggunaan dapat dengan mudah diidentifikasi dari dokumen SRS. Faktanya, persyaratan fungsional tingkat tinggi biasanya sesuai dengan kasus penggunaan. Namun, seringkali ternyata sebagian besar masalah praktis memiliki terlalu banyak use case. Oleh karena itu, kasus penggunaan ini harus dikemas dengan tepat. Namun, bagaimana cara menentukan struktur paket kasus penggunaan? Perlu dicatat bahwa ada korespondensi yang erat dengan struktur GUI, organisasi manual pengguna menjadi beberapa bagian, dan pengemasan kasus penggunaan. Prinsip utama saat mengidentifikasi dan mengemas kasus penggunaan adalah harus ada korelasi yang kuat antara prototipe GUI, isi manual pengguna dan model kasus penggunaan sistem.

Pertimbangkan perangkat lunak editor teks (seperti openOffice, MS-Word, atau Wordperfect, dll.) yang dapat digunakan untuk membuat berbagai jenis dokumen teks. Opsi menu tingkat atas untuk perangkat lunak pengolah kata seperti itu adalah Operasi file, Edit, Lihat, Sisipkan, Alat, dll. Masing-masing opsi menu tingkat tinggi ini dapat dianggap sebagai wadah untuk serangkaian fungsi dan biasanya diwakili menggunakan paket dalam diagram use case. Misalnya, menu File mungkin berisi fungsi seperti—memuat file, menyimpan file, menyimpan, menyimpan sebagai, mencetak, dll. Menu Edit mungkin berisi fungsi untuk memotong, menempel, memilih, mengelompokkan, dll. Setiap opsi menu di menu tingkat atas GUI biasanya akan sesuai dengan paket dalam diagram kasus penggunaan.

Beberapa fungsi di bawah opsi menu sendiri dapat berisi fungsi dalam arti bahwa mengkliknya akan membuka submenu. Submenu tersebut akan sesuai dengan paket dalam paket tingkat pertama dalam diagram use case. Cara use case dikelompokkan ke dalam paket juga memiliki korelasi kuat dengan organisasi manual pengguna ke dalam bab dan bagian. Manual pengguna akan memiliki bab yang sama persis dengan paket tingkat pertama dari diagram kasus penggunaan (atau organisasi menu tingkat atas di GUI). Bagian dari sebuah bab akan sesuai dengan paket dalam paket tingkat pertama yang sesuai, dll. Sebagai contoh, untuk kasus perangkat lunak pengolah kata, bab yang berbeda dari manual pengguna akan diatur ke dalam operasi File, Edit, Lihat, dan operasi Alat, dll., sesuai dengan struktur menu GUI dan juga paket kasus penggunaan.

Mempertimbangkan korespondensi yang erat antara manual pengguna dan prototipe GUI dengan organisasi use case, akan bermanfaat untuk mengembangkan manual pengguna terlebih dahulu dan kemudian mengembangkan model use case berdasarkan ini. Ini memberikan ringkasan yang nyaman tentang apa arti use case-driven development.

Biasanya, melakukan prototyping GUI secara paralel dengan pengembangan dokumen SRS (seperti yang ditunjukkan pada Gambar 8.9) dianggap sebagai pendekatan yang sangat baik. Ini melibatkan pengulangan aspek presentasi sistem dengan pengguna. Setelah mencapai penutupan pada layar yang berbeda, model kasus penggunaan yang sesuai dapat dikembangkan. Meskipun korespondensi yang erat harus ada antara prototipe GUI dan model use case, use case tidak boleh terlalu terikat erat dengan GUI. Misalnya, kasus penggunaan tidak boleh membuat referensi apa pun ke jenis elemen GUI yang muncul di layar, misalnya, radioButton, pushButton, dll. Ini diperlukan karena, jenis komponen antarmuka pengguna yang digunakan dapat sering berubah selama pengembangan perangkat lunak dan lebih lagi selama pemeliharaan nanti. Namun, fungsionalitas yang disediakan oleh sistem tidak terlalu sering berubah.

Sebuah use case paling efektif dinamai dari sudut pandang pengguna. Itu harus dinamai menggunakan frase kata kerja present tense, dan harus dalam suara aktif, misalnya, mengakui pasien, menerbitkan buku, membuat laporan, dll., daripada penerimaan pasien, penerbitan buku, pembuatan laporan, dll.

Kesalahan umum yang dilakukan dalam pengembangan model kasus penggunaan

Berikut ini adalah beberapa kesalahan umum yang dilakukan pemula selama pengembangan model use case:

Clutter: Terlalu banyak use case pada diagram use case tingkat atas membuat sangat sulit untuk memahami model. Ketika sejumlah besar kasus penggunaan hadir di tingkat atas diagram kasus penggunaan, mereka harus diatur ke dalam paket. Paket adalah cara yang efektif untuk mengelola kompleksitas. Setiap paket kasus penggunaan harus sesuai dengan satu bab atau bagian dari manual pengguna atau pilihan menu tingkat atas di GUI.

Terlalu detail: Seringkali pemula mengacaukan sub-langkah kasus penggunaan dengan kasus penggunaan terpisah. Misalnya, Seseorang mungkin salah mendefinisikan tanda terima cetak use case, padahal seharusnya hanya menjadi sub-langkah dari use case penarikan tunai. Model use case di mana subfungsi dari fungsi tingkat tinggi direpresentasikan sebagai use case penuh akan membuat tugas analisis dan desain selanjutnya menjadi sulit.

Menghilangkan deskripsi teks: Menghilangkan deskripsi teks dari kasus penggunaan membuat sangat sulit bagi siapa pun untuk mendapatkan pemahaman penuh tentang perilaku kasus penggunaan dan juga membuatnya sangat sulit untuk merancang sistem.

Menghadapi beberapa skenario alternatif: Penting untuk menangkap semua skenario alternatif dari setiap kasus penggunaan. Skenario yang diabaikan nantinya dapat muncul sebagai fungsi atau bug yang hilang.

Pemodelan Domain

Pemodelan domain juga dikenal sebagai pemodelan konseptual. Model konseptual menggambarkan konsep (atau objek) yang mudah diidentifikasi dari deskripsi masalah. Model domain biasanya berisi tiga jenis objek—objek yang sesuai dengan entitas fisik dalam deskripsi masalah, objek yang akan menangani antarmuka pengguna (juga disebut objek batas), dan objek yang sepenuhnya konseptual (juga disebut objek pengontrol). Objek yang diidentifikasi selama analisis domain dapat diklasifikasikan menjadi tiga jenis:

- Objek batas
- Objek pengontrol
- Objek entitas

Contoh objek fisik (entitas) dalam perangkat lunak otomasi perpustakaan adalah buku, register buku, register anggota, anggota perpustakaan, dll. Contoh objek konseptual dan objek batas masing-masing dapat berupa pengontrol buku, dan antarmuka pengguna buku. Model domain juga harus menangkap hubungan yang dapat dengan mudah diidentifikasi di antara objek-objek ini.

Model domain dapat dianggap sebagai diagram kelas potongan pertama dan diperoleh dari analisis deskripsi masalah. Dalam model domain, tidak ada metode atau atribut yang terkait dengan kelas dan hanya nama kelas yang diwakili. Metode (tanggung jawab) dan data (atribut) biasanya tidak direpresentasikan. Metode dan atribut diidentifikasi dan direpresentasikan kemudian dalam proses desain.

Selanjutnya, kita akan membahas karakteristik penting dari ketiga jenis objek berbeda yang diidentifikasi selama konstruksi model domain. Sementara objek batas dan pengontrol dapat diidentifikasi secara mekanis dari pemeriksaan diagram use case, identifikasi objek entitas memerlukan latihan. Jadi, kita dapat mengatakan bahwa inti dari aktivitas pemodelan domain adalah mengidentifikasi objek entitas secara bijaksana. Pertama membahas

bagaimana batas dan objek konseptual diidentifikasi dari pemeriksaan diagram use case. Selanjutnya, kita akan membahas metodologi untuk mengidentifikasi objek entitas.

Objek batas

Objek batas adalah objek yang berinteraksi dengan aktor. Objek batas termasuk layar, menu, formulir, dialog, dll. Objek batas terutama bertanggung jawab untuk mengevaluasi interaksi pengguna melalui antarmuka pengguna grafis (GUI) yang sesuai. Objek-objek ini biasanya tidak menyertakan logika pemrosesan apa pun. Namun, mereka mungkin bertanggung jawab untuk memvalidasi input, memformat output, dll. Kita dapat mengatakan dengan kata lain bahwa tanggung jawab utama objek batas dapat membaca input dari pengguna, memvalidasi input, memformat output, dan menampilkan hasilnya. .

Objek batas sebelumnya disebut sebagai objek antarmuka. Namun, istilah kelas antarmuka sekarang digunakan dengan arti yang berbeda (seperti yang ditunjukkan dalam Bab 7) untuk UML dan juga untuk Java dan COM/DCOM. Identifikasi awal dari kelas batas dapat dilakukan dengan mendefinisikan satu kelas batas per pasangan aktor/use case.

Anda mungkin bertanya-tanya bahwa ketika dua atau lebih aktor yang berbeda diasosiasikan dengan use case (misalnya, use case pelanggan register pada Gambar 8.16) mengapa dua kelas batas yang berbeda digunakan? Alasan di balik ini adalah bahwa kategori pengguna yang berbeda memiliki hak istimewa yang berbeda dan tingkat keakraban yang berbeda dengan paket perangkat lunak. Misalnya, petugas akan menggunakan antarmuka setiap hari dan akan membutuhkan antarmuka yang efisien daripada antarmuka yang sangat ramah pengguna tetapi lambat.

Kemudian selama proses desain, kelas batas dapat dipecah menjadi dua atau lebih kelas jika ditemukan melakukan serangkaian tugas yang besar dan kompleks. Dua kelas batas dapat digabungkan menjadi satu, jika mereka memiliki tanggung jawab yang sama.

Objek entitas

Objek entitas biasanya menyimpan informasi seperti tabel data dan file (misalnya, Book, BookRegister, LibraryMember, dll.). Di antara ketiga jenis kelas yang kita bahas, kelas entitas adalah satu-satunya kelas yang dapat menyimpan data secara permanen atau semi permanen. Objek entitas biasanya menyimpan data di seluruh eksekusi kasus penggunaan dan oleh karena itu perlu hidup lebih lama dari eksekusi kasus penggunaan. Biasanya objek entitas bertindak seperti "server bodoh". Ini berarti bahwa objek-objek ini hanya melakukan pemrosesan sederhana pada data yang disimpannya seperti memperbarui, menyimpan, mencari, mengambil, dll. Operasi primitif seperti itu pada data biasanya tidak sering berubah seiring waktu. Oleh karena itu, objek entitas mengalami perubahan yang jauh lebih sedikit selama fase operasi perangkat lunak dibandingkan dengan jenis objek lainnya dan dikatakan lebih stabil daripada jenis objek lainnya.

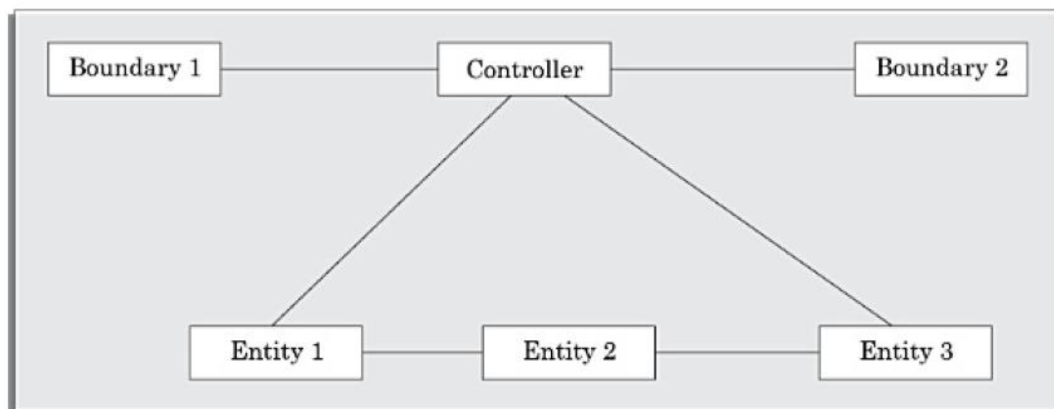
Objek pengontrol

Biasanya, setiap eksekusi use case melibatkan beberapa interaksi di antara sekelompok objek. Setiap objek yang terlibat dalam eksekusi use case memainkan perannya (melakukan tindakan tertentu) untuk membantu menyelesaikan eksekusi use case. Dalam konteks ini, biasanya objek pengontrol mengoordinasikan aktivitas sekumpulan objek yang berkolaborasi untuk memberikan hasil yang sesuai dengan permintaan aktor. Untuk setiap kasus penggunaan, objek pengontrol terpisah harus dibuat dan diberi tanggung jawab untuk menangani permintaan aktor.

Untuk setiap pemanggilan use case oleh aktor yang sama (atau berbeda), objek pengontrol yang terpisah harus diinstansiasi dari kelas pengontrol yang sesuai. Ketika seorang aktor memanggil use case, objek pengontrol yang sama harus menangani semua interaksi dengan aktor itu. Dengan cara ini, menjadi mungkin untuk dengan mudah mempertahankan

informasi yang diperlukan tentang keadaan eksekusi use case oleh seorang aktor. Informasi status yang dikelola oleh pengontrol dapat digunakan untuk mengidentifikasi permintaan aktor yang tidak berurutan (misalnya, apakah permintaan voucher cetak diterima sebelum mengatur permintaan pembayaran), dan kemudian dapat mengambil tindakan yang sesuai.

Nama "controller" sesuai karena objek ini mengoordinasikan (atau mengontrol) aktivitas beberapa objek untuk melayani permintaan pengguna. Objek pengontrol biasanya mengatur aktivitas sekumpulan kelas entitas untuk mengimplementasikan perilaku kasus penggunaan dan juga mengelola interaksi pengguna melalui serangkaian kelas batas (lihat Gambar 8.10). Dari Gambar 8.10 dapat dilihat bahwa controller menerima input dan menampilkan output kepada pengguna melalui dua kelas batas (Boundary 1 dan Boundary 2), juga mengkoordinasikan aktivitas kelas entitas (Entitas 1, Entitas 2, dan Entitas 3) untuk mewujudkan perilaku yang terkait dengan use case.



Gambar 8.10 Realisasi khas dari use case melalui kolaborasi objek batas, pengontrol, dan entitas.

Beberapa pertanyaan yang mungkin muncul di benak kita mengenai kelas controller adalah sebagai berikut:

Apa keuntungan menggunakan kelas pengontrol?

Objek pengontrol secara efektif memisahkan objek batas dan entitas satu sama lain, mengurangi kompleksitas desain dan membuat sistem toleran terhadap perubahan antarmuka pengguna dan logika pemrosesan. Objek pengontrol mewujudkan sebagian besar logika pemrosesan bisnis yang diperlukan untuk realisasi kasus penggunaan (logika bisnis dapat berubah dari waktu ke waktu). Untuk alasan ini, kita dapat menyukai objek pengontrol dengan manajer yang memerintahkan dan mengawasi aktivitas sekumpulan pekerja. Ketika logika bisnis berubah, hanya pengontrol dan objek batas yang dapat berubah, objek entitas tidak.

Apa peran objek pengontrol?

Untuk setiap use case, objek controller diberi tugas untuk mewujudkan perilaku yang terkait dengan use case dengan memanfaatkan layanan entitas dan kelas batas yang diperlukan. Sebagai contoh, pengontrol use case pembaruan Buku dalam sistem otomasi perpustakaan mungkin pertama-tama meminta untuk mengambil dan memasok buku-buku yang telah dipinjam oleh pengguna, kemudian akan meminta pengguna untuk membuat pilihan buku yang perlu diperbarui melalui kelas batas. Kemudian akan meminta BookRegister untuk memeriksa apakah ada orang yang telah memesan buku yang diminta, dalam kasus seperti itu akan memberi tahu pengguna bahwa tidak mungkin untuk memperbarui buku. Jika tidak, itu akan meminta BookRegister untuk memperbarui buku dan menampilkan informasi kepada pengguna. Dengan demikian, controller adalah kelas cerdas yang mengetahui

subtugas yang tepat untuk diselesaikan untuk memberikan hasil yang diperlukan ketika klien memanggil use case. Pengontrol memerintahkan objek yang berbeda untuk menyelesaikan subtugas yang berbeda. Dalam terminologi OOAD, kelas pengontrol dikatakan "merealisasikan kasus penggunaan yang sesuai." Kelas pengontrol dapat dianggap sebagai kelas cerdas dan kelas lainnya (entitas dan batas) sebagai kelas bodoh yang digerakkan oleh kelas pengontrol.

Apakah tepat satu kelas pengontrol diperlukan per kasus penggunaan?

Setiap use case (penggunaan kasus) biasanya direalisasikan menggunakan satu objek pengontrol. Namun, kasus penggunaan yang sangat sederhana dapat direalisasikan tanpa menggunakan objek pengontrol apa pun, yaitu melalui objek batas dan entitas saja. Hal ini sering benar untuk kasus penggunaan sepele yang hanya melakukan beberapa operasi sederhana yang dapat dilakukan baik oleh kelas batas itu sendiri atau dengan bantuan kelas entitas tunggal. Kasus penggunaan yang lebih kompleks yang perlu menerapkan logika bisnis yang signifikan mungkin memerlukan lebih dari satu objek pengontrol untuk mewujudkan kasus penggunaan. Kasus penggunaan yang kompleks dapat memiliki beberapa objek pengontrol dengan tanggung jawab seperti manajer transaksi, koordinator sumber daya, dan penangan kesalahan. Untuk setiap pemanggilan use case, instance terpisah dari kelas pengontrol yang sesuai perlu dibuat. Misalnya, kasus penggunaan memerlukan objek pengontrol untuk transit melalui sejumlah status. Oleh karena itu, keadaan pengontrol yang tepat untuk setiap contoh eksekusi kasus penggunaan perlu dilacak. Dalam kasus seperti itu, satu objek pengontrol mungkin harus dibuat untuk setiap eksekusi kasus penggunaan.

Dalam metodologi solusi awal, satu kelas pengontrol dibuat untuk setiap kasus penggunaan. Kemudian, jika ditentukan bahwa kelas pengontrol memiliki tanggung jawab yang sangat dangkal atau sepele, itu dapat dihilangkan dari desain. Jika ditemukan terlalu kompleks, mungkin akan dipecah menjadi beberapa kelas pengontrol.

Bagaimana jika beberapa kelas terlewatkan dalam model domain?

Untuk membuat model domain, strategi yang disarankan adalah dengan cepat membuat model konseptual kasar di mana penekanannya adalah pada menemukan konsep (objek) yang jelas yang diungkapkan dalam persyaratan sambil menunda penyelidikan terperinci. Tentu saja, untuk masalah yang kompleks, ada kemungkinan bahwa beberapa kelas dapat dilewatkan oleh perancang. Kemudian selama proses desain, model konseptual secara bertahap disempurnakan dan diperluas. Dalam proses ini, beberapa kelas yang terlewatkan dalam model kelas akan diidentifikasi dan digabungkan, dan beberapa kelas yang memiliki terlalu banyak tanggung jawab dapat dipecah menjadi beberapa kelas, dan bahkan beberapa kelas yang dianggap tidak perlu dapat dihapus. Metode dan atribut kelas serta hubungan kelas akan dibuat dan ditambahkan nanti.

Identifikasi Objek Entitas

Dalam setiap metodologi desain berorientasi objek, identifikasi objek entitas merupakan langkah penting. Faktanya, kualitas desain akhir sangat bergantung pada kesesuaian dan kelengkapan objek entitas yang diidentifikasi. Namun, hingga saat ini, tidak ada metodologi langkah-demi-langkah yang sistematis untuk identifikasi objek entitas, meskipun kelas batas dan pengontrol mudah diidentifikasi dan dapat ditentukan hampir secara mekanis dari analisis diagram use case. Beberapa pendekatan semi formal dan informal telah diusulkan untuk identifikasi objek entitas. Semua teknik ini memerlukan penerapan akal sehat dan beberapa penilaian subjektif perlu dibuat berdasarkan pengalaman masa lalu saat menerapkan teknik identifikasi objek. Berbagai teknik identifikasi objek dapat diklasifikasikan ke dalam kelas luas berikut:

- Analisis gramatikal dari deskripsi masalah.
- Derivasi dari aliran data.

- Derivasi dari diagram hubungan entitas (E R).

Pendekatan identifikasi objek yang diterima secara luas adalah pendekatan analisis gramatikal yang diusulkan oleh Grady Booch[1991]. Dalam pendekatan analisis gramatikal Booch, kata benda yang muncul dalam deskripsi masalah yang diperluas (narasi pemrosesan) dipetakan ke objek dan kata kerja dipetakan ke metode. Uraian masalah yang diperluas adalah deskripsi fungsi (kotak hitam) dari data input, data output, dan pemrosesan yang perlu dilakukan pada input untuk mendapatkan output. Pendekatan untuk identifikasi kelas entitas berdasarkan derivasi dari diagram aliran data dan model hubungan entitas dapat digunakan untuk menyempurnakan hasil yang diperoleh dengan menggunakan metodologi identifikasi objek Booch.

Metode Identifikasi Objek Booch

Pendekatan identifikasi objek Booch membutuhkan narasi pemrosesan dari masalah yang diberikan untuk dikembangkan terlebih dahulu. Narasi pemrosesan adalah deskripsi keseluruhan masalah dan juga diskusi tentang bagaimana masalah yang diberikan dapat diselesaikan. Dari objek yang diidentifikasi dari narasi pemrosesan, jika suatu objek berkaitan dengan penerapan solusi, maka dikatakan sebagai bagian dari ruang solusi. Sebaliknya, jika suatu objek hanya diperlukan untuk menggambarkan masalah, maka dikatakan sebagai bagian dari ruang masalah. Objek diidentifikasi melalui analisis leksikal narasi pemrosesan dengan mencatat kata benda (kata nama) dalam narasi pemrosesan. Tentu saja, banyak kata benda yang terdaftar dari tujuan pemrosesan mungkin bukan objek. Dari daftar kata benda ini, sinonim kata benda tentunya harus segera dihilangkan.

Perlu dicatat bahwa beberapa kata benda mungkin tidak sesuai dengan objek. Berikut ini adalah beberapa kriteria yang dapat digunakan untuk menghilangkan beberapa kata benda yang telah diidentifikasi melalui analisis gramatikal dari narasi pemrosesan dan menyatu pada kumpulan objek yang sebenarnya:

Pengguna: Ada berbagai kategori pengguna (aktor) perangkat lunak dan pengguna ini biasanya muncul sebagai kata nama dalam deskripsi masalah. Para aktor itu sendiri dan interaksi di antara para aktor harus dikeluarkan dari latihan identifikasi entitas. Namun, terkadang ada kebutuhan untuk memelihara informasi tentang aktor di dalam sistem. Hanya dalam kasus seperti itu, kelas yang sesuai dengan nama aktor harus dipertimbangkan. Kelas seperti itu kadang-kadang disebut pengganti. Misalnya, dalam sistem informasi perpustakaan (LIS) kita perlu menyimpan informasi tentang setiap anggota perpustakaan seperti nama, alamat, nomor identifikasi, nomor telepon, dll. Ini tidak tergantung pada fakta bahwa anggota perpustakaan juga berperan dari aktor sistem.

Kata nama sebenarnya: Nama prosedur imperatif, yaitu, bentuk kata benda dari kata kerja sebenarnya mewakili suatu tindakan dan tidak boleh dianggap sebagai objek. Misalnya, penarikan tunai dalam deskripsi pemrosesan ATM mengacu pada penarikan uang tunai dan merupakan tindakan dan bukan kata benda, dan karenanya dapat dihapus dari daftar kata benda.

Informasi yang disimpan: Setiap objek entitas harus memiliki beberapa atribut (data tersimpan) yang terkait dengannya dan metode objek entitas harus beroperasi pada data ini. Akan sangat tidak biasa jika kita memiliki objek yang hanya memiliki seperangkat metode dan tidak ada informasi yang disimpan.

Informasi yang disimpan ini adalah data pribadi dari objek yang telah kita bahas sebelumnya. Jika suatu objek tidak berisi data pribadi apa pun, biasanya objek tersebut tidak dapat diharapkan menjadi objek entitas. Jadi, jika kita tidak dapat mengaitkan atribut yang berarti dengan suatu objek, kita harus menghilangkannya dari daftar kata benda. Misalnya, perhatikan pernyataan berikut dalam narasi pemrosesan—Hasilnya ditampilkan di layar

komputer. Di sini, layar komputer tidak dapat menjadi objek karena kita tidak dapat mengaitkan informasi apa pun yang disimpan di layar komputer untuk diproses nanti.

Beberapa atribut: Biasanya objek memiliki banyak atribut dan mendukung banyak metode. Sangat jarang menemukan objek berguna yang hanya menyimpan satu elemen data atau hanya mendukung satu metode, karena objek yang hanya memiliki satu elemen data atau metode biasanya diimplementasikan sebagai bagian dari objek lain. Jadi, jika kita tidak dapat mengaitkan beberapa atribut dan metode dengan kata benda, kita harus dengan mudah mengecualikannya dari daftar kata benda. Perhatikan pernyataan: "Setiap mahasiswa memiliki nomor telepon." Di sini, nomor telepon tidak boleh dijadikan objek entitas karena tidak dapat mengaitkan informasi apa pun selain nomor telepon dengan objek telepon.

Operasi umum: Setelah menentukan satu set operasi yang dapat diidentifikasi untuk satu objek potensial, operasi ini harus berlaku untuk semua kemunculan objek yang sama. Hanya dengan begitu kita harus mendefinisikan kata benda sebagai kelas. Kita telah melihat bahwa sebuah atribut atau operasi yang didefinisikan untuk sebuah kelas harus berlaku untuk setiap instance dari kelas tersebut. Jika beberapa atribut atau operasi hanya berlaku untuk beberapa instance kelas tertentu, maka kita perlu memiliki satu atau lebih subkelas untuk objek khusus ini. Pertimbangkan pernyataan —"Siswa residen perlu diberi kamar asrama, sedangkan untuk mahasiswa non-residen nomor telepon kontak lokal dan alamat lokal perlu disimpan." Di sini, kita membutuhkan dua kelas: mahasiswa residen dan mahasiswa non-residen yang perlu diturunkan dari kelas siswa, karena kedua jenis ini harus memiliki atribut dan operasi yang berbeda, meskipun beberapa operasi dan atribut akan serupa.

Meskipun pendekatan gramatikal sederhana dan menarik secara intuitif, namun melalui penggunaan pendekatan ini secara naif, sangat sulit untuk mencapai hasil berkualitas tinggi jika pendekatan tersebut digunakan secara naif. Secara khusus, sangat sulit untuk menghasilkan abstraksi yang berguna hanya dengan melakukan analisis gramatikal dari deskripsi masalah. Abstraksi yang berguna biasanya dihasilkan dari pemfaktoran yang cerdas dari deskripsi masalah menjadi elemen yang independen dan benar secara intuitif. Namun, pendekatan tata bahasa dapat berfungsi sebagai panduan kasar yang perlu digunakan dengan beberapa pemikiran daripada dengan surat.

Pedoman bermanfaat lainnya

Berikut ini adalah beberapa panduan yang lebih bermanfaat untuk mengidentifikasi objek entitas dari narasi pemrosesan:

Objek agregat: Kelas entitas biasanya muncul sebagai objek agregat. Ini karena data yang akan disimpan biasanya didistribusikan di antara beberapa objek. Misalnya, objek mahasiswa dapat digabungkan ke dalam register siswa, objek buku dapat digabungkan ke dalam register buku, dll.

Sesuai dengan penyimpanan data di DFD: Seringkali, objek entitas secara kasar sesuai dengan penyimpanan data dalam DFD yang dirancang dengan baik. Dengan demikian, model DFD, jika tersedia, dapat membantu mengidentifikasi objek entitas.

Register di dunia fisik: Objek entitas biasanya sesuai dengan register yang perlu dipelihara dalam kerja manual sistem, Misalnya, di perpustakaan, biasanya register buku dan register anggota dipelihara. Ini dapat dianggap sesuai dengan objek entitas agregat.

Contoh 8.1 Mari kita mengidentifikasi objek entitas dari perangkat lunak Tic-tac-toe berikut: Tic-tac-toe adalah permainan komputer di mana pemain manusia dan komputer melakukan gerakan bergantian pada kotak 3 × 3. Sebuah langkah terdiri dari menandai kotak yang sebelumnya tidak bertanda. Seorang pemain yang pertama kali menempatkan tiga tanda berturut-turut di sepanjang garis lurus (yaitu, di sepanjang baris, kolom, atau diagonal) di kotak menang. Segera setelah pemain manusia atau komputer menang, pesan ucapan

selamat kepada pemenang akan ditampilkan. Jika tidak ada pemain yang berhasil mendapatkan tiga nilai berturut-turut di sepanjang garis lurus, dan semua kotak di papan terisi, maka permainan ditarik. Komputer selalu mencoba untuk memenangkan permainan.

Jika kita melakukan analisis gramatikal dari pernyataan masalah ini, kita akan menemukan kata benda yang telah dicetak miring. Namun, pada pemeriksaan lebih dekat kita dapat menghilangkan sinonim dari kata benda yang diidentifikasi. Daftar kata benda setelah menghilangkan sinonimnya adalah sebagai berikut—Tic-tac-toe, computer game, human player, move, square, mark, straight line, board, row, column, dan diagonal.

Dari daftar objek yang mungkin ini, kita dapat menghilangkan kata benda seperti pemain manusia seperti halnya aktor. Selain itu, kita dapat menghilangkan kata benda persegi, game, komputer, Tic-tac-toe, garis lurus, baris, kolom, dan diagonal, karena kita tidak dapat mengaitkan data dan metode apa pun dengannya. Kita juga dapat menghilangkan perpindahan kata benda dari daftar objek potensial karena ini adalah kata kerja imperatif dan benar-benar mewakili suatu tindakan. Jadi, kita hanya memiliki satu objek yang berarti — board.

Setelah kita menjadi sedikit berpengalaman dalam identifikasi objek setelah menyelesaikan beberapa masalah, biasanya tidak perlu untuk benar-benar mengidentifikasi semua kata benda dalam deskripsi masalah dengan menggarisbawahi atau benar-benar mendaftarnya, dan secara sistematis menghilangkan non-objek untuk sampai pada akhir kumpulan objek. Dengan beberapa pengalaman, kita dapat mengidentifikasi kumpulan objek dengan membaca deskripsi masalah dengan cermat dan menganalisisnya secara mental.

Pemodelan Interaksi

Ingat dari diskusi kita di Bab 7 bahwa perilaku yang terkait dengan use case diwujudkan melalui interaksi beberapa objek. Interaksi ini dikoordinasikan oleh objek pengontrol. Interaksi yang terjadi di antara sekelompok objek untuk mewujudkan use case ditangkap melalui diagram interaksi. Tujuan utama dari pemodelan interaksi adalah sebagai berikut:

- Untuk mengalokasikan tanggung jawab realisasi use case di antara objek batas, entitas, dan pengontrol. Tanggung jawab untuk setiap kelas direfleksikan sebagai operasi (metode) yang harus didukung oleh kelas tersebut.
- Untuk menunjukkan untuk setiap use case interaksi terperinci yang terjadi dari waktu ke waktu di antara objek terkait untuk menyelesaikan eksekusi use case.

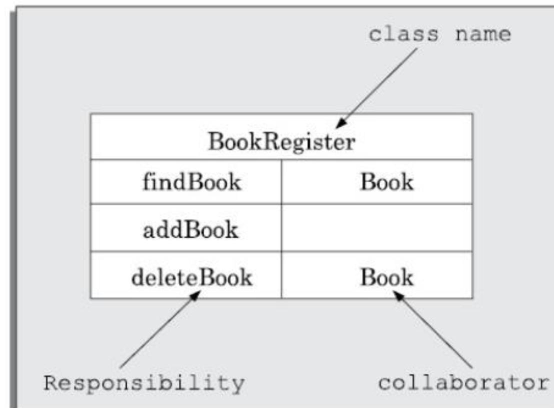
Dalam bab 7 pemodelan interaksi ditangkap melalui urutan UML dan diagram kolaborasi. Namun, diagram kolaborasi dapat diperoleh secara mekanis dari diagram urutan. Oleh karena itu, biasanya cukup untuk mengembangkan diagram urutan saja. Anda biasanya perlu melakukan satu diagram urutan untuk setiap kasus penggunaan. Artinya, untuk setiap kasus penggunaan (mungkin terdiri dari banyak skenario) hanya diagram urutan terpisah yang harus dirancang untuk menangkap interaksi yang terjadi di antara objek terkait. Kemudian dalam proses desain, diagram interaksi dapat dipecah menjadi dua atau lebih diagram terpisah, jika diagram interaksi menjadi terlalu rumit atau tidak dapat diakomodasi dengan jelas pada kertas ukuran standar.

Kartu Kelas-Tanggung Jawab-Kolaborator/*Class-Responsibility-Collaborator* (CRC)

Pendekatan *Class-Responsibility-Collaborator* (CRC) dipelopori oleh Ward Cunningham dan Kent Beck di laboratorium penelitian Tektronix di Portland, Oregon, AS. Diagram interaksi dari use case sederhana biasanya melibatkan menangkap kolaborasi di antara beberapa objek. Untuk kasus penggunaan sederhana seperti itu, diagram interaksi dapat dengan mudah ditarik dari pemeriksaan deskripsi kasus penggunaan. Namun, merancang diagram interaksi untuk kasus penggunaan yang lebih kompleks mungkin melibatkan kolaborasi banyak objek dan interaksi di antara objek-objek ini bisa sulit untuk dipahami oleh seorang individu.

Mengembangkan diagram interaksi untuk kasus penggunaan seperti itu mungkin memerlukan partisipasi tim developer melalui penggunaan kartu CRC. Dalam pendekatan kartu CRC, sejumlah anggota tim berpartisipasi untuk menetapkan tanggung jawab ke kelas yang terlibat dalam realisasi use case.

Kartu CRC pada dasarnya adalah kartu indeks yang digunakan untuk menetapkan tanggung jawab ke kelas. Satu kartu CRC disiapkan untuk setiap kelas yang diwakili dalam model domain. Pada masing-masing kartu ini, tanggung jawab masing-masing kelas ditulis secara singkat ketika dan ketika mereka diidentifikasi. Objek-objek yang perlu dikolaborasikan dengan objek ini untuk memenuhi tanggung jawabnya juga ditulis.



Gambar 8.11 Kartu CRC untuk kelas BookRegister.

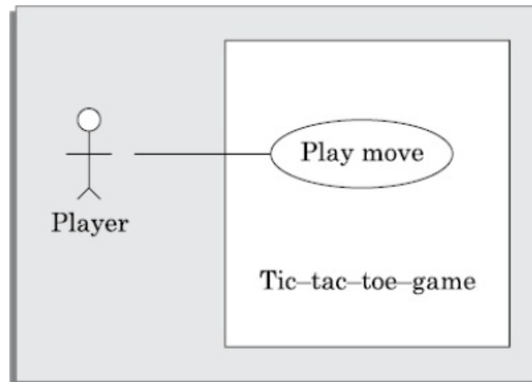
Kartu CRC biasanya dikembangkan dalam sesi kelompok kecil di mana orang-orang bermain peran dalam berbagai kelas. Setiap orang memegang kartu CRC dari kelas yang dia mainkan. Kartu-kartu tersebut sengaja dibuat kecil (4 inci × 6 inci) sehingga setiap kelas hanya dapat memiliki jumlah tanggung jawab yang terbatas. Tanggung jawab adalah deskripsi tingkat tinggi dari bagian yang perlu dimainkan oleh kelas dalam realisasi kasus penggunaan. Tanggung jawab biasanya berupa metode tunggal, tetapi juga dapat diimplementasikan oleh beberapa metode. Kasus penggunaan yang berbeda diambil satu per satu. Mulai dari inisiasi use case oleh aktor, interaksi dengan pengguna dianalisis dan tugas spesifik yang perlu dilakukan oleh sistem ditentukan. Permainan peran anggota tim untuk kelas menentukan apakah kelas mereka harus mengambil tanggung jawab dengan mempertimbangkan pro dan kontra. Contoh kartu CRC untuk kelas BookRegister dari Sistem Otomasi Perpustakaan ditunjukkan pada Gambar 8.11. Setelah Anda menetapkan tanggung jawab untuk kelas menggunakan kartu CRC, Anda dapat mengembangkan diagram interaksi dengan membalik-balik kartu CRC.

8.11 APLIKASI ANALISIS DAN PROSES DESAIN

Selanjutnya adalah bagaimana proses analisis dan desain dapat digunakan dengan menerapkannya pada dua contoh masalah.

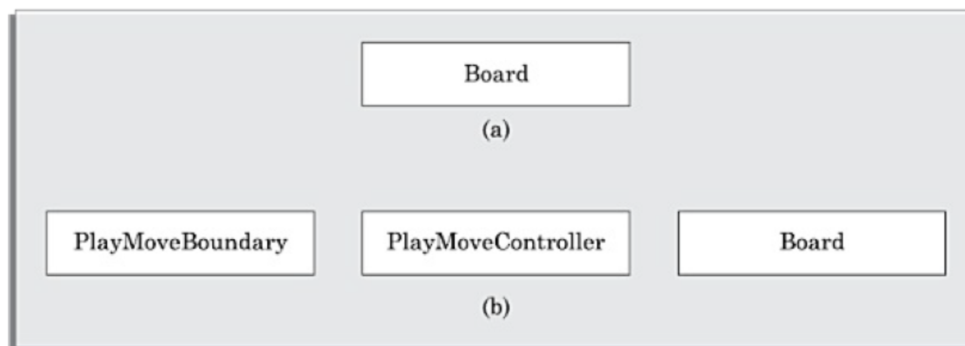
Contoh 8.2 Perhatikan permainan komputer Tic-tac-toe yang diuraikan dalam Contoh 6.2 dari Bab 6. Latihan langkah demi langkah dari contoh ini adalah sebagai berikut:

- Model use case ditunjukkan pada Gambar 8.12.
- Model domain awal ditunjukkan pada Gambar 8.13(a).
- Model domain setelah menambahkan kelas batas dan kontrol ditunjukkan pada Gambar 8.13(b).

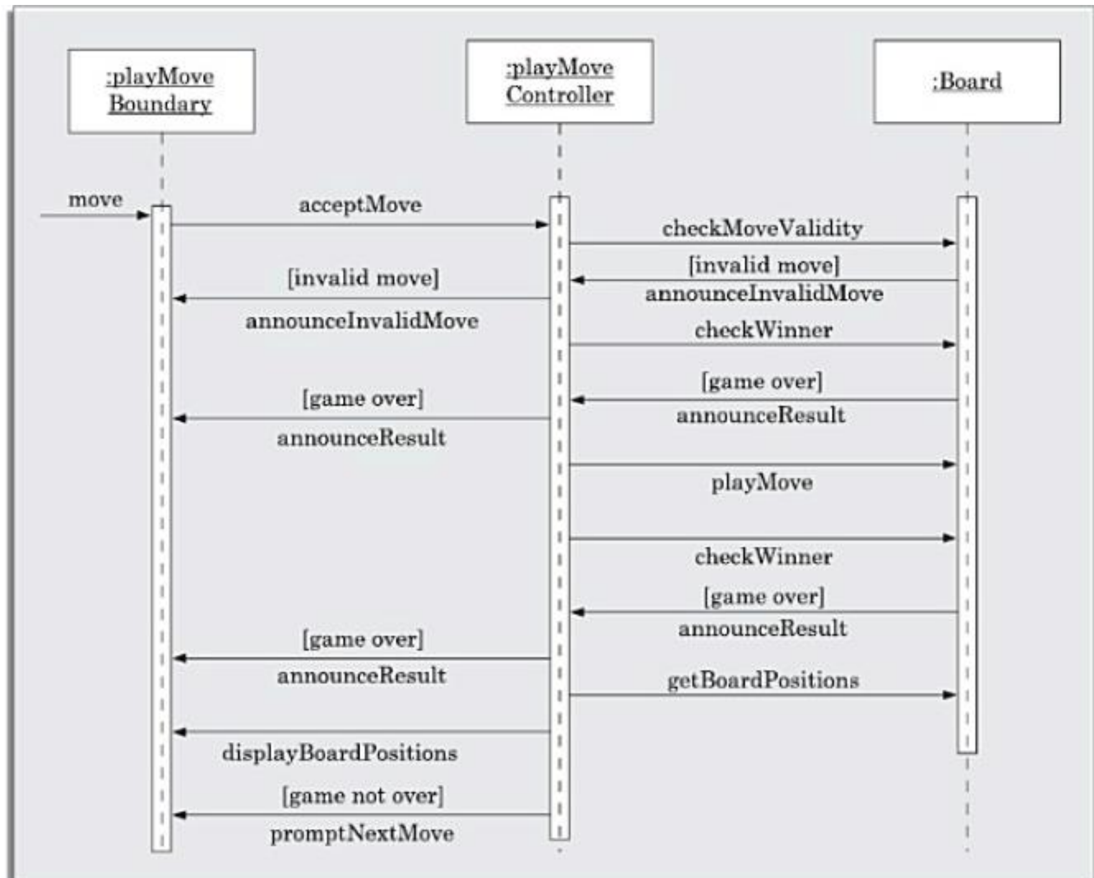


Gambar 8.12 Model kasus penggunaan untuk Contoh 8.2.

- Diagram interaksi untuk use case play move ditunjukkan pada Gambar 8.14. Perhatikan bahwa pola pakar digunakan saat menentukan kelas mana yang harus bertanggung jawab untuk (yaitu, berisi) metode checkWinner, pengontrol atau kelas batas? Kelas Dewan memiliki semua informasi yang diperlukan, berdasarkan mana pemenang dapat ditentukan. Oleh karena itu dibuatlah kelas papan untuk mendukung metode checkWinner.

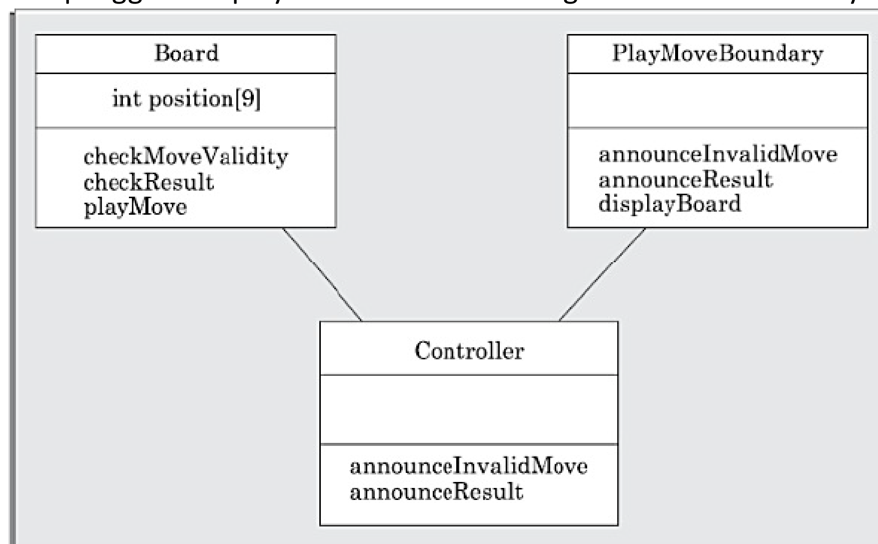


Gambar 8.13 (a) Model domain awal (b) Model domain halus untuk Contoh 8.2.



Gambar 8.14 Diagram urutan untuk kasus penggunaan play move pada Contoh 8.2.

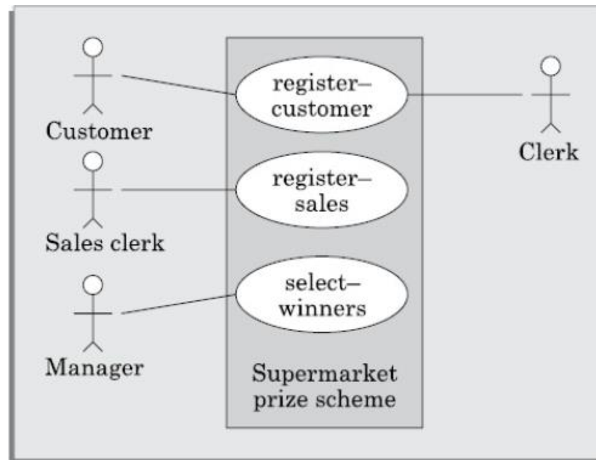
- Diagram kelas ditunjukkan pada Gambar 8.15. Pesan dari diagram interaksi dari kasus penggunaan play move telah diisi sebagai metode dari kelas yang sesuai.



Gambar 8.15 Diagram kelas untuk Contoh 8.2.

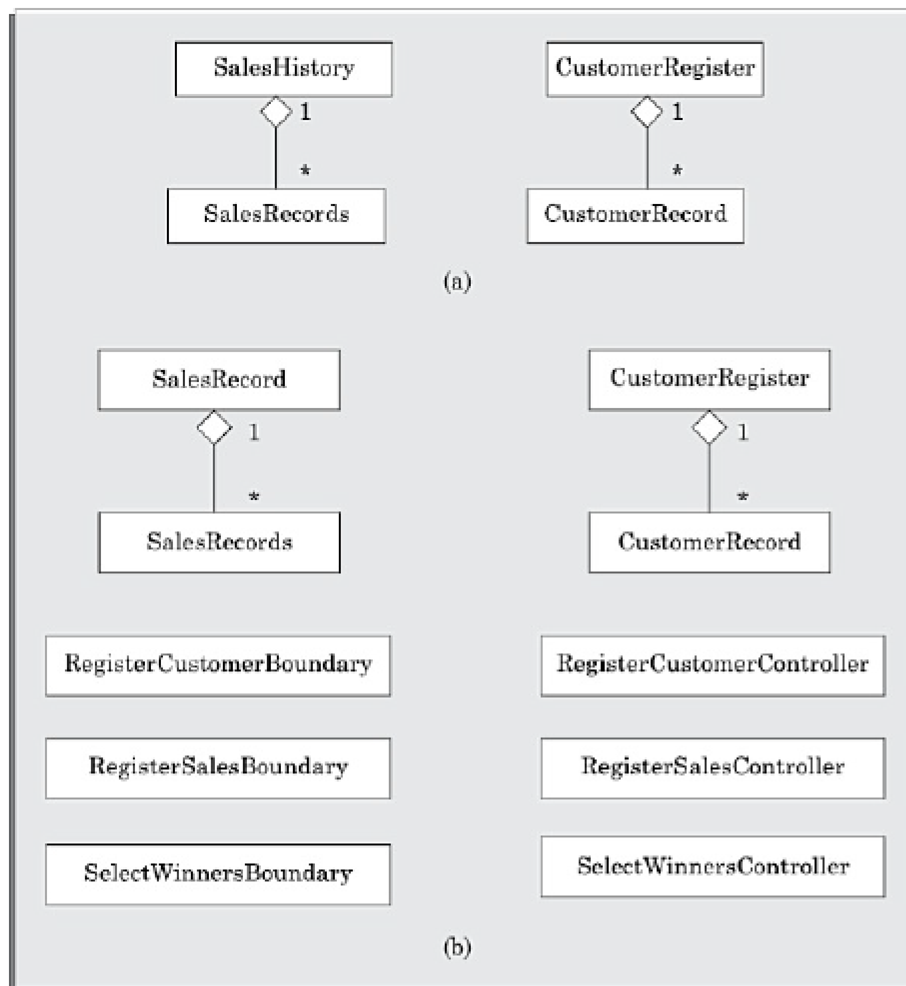
Contoh 8.3 Pertimbangkan perangkat lunak skema hadiah Supermarket yang dibahas dalam Contoh 6.3. Analisis langkah-demi-langkah dan latihan desain dari masalah ini adalah sebagai berikut:

- Model use case ditunjukkan pada Gambar 8.16.



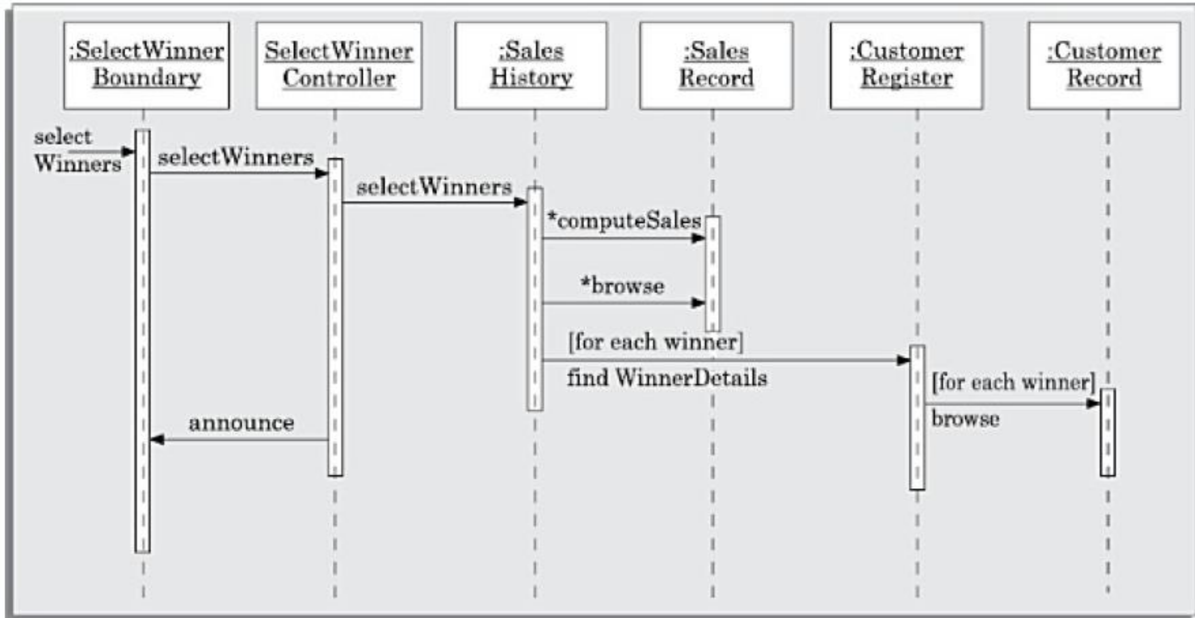
Gambar 8.16 Model kasus penggunaan untuk Contoh 8.3.

- Model domain awal ditunjukkan pada Gambar 8.17 (a).
- Model domain setelah menambahkan kelas batas dan kontrol ditunjukkan pada Gambar 8.17 (b).



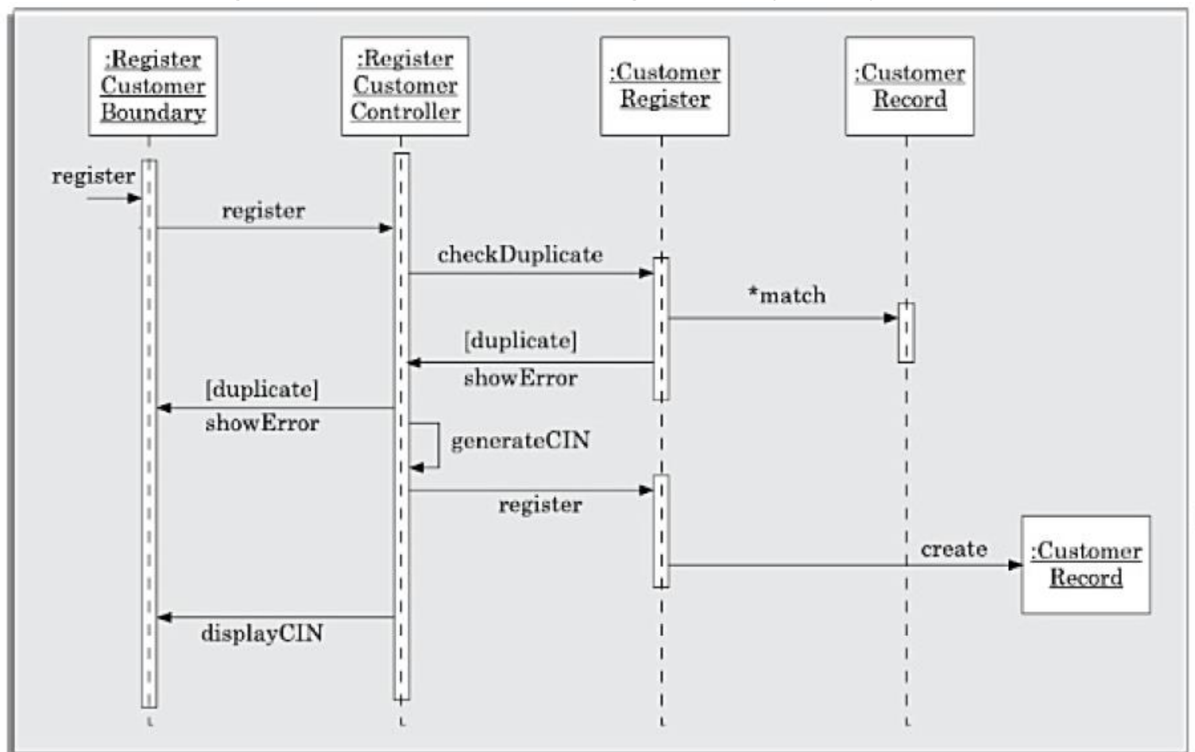
Gambar 8.17 (a) Model domain awal (b) Model domain halus untuk Contoh 8.3.

- Diagram urutan untuk use case daftar pemenang terpilih Gambar 8.18.



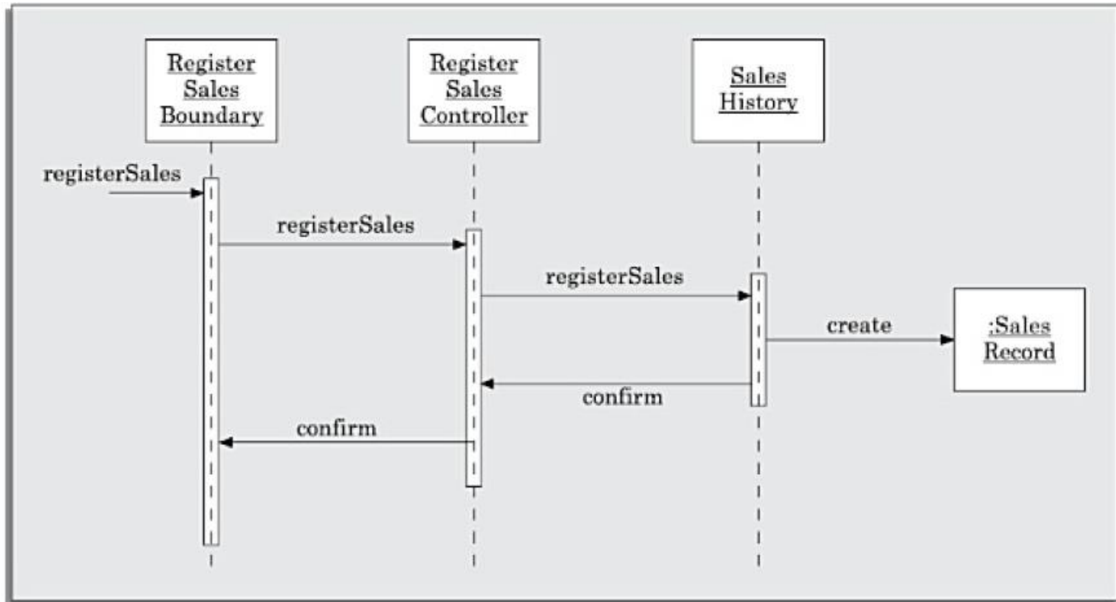
Gambar 8.18 Diagram urutan untuk kasus penggunaan daftar pemenang terpilih dari Contoh 8.3.

- Diagram urutan untuk use case register ditunjukkan pada Gambar 8.19.

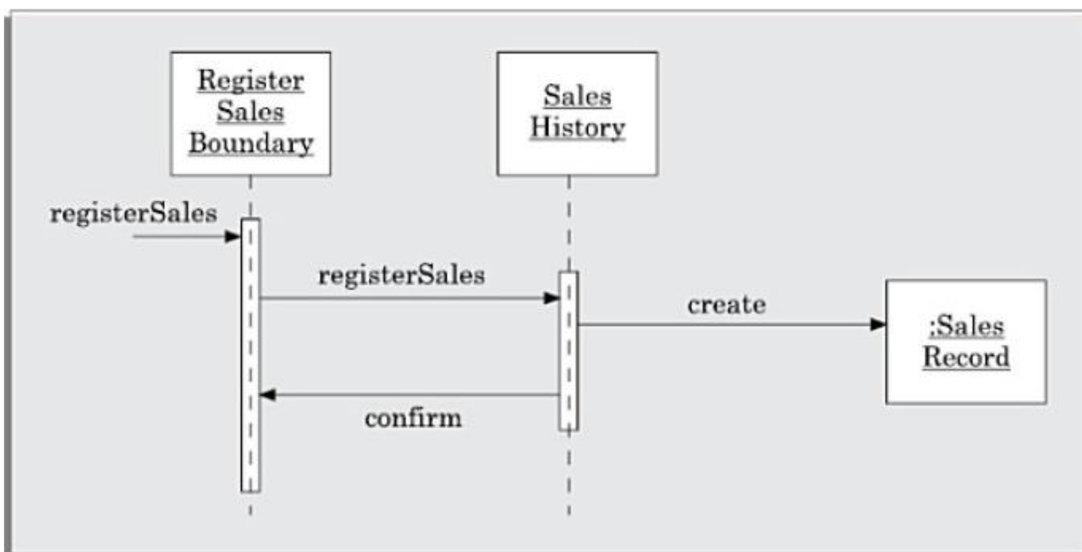


Gambar 8.19 Diagram urutan untuk kasus penggunaan pelanggan register Contoh 8.3.

- Diagram urutan use case register penjualan Gambar 8.20. Dalam kasus penggunaan ini, karena tanggung jawab RegisterSalesController sepele, kelas pengontrol dapat dihapus dan diagram urutan telah digambar ulang pada Gambar 8.21 setelah memasukkan perubahan ini.

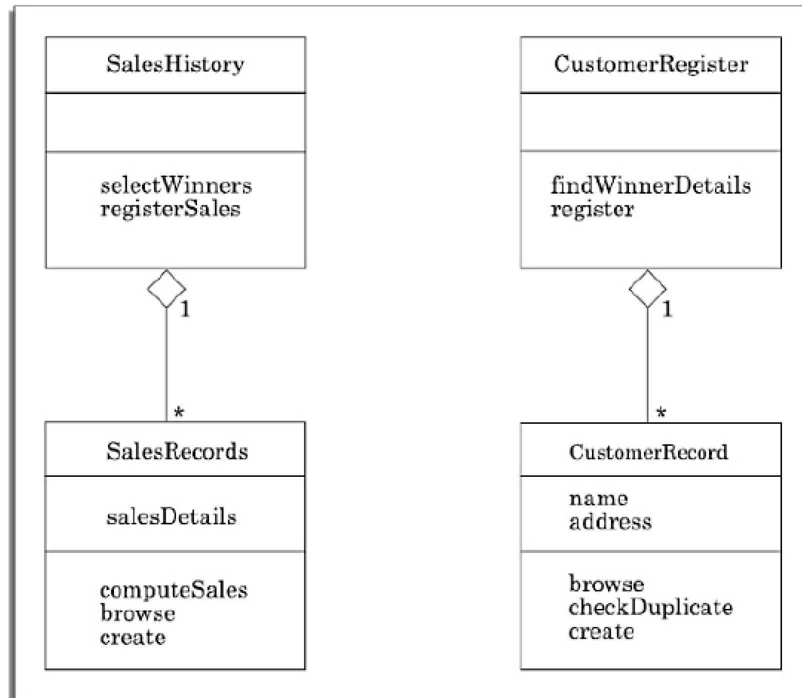


Gambar 8.20 Diagram urutan untuk kasus penggunaan register penjualan Contoh 8.3.



Gambar 8.21 Diagram urutan yang disempurnakan untuk kasus penggunaan register penjualan pada Contoh 8.3.

- Diagram kelas ditunjukkan pada Gambar 8.22. Amati bahwa pesan dari diagram urutan (Gbr. 8.18 hingga 8.20) dari kasus penggunaan yang berbeda telah diisi sebagai metode dari kelas yang sesuai.



Gambar 8.22 Diagram kelas untuk Contoh 8.3.

8.12 KRITERIA KEBAIKAN

Kita telah melihat bahwa beberapa penilaian subjektif dibuat saat sampai pada solusi desain berorientasi objek (OOD). Tergantung pada penilaian yang tepat, beberapa solusi desain alternatif untuk masalah yang sama dimungkinkan. Untuk dapat menentukan mana dari beberapa alternatif solusi desain yang lebih baik, kita perlu mengidentifikasi beberapa kriteria untuk menentukan mana dari dua alternatif desain yang lebih baik. Berikut ini adalah beberapa kriteria yang diterima untuk menilai kebaikan suatu desain:

Kopling: Kopling yang berlebihan antara objek merusak desain modular dan mencegah penggunaan kembali. Jumlah pesan antara dua objek atau di antara sekelompok objek harus minimum. Peningkatan jumlah pertukaran pesan antara dua objek menghasilkan peningkatan kopling di antara mereka.

Kohesi: Kohesi dalam desain harus tinggi. Dalam OOD, kohesi pada tiga tingkatan adalah sebagai berikut:

Kekompakan metode individu: Kekompakan setiap metode individu dalam kelas diinginkan. Ini mensyaratkan bahwa setiap metode harus melakukan hanya fungsi tertentu yang terdefinisi dengan baik. Ini diinginkan karena memastikan bahwa metode suatu objek melakukan tindakan yang menjadi tanggung jawab objek secara alami, yaitu, memastikan bahwa tidak ada tindakan yang dipetakan secara tidak tepat ke objek.

Kekompakan data dan metode di dalam kelas: Kelas-kelas dalam hierarki harus koheren. Misalnya, dari kelas dasar yang Dapat Diterbitkan di perpustakaan, hanya item yang dapat diterbitkan yang harus diturunkan. Akan sangat tidak tepat untuk menurunkan kelas LibraryMember dari kelas Issuable.

Keterpaduan dari seluruh hierarki kelas: Kepaduan metode dalam kelas diinginkan karena mempromosikan enkapsulasi objek.

Pedoman hierarki dan pemfaktoran: Kelas dasar tidak boleh memiliki terlalu banyak subkelas. Jika terlalu banyak subclass yang diturunkan dari satu kelas dasar, maka akan sulit untuk memahami desainnya. Bahkan, seharusnya tidak lebih dari 7 ± 2 kelas yang diturunkan dari kelas dasar di tingkat mana pun.

Menjaga protokol pesan tetap sederhana : Protokol pesan yang kompleks merupakan indikasi kopling berlebihan di antara objek. Jika sebuah pesan membutuhkan lebih dari 3 parameter, maka itu merupakan indikasi dari desain yang lebih rendah.

Jumlah metode: Kelas yang memiliki banyak metode cenderung lebih spesifik untuk aplikasi dan juga sulit untuk dipahami—membatasi kemungkinan penggunaannya kembali. Oleh karena itu, kelas tidak boleh memiliki terlalu banyak metode. Bahkan, jumlah metode dapat digunakan sebagai ukuran kompleksitas suatu kelas. Mempertahankan dan men-debug kelas yang memiliki lebih dari sekitar tujuh metode dapat menjadi masalah.

Kedalaman pohon pewarisan: Semakin dalam suatu kelas dalam hierarki pewarisan kelas, semakin besar jumlah metode yang mungkin diwarisinya. Oleh karena itu, tinggi pohon warisan tidak boleh terlalu besar. Biasanya, kompleksitas kelas dalam hierarki kelas meningkat dari kelas akar (basis) menuju kelas daun dalam hierarki kelas.

Jumlah pesan per use case: Jumlah pesan per use case bisa tinggi, ketika beberapa objek berinteraksi dengan menghasilkan terlalu banyak pesan atau sejumlah besar objek berinteraksi selama eksekusi use case. Jika metode dari sejumlah besar objek dipanggil dalam tindakan berantai sebagai respons terhadap satu pesan, pengujian dan debugging menjadi rumit. Jika eksekusi use case menunjukkan kegagalan, mendeteksi kesalahan (bug) akan sangat sulit karena kesalahan berpotensi ada di salah satu objek yang berpartisipasi. Selanjutnya, mungkin ada terlalu banyak overhead yang terjadi karena pengiriman pesan dan kelas yang berbeda mungkin melakukan pekerjaan yang terlalu sedikit. Setiap objek pada dasarnya "menghasilkan uang," daripada melakukan pekerjaan yang berarti. Oleh karena itu, satu kasus penggunaan tidak boleh menghasilkan generasi dan transmisi pesan yang berlebihan dalam suatu sistem.

***Response for a class* (RFC):** Response for a class (RFC) didefinisikan sebagai jumlah maksimum metode dan konstruktor dari objek lain yang dipanggil oleh instance dari kelas ini. Harap dicatat bahwa jika metode yang sama dipanggil lebih dari sekali, maka itu dihitung hanya sekali. Kelas yang memanggil lebih dari sekitar tujuh metode berbeda rentan terhadap kesalahan.

8.13 RINGKASAN

- Pola adalah solusi yang dapat digunakan kembali untuk masalah yang berulang. Kita dapat melihat pola dalam suatu masalah jika kita terbiasa dengannya.
- Analisis berorientasi objek generik dan proses desain. Proses analisis dan desain ini membutuhkan konstruksi *use case* dan *model domain* terlebih dahulu dan kemudian iterasi melalui diagram interaksi untuk mendapatkan diagram kelas akhir.
- Desain yang baik hanya dapat diperoleh melalui beberapa iterasi, mencoba beberapa alternatif solusi. Praktek memecahkan sejumlah besar masalah sangat berguna dalam mencapai desain yang baik untuk suatu masalah.
- Karakteristik solusi desain yang baik dan mendefinisikan beberapa metrik yang dapat digunakan untuk menilai solusi desain.

8.14 LATIHAN

1. Pilih opsi yang benar:
   a. Saat menggunakan deskripsi informal (bahasa alami) dari masalah pemrograman, bagian mana dari deskripsi yang akan mewakili objek?
      i. Semua kata benda dan beberapa kata kerja.
      ii. Semua kata kerja dan beberapa kata benda.

   iii. Beberapa kata benda.
   iv. Beberapa kata kerja
  b. Manakah dari parameter berikut untuk kelas yang berkorelasi positif dengan kualitas solusi desain:
   i. Respon untuk kelas (RFC)
   ii. Kedalaman pohon pewarisan (DIT)
   iii. Jumlah pesan per kasus penggunaan
   iv. Kekompakan metode iklan data dalam sebuah kelas
  c. Manakah dari berikut ini yang merupakan karakteristik dari desain berorientasi objek yang baik:
   i. Hirarki kelas dalam
   ii. Sejumlah besar metode per kelas
   iii. Sejumlah besar pertukaran pesan per kasus penggunaan
   iv. Jumlah metode sedang per kelas
2. Apa perbedaan antara analisis berorientasi objek (OOA) dan desain berorientasi objek (OOD)?
3. Apa yang dimaksud dengan pola dalam konteks pengembangan perangkat lunak? Jelaskan mengapa pola dianggap sebagai bentuk penggunaan kembali perangkat lunak yang efektif. Apa batasan pendekatan ini untuk penggunaan kembali perangkat lunak?
4. Masalah apa yang coba dipecahkan oleh pola desain fasad? Solusi apa yang ditawarkannya?
5. Apakah Anda perlu mengembangkan semua tampilan sistem menggunakan semua diagram pemodelan yang didukung oleh UML? Justifikasi jawaban Anda.
6. Apa itu pola desain? Apa keuntungan menggunakan pola desain? Sebutkan beberapa pola desain populer.
7. Apa itu proses terpadu? Apa saja fase-fase yang berbeda dari proses terpadu? Kegiatan apa yang dilakukan selama setiap fase dari proses terpadu?
8. Algoritma seperti penyortiran terkenal sebagai solusi yang dapat digunakan kembali untuk masalah. Karena pola memberikan solusi yang dapat digunakan kembali, apakah benar untuk mengatakan bahwa pola pada dasarnya adalah algoritma? Jelaskan jawaban Anda.
9. Mengapa model interaksi "dorong dari bawah" antara objek GUI dengan pengontrol atau objek domain bukanlah ide yang baik? Solusi apa yang ditawarkan pola MVC dalam hal ini?
10. Nyatakan apakah pernyataan berikut BENAR atau SALAH. Berikan alasan di balik jawaban Anda.
  a. Hirarki kelas dalam adalah tanda dari desain berorientasi objek yang dilakukan dengan baik.
  b. Sejumlah besar pertukaran pesan antar objek selama realisasi use case menunjukkan pendelegasian tanggung jawab yang efektif dan merupakan tanda desain yang baik.
  c. Kesulitan memahami kelas dalam hierarki kelas meningkat dari akar ke daun dalam hierarki kelas.
11. Definisikan istilah kohesi dalam konteks desain perangkat lunak berorientasi objek.
12. Apa saja kriteria penting yang menjadi dasar untuk menentukan mana dari dua solusi desain berorientasi objek untuk suatu masalah yang lebih baik?
13. Jelaskan perbedaan antara pola arsitektur, pola desain, dan idiom dalam konteks pengembangan perangkat lunak berorientasi objek.

14. Apa itu anti-pola? Bagaimana antipattern membantu dalam mencapai solusi desain yang baik untuk suatu masalah?
15. Secara singkat menguraikan langkah-langkah penting yang terlibat dalam mengembangkan sistem perangkat lunak menggunakan metodologi desain berorientasi objek yang populer.
16. Melakukan desain berorientasi objek untuk pengembangan perangkat lunak otomatisasi Hotel (masalah nomor 6.13) dari Bab 6.
17. Melakukan desain berorientasi objek untuk pengembangan Perangkat Lunak Perbaikan dan Pelacakan Jalan (RRTS) (masalah nomor 6.15) dari Bab 6.
18. Melakukan desain berorientasi objek untuk pengembangan Toko Buku Automation Software (BAS) (masalah nomor 6.14) dari Bab 6.
19. Melakukan perancangan berorientasi objek untuk pengembangan perangkat lunak *Library Information System* (LIS) (soal nomor 6.18) Bab 6.
20. Lakukan desain berorientasi objek untuk mengembangkan perangkat lunak simulasi berikut: Sebuah pabrik memiliki kategori mesin tertentu yang memerlukan penyesuaian dan perbaikan yang sering. Setiap kategori mesin gagal secara seragam setelah operasi terus menerus dan profil kegagalan dari berbagai kategori mesin diberikan oleh *mean time to failure* (MTTF). Sejumlah adjuster digunakan untuk menjaga agar mesin tetap berjalan. Seorang manajer layanan mengoordinasikan kegiatan para penyesuai. Manajer layanan mempertahankan antrian mesin yang tidak beroperasi. Jika ada mesin yang menunggu untuk diperbaiki, manajer layanan menugaskan mesin di depan antrian ke penyetel berikutnya yang tersedia. Demikian juga, ketika beberapa penyetel tidak sibuk, manajer layanan mempertahankan antrian penyetel yang menganggur dan menugaskan penyetel di depan antrian ke mesin berikutnya yang rusak. Pada waktu tertentu, salah satu dari dua antrian akan kosong. Dengan demikian, manajer layanan hanya perlu memelihara satu antrian, yang bila tidak kosong hanya berisi mesin atau pengatur saja. Manajemen pabrik ingin mengeluarkan sebanyak mungkin mesin dan penyetelnya. Oleh karena itu tertarik pada pemanfaatan mesin—persentase waktu mesin menyala dan berjalan dan penggunaan penyetel—persentase waktu penyetel sibuk. Tujuan dari simulasi ini adalah untuk melihat bagaimana rata-rata penggunaan mesin dan adjuster bergantung pada faktor-faktor seperti jumlah mesin, jumlah adjuster, keandalan mesin dalam hal mean time to failure (MTTF), dan produktivitas. dari adjuster.
21. Perhatikan Masalah Kontrol Lift berikut ini. Sistem perangkat lunak (Elevator Controller) harus mengontrol satu set 4 elevator untuk gedung dengan 10 lantai. Setiap lift berisi satu set tombol, masing-masing sesuai dengan lantai yang diinginkan. Ini disebut tombol permintaan lantai, karena mereka menunjukkan permintaan untuk pergi ke lantai tertentu. Setiap lift juga memiliki indikator lantai saat ini di atas pintu. Setiap lantai memiliki dua tombol untuk meminta lift yang disebut tombol permintaan lift karena mereka meminta lift. Kontroler lift akan menerima semua sinyal dari penumpang dan memutuskan tindakan kontrol yang akan diumpankan ke lift. Setiap lantai memiliki pintu geser untuk setiap poros yang diatur sehingga dua bagian pintu bertemu di tengah saat ditutup. Ketika lift tiba di lantai, pintu terbuka bersamaan dengan pintu lift terbuka. Lantai memang memiliki sensor tekanan dan optik untuk mencegah penutupan saat ada penghalang di antara dua bagian pintu. Jika halangan terdeteksi oleh salah satu sensor, pintu harus terbuka. Pintu akan menutup secara otomatis setelah jangka waktu lima detik setelah pintu terbuka. Deteksi suatu penghalang harus memulai kembali waktu penutupan pintu setelah suatu penghalang

dihilangkan. Ada speaker di setiap lantai yang mengumumkan kedatangan lift. Di setiap lantai, ada dua tombol permintaan lift, satu untuk NAIK dan satu lagi untuk TURUN. Di setiap lantai di atas setiap pintu lift, ada indikator yang menentukan lantai lift saat ini dan indikator lain untuk arahnya saat ini. Sistem akan menanggapi permintaan lift dengan mengirimkan lift terdekat yang sedang menganggur atau sudah menuju ke arah yang diminta. Jika tidak ada lift saat ini adalah kriteria yang disebutkan sebelumnya. Setelah ditekan, tombol permintaan akan menyala untuk menunjukkan bahwa permintaan tertunda. Menekan tombol permintaan lift ketika permintaan untuk arah itu sudah tertunda, tidak akan berpengaruh. Ketika lift tiba untuk menangani permintaan (yaitu, dijadwalkan untuk pergi ke arah yang dipilih), maka pintu akan berhenti menutup dan pengatur waktu penutupan pintu akan diatur ulang. Untuk meningkatkan keamanan, sensor tegangan kabel memantau tegangan pada kabel yang mengontrol lift. Jika terjadi kegagalan (diukur tegangan turun di bawah nilai kritis), kemudian empat klem pengunci eksternal yang terhubung ke lintasan lari di poros menghentikan elevator dan menahannya di tempatnya.

a. Kembangkan model domain.
b. Kembangkan model bagan keadaan untuk kelas yang memiliki jumlah keadaan dan perilaku yang signifikan.

BAB 9
DESAIN USER INTERFACE

Bagian antarmuka pengguna dari produk perangkat lunak bertanggung jawab atas semua interaksi dengan pengguna. Hampir setiap produk perangkat lunak memiliki antarmuka pengguna (dapatkah Anda memikirkan produk perangkat lunak yang tidak memiliki antarmuka pengguna?). Pada hari-hari awal komputer, tidak ada produk perangkat lunak yang memiliki antarmuka pengguna. Komputer saat itu adalah sistem batch dan tidak ada interaksi dengan pengguna yang didukung. Sekarang, kita tahu bahwa segala sesuatunya sangat berbeda—hampir setiap produk perangkat lunak sangat interaktif. Bagian antarmuka pengguna dari produk perangkat lunak bertanggung jawab atas semua interaksi dengan pengguna akhir. Akibatnya, bagian antarmuka pengguna dari setiap produk perangkat lunak menjadi perhatian langsung pengguna akhir. Tak heran jika kemudian banyak pengguna sering menilai suatu produk perangkat lunak berdasarkan antarmuka penggunanya. Selain estetika, antarmuka yang sulit digunakan menyebabkan tingkat kesalahan pengguna yang lebih tinggi dan pada akhirnya menyebabkan ketidakpuasan pengguna. Pengguna menjadi sangat kesal ketika sistem berperilaku dengan cara yang tidak terduga, yaitu, perintah yang dikeluarkan tidak melakukan tindakan sesuai dengan harapan intuitif pengguna. Biasanya, ketika pengguna mulai menggunakan sistem, ia membangun model mental sistem dan mengharapkan perilaku sistem untuk menyesuaikannya. Misalnya, jika tindakan pengguna menyebabkan satu jenis aktivitas dan respons sistem dalam beberapa konteks, maka pengguna akan mengharapkan aktivitas dan respons sistem yang serupa terjadi untuk tindakan pengguna yang serupa dalam konteks yang serupa. Oleh karena itu, perhatian dan perhatian yang memadai harus diberikan pada desain antarmuka pengguna produk perangkat lunak apa pun.

Pengembangan antarmuka pengguna yang sistematis juga penting dari pertimbangan lain. Pengembangan antarmuka pengguna yang baik biasanya membutuhkan porsi yang signifikan dari total upaya pengembangan sistem. Untuk banyak aplikasi interaktif, sebanyak 50 persen dari total upaya pengembangan dihabiskan untuk mengembangkan bagian antarmuka pengguna. Kecuali jika antarmuka pengguna dirancang dan dikembangkan secara sistematis, upaya total yang diperlukan untuk mengembangkan antarmuka akan meningkat pesat. Oleh karena itu, perlu mempelajari dengan cermat berbagai konsep yang terkait dengan desain antarmuka pengguna dan memahami berbagai teknik sistematis yang tersedia untuk pengembangan antarmuka pengguna.

Dalam bab ini, pertama-tama kita membahas beberapa terminologi dan konsep umum yang terkait dengan pengembangan antarmuka pengguna, lalu mengklasifikasikan berbagai jenis antarmuka yang biasa digunakan. Disini juga diberikan beberapa panduan untuk merancang antarmuka yang baik, dan membahas beberapa alat untuk pengembangan antarmuka pengguna grafis (GUI). Terakhir adalah metodologi pengembangan GUI.

9.1 KARAKTERISTIK USER INTERFACE YANG BAIK

Sebelum kita mulai membahas apa pun tentang bagaimana mengembangkan antarmuka pengguna, penting untuk mengidentifikasi karakteristik berbeda yang biasanya diinginkan dari antarmuka pengguna yang baik. Kecuali kita tahu persis apa yang diharapkan dari antarmuka pengguna yang baik, kita tidak mungkin mendesainnya. Ada beberapa karakteristik penting dari antarmuka pengguna yang baik, diantaranya adalah:

Kecepatan belajar: Antarmuka pengguna yang baik harus mudah dipelajari. Kecepatan belajar terhambat oleh sintaks dan semantik yang kompleks dari prosedur masalah perintah. Antarmuka pengguna yang baik seharusnya tidak mengharuskan penggunanya untuk menghafal perintah. Pengguna juga tidak boleh diminta untuk mengingat informasi dari satu layar ke layar lainnya saat melakukan berbagai tugas menggunakan antarmuka. Selain itu, tiga hal berikut ini sangat penting untuk meningkatkan kecepatan belajar:

- Penggunaan metafora1 dan nama perintah intuitif: Kecepatan mempelajari antarmuka sangat difasilitasi jika ini didasarkan pada beberapa contoh kehidupan nyata sehari-hari atau beberapa objek fisik yang akrab dengan pengguna. Abstraksi objek atau konsep kehidupan nyata yang digunakan dalam desain antarmuka pengguna disebut metafora. Jika antarmuka pengguna editor teks menggunakan konsep yang mirip dengan alat yang digunakan oleh penulis untuk mengedit teks seperti memotong garis dan paragraf dan menempelkannya di tempat lain, pengguna dapat langsung menghubungkannya. Metafora populer lainnya adalah keranjang belanja. Semua orang tahu bagaimana keranjang belanja digunakan untuk membuat pilihan saat membeli barang di supermarket. Jika antarmuka pengguna menggunakan metafora keranjang belanja untuk merancang gaya interaksi untuk situasi di mana jenis pilihan yang serupa harus dibuat, maka pengguna dapat dengan mudah memahami dan belajar menggunakan antarmuka. Juga, pembelajaran difasilitasi oleh nama perintah intuitif dan prosedur masalah perintah simbolis.
- Konsistensi: Sekali, seorang pengguna mempelajari tentang suatu perintah, ia harus dapat menggunakan perintah yang serupa dalam situasi yang berbeda untuk melakukan tindakan yang serupa. Hal ini memudahkan untuk mempelajari antarmuka karena pengguna dapat memperluas pengetahuannya tentang satu bagian antarmuka ke bagian lain. Dengan demikian, perintah yang berbeda yang didukung oleh antarmuka harus konsisten.
- Antarmuka berbasis komponen: Pengguna dapat mempelajari antarmuka lebih cepat jika gaya interaksi antarmuka sangat mirip dengan antarmuka aplikasi lain yang sudah dikenal pengguna. Ini dapat dicapai jika antarmuka aplikasi yang berbeda dikembangkan menggunakan beberapa komponen antarmuka pengguna standar.

Kecepatan karakteristik pembelajaran antarmuka pengguna dapat ditentukan dengan mengukur waktu pelatihan dan praktik yang dibutuhkan pengguna sebelum mereka dapat menggunakan perangkat lunak secara efektif.

Kecepatan penggunaan: Kecepatan penggunaan antarmuka pengguna ditentukan oleh waktu dan upaya pengguna yang diperlukan untuk memulai dan menjalankan perintah yang berbeda. Karakteristik antarmuka ini kadang-kadang disebut sebagai dukungan produktivitas antarmuka. Ini menunjukkan seberapa cepat pengguna dapat melakukan tugas yang diinginkan. Waktu dan upaya pengguna yang diperlukan untuk memulai dan menjalankan perintah yang berbeda harus minimal. Ini dapat dicapai melalui desain antarmuka yang cermat. Misalnya, antarmuka yang mengharuskan pengguna mengetikkan perintah yang panjang atau melibatkan gerakan mouse ke berbagai area layar yang terpisah lebar untuk mengeluarkan perintah dapat memperlambat kecepatan pengoperasian pengguna. Perintah yang paling sering digunakan harus memiliki panjang terkecil atau tersedia di bagian atas menu untuk meminimalkan gerakan mouse yang diperlukan untuk mengeluarkan perintah.

Kecepatan mengingat: Setelah pengguna mempelajari cara menggunakan antarmuka, kecepatan mengingat prosedur masalah perintah harus dimaksimalkan. Karakteristik ini sangat penting bagi pengguna intermiten. Kecepatan mengingat ditingkatkan jika antarmuka didasarkan pada beberapa metafora, prosedur masalah perintah simbolis, dan nama perintah intuitif.

Pencegahan kesalahan: Antarmuka pengguna yang baik harus meminimalkan ruang lingkup melakukan kesalahan saat memulai perintah yang berbeda. Tingkat kesalahan antarmuka dapat dengan mudah ditentukan dengan memantau kesalahan yang dilakukan oleh rata-rata pengguna saat menggunakan antarmuka. Pemantauan ini dapat diotomatisasi dengan melengkapi kode antarmuka pengguna dengan kode pemantauan yang dapat merekam frekuensi dan jenis kesalahan pengguna dan kemudian menampilkan statistik berbagai jenis kesalahan yang dilakukan oleh pengguna yang berbeda. Konsistensi nama, prosedur masalah, dan perilaku perintah serupa dan kesederhanaan prosedur masalah perintah meminimalkan kemungkinan kesalahan. Selain itu, antarmuka harus mencegah pengguna memasukkan nilai yang salah.

Estetika dan menarik: Antarmuka pengguna yang baik harus menarik untuk digunakan. Antarmuka pengguna yang menarik menarik perhatian pengguna dan mewah. Dalam hal ini, antarmuka pengguna berbasis grafik memiliki keunggulan yang pasti dibandingkan antarmuka berbasis teks.

Konsistensi: Perintah yang didukung oleh antarmuka pengguna harus konsisten. Tujuan dasar dari konsistensi adalah untuk memungkinkan pengguna untuk menggeneralisasi pengetahuan tentang aspek antarmuka dari satu bagian ke bagian lain. Dengan demikian, konsistensi memfasilitasi kecepatan belajar, kecepatan mengingat, dan juga membantu mengurangi tingkat kesalahan

Umpan Balik: Antarmuka pengguna yang baik harus memberikan umpan balik ke berbagai tindakan pengguna. Terutama, jika ada permintaan pengguna yang membutuhkan waktu lebih dari beberapa detik untuk diproses, pengguna harus diberi tahu tentang status pemrosesan permintaannya. Dengan tidak adanya respons dari komputer untuk waktu yang lama, pengguna pemula bahkan mungkin memulai prosedur pemulihan/penonaktifan dengan panik. Jika diperlukan, pengguna harus diberitahu secara berkala tentang kemajuan yang dibuat dalam memproses perintahnya.

Dukungan untuk beberapa tingkat keahlian: Antarmuka pengguna yang baik harus mendukung berbagai tingkat kecanggihan prosedur masalah perintah untuk berbagai kategori pengguna. Hal ini diperlukan karena pengguna dengan tingkat pengalaman yang berbeda dalam menggunakan aplikasi lebih menyukai jenis antarmuka pengguna yang berbeda. Pengguna berpengalaman lebih memperhatikan efisiensi prosedur masalah perintah, sedangkan pengguna pemula memperhatikan aspek kegunaan. Perintah yang sangat samar dan kompleks membuat pemula enggan, sedangkan urutan perintah yang rumit membuat prosedur masalah perintah menjadi sangat lambat dan karena itu menunda pengguna berpengalaman. Ketika seseorang menggunakan aplikasi untuk pertama kalinya, perhatian utamanya adalah kecepatan belajar. Setelah menggunakan aplikasi untuk waktu yang lama, ia menjadi terbiasa dengan pengoperasian perangkat lunak. Saat pengguna menjadi semakin akrab dengan antarmuka, fokusnya beralih dari aspek kegunaan ke aspek kecepatan masalah perintah. Pengguna berpengalaman mencari opsi seperti "hot-keys", "makro", dll. Dengan demikian, tingkat keterampilan pengguna meningkat karena mereka terus menggunakan produk perangkat lunak dan mereka mencari perintah yang sesuai dengan tingkat keterampilan mereka.

Pemulihan kesalahan (membatalkan fasilitas): Saat mengeluarkan perintah, bahkan pengguna ahli pun dapat melakukan kesalahan. Oleh karena itu, antarmuka pengguna yang baik harus memungkinkan pengguna untuk membatalkan kesalahan yang dilakukan olehnya saat menggunakan antarmuka. Pengguna merasa tidak nyaman jika mereka tidak dapat memulihkan dari kesalahan yang mereka lakukan saat menggunakan perangkat lunak. Jika pengguna tidak dapat memulihkan bahkan dari jenis kesalahan yang sangat sederhana, mereka merasa kesal, tidak berdaya, dan di luar kendali.

Panduan pengguna dan bantuan online: Pengguna mencari panduan dan bantuan online ketika mereka lupa perintah atau tidak mengetahui beberapa fitur perangkat lunak. Kapan pun pengguna membutuhkan panduan atau mencari bantuan dari sistem, mereka harus diberikan panduan dan bantuan yang sesuai.

9.2 KONSEP DASAR

Pada bagian ini, pertama-tama kita membahas beberapa konsep dasar dalam panduan pengguna dan sistem bantuan online. Selanjutnya adalah memeriksa konsep antarmuka berbasis mode dan tanpa mode dan keuntungan dari antarmuka grafis.

Panduan Pengguna dan Bantuan Online

Pengguna dapat mencari bantuan tentang pengoperasian perangkat lunak kapan saja saat menggunakan perangkat lunak. Ini disediakan oleh sistem bantuan online. Ini berbeda dengan panduan dan pesan kesalahan yang di-flash secara otomatis tanpa diminta oleh pengguna. Pesan panduan meminta pengguna mengenai opsi yang dia miliki mengenai perintah berikutnya, dan status perintah terakhir, dll.

Sistem bantuan online: Pengguna mengharapkan pesan bantuan online disesuaikan dengan konteks di mana mereka memanggil "sistem bantuan". Oleh karena itu, sistem bantuan online yang baik harus melacak apa yang dilakukan pengguna saat menjalankan sistem bantuan dan memberikan pesan keluaran dengan cara yang bergantung pada konteks. Selain itu, pesan bantuan harus disesuaikan dengan tingkat pengalaman pengguna. Selanjutnya, sistem bantuan online yang baik harus memanfaatkan karakteristik grafis dan animasi apa pun dari layar dan tidak boleh hanya berupa salinan manual pengguna.

Pesan panduan: Pesan panduan harus dirancang dengan hati-hati untuk mendorong pengguna tentang tindakan selanjutnya yang mungkin dia lakukan, status sistem saat ini, kemajuan yang dicapai sejauh ini dalam memproses perintah terakhirnya, dll. Sistem panduan yang baik harus memiliki tingkat yang berbeda kecanggihan untuk berbagai kategori pengguna. Misalnya, pengguna yang menggunakan antarmuka bahasa perintah mungkin memerlukan jenis panduan yang berbeda dibandingkan dengan pengguna yang menggunakan menu atau antarmuka ikonik (Jenis antarmuka yang berbeda ini akan dibahas nanti dalam bab ini). Selain itu, pengguna harus memiliki opsi untuk mematikan pesan terperinci.

Pesan kesalahan: Pesan kesalahan dihasilkan oleh sistem baik ketika pengguna melakukan kesalahan atau ketika beberapa kesalahan ditemukan oleh sistem selama pemrosesan karena beberapa kondisi luar biasa, seperti kehabisan memori, tautan komunikasi rusak, dll. Pengguna tidak menyukai kesalahan pesan yang ambigu atau terlalu umum seperti “input tidak valid atau kesalahan sistem”. Pesan kesalahan harus sopan. Pesan kesalahan seharusnya tidak memiliki gangguan terkait yang mungkin mempermalukan pengguna. Pesan tersebut harus menyarankan bagaimana kesalahan yang diberikan dapat diperbaiki. Jika sesuai, pengguna harus diberikan pilihan untuk menggunakan sistem bantuan online untuk mengetahui lebih lanjut tentang situasi kesalahan.

Antarmuka Berbasis Mode versus Tanpa Mode

Mode adalah keadaan atau kumpulan keadaan di mana hanya sebagian dari semua tugas interaksi pengguna yang dapat dilakukan. Dalam antarmuka tanpa mode, kumpulan perintah yang sama dapat dipanggil kapan saja selama menjalankan perangkat lunak. Dengan demikian, antarmuka tanpa mode hanya memiliki satu mode dan semua perintah tersedia sepanjang waktu selama pengoperasian perangkat lunak. Di sisi lain, dalam antarmuka berbasis mode, set perintah yang berbeda dapat dipanggil tergantung pada mode di mana sistem berada, yaitu, mode setiap saat ditentukan oleh urutan perintah yang sudah dikeluarkan oleh pengguna. Antarmuka berbasis mode dapat direpresentasikan menggunakan diagram transisi keadaan, di mana setiap simpul dari diagram transisi keadaan akan mewakili mode. Setiap keadaan dari diagram transisi keadaan

Antarmuka Pengguna Grafis (GUI) versus Antarmuka Pengguna Berbasis Teks

Mari kita bandingkan berbagai karakteristik GUI dengan antarmuka pengguna berbasis teks:

- Dalam GUI beberapa jendela dengan informasi yang berbeda secara bersamaan dapat ditampilkan di layar pengguna. Ini mungkin salah satu keuntungan terbesar GUI dibandingkan antarmuka berbasis teks karena pengguna memiliki fleksibilitas untuk berinteraksi secara bersamaan dengan beberapa item terkait kapan saja dan dapat memiliki akses ke informasi sistem yang berbeda yang ditampilkan di jendela yang berbeda.
- Representasi informasi ikonik dan manipulasi informasi simbolik dimungkinkan dalam GUI. Manipulasi informasi simbolik seperti menyeret ikon yang mewakili file ke tempat sampah untuk dihapus secara intuitif sangat menarik dan pengguna dapat langsung mengingatnya.
- GUI biasanya mendukung pemilihan perintah menggunakan sistem pemilihan menu yang menarik dan ramah pengguna.
- Dalam GUI, perangkat penunjuk seperti mouse atau pena ringan dapat digunakan untuk mengeluarkan perintah. Penggunaan perangkat penunjuk meningkatkan kemanjuran prosedur masalah perintah.
- Di sisi lain, GUI membutuhkan terminal khusus dengan kemampuan grafis untuk berjalan dan juga membutuhkan perangkat input khusus seperti mouse. Di sisi lain, antarmuka pengguna berbasis teks dapat diimplementasikan bahkan pada terminal tampilan alfanumerik yang murah. Terminal grafis biasanya jauh lebih mahal daripada terminal alfanumerik. Namun, terminal tampilan dengan kemampuan grafis dengan tampilan resolusi tinggi yang dipetakan dan sejumlah besar daya pemrosesan lokal telah menjadi terjangkau dan selama bertahun-tahun telah menggantikan terminal berbasis teks di semua desktop. Oleh karena itu, penekanan bab ini adalah pada desain GUI daripada desain antarmuka pengguna berbasis teks.

9.3 JENIS ANTARMUKA PENGGUNA

Secara garis besar, antarmuka pengguna dapat diklasifikasikan ke dalam tiga kategori berikut:

- Antarmuka berbasis bahasa perintah
- Antarmuka berbasis menu
- Antarmuka manipulasi langsung

Masing-masing kategori antarmuka ini memiliki kelebihan dan kekurangan karakteristiknya sendiri. Oleh karena itu, sebagian besar aplikasi modern menggunakan kombinasi yang cermat dari ketiga jenis antarmuka pengguna ini untuk mengimplementasikan

repertoar perintah pengguna. Sangat sulit untuk menghasilkan seperangkat pedoman sederhana tentang bagian antarmuka mana yang harus diimplementasikan menggunakan jenis antarmuka apa. Pilihan ini sebagian besar tergantung pada pengalaman dan kebijaksanaan perancang antarmuka. Namun, studi tentang karakteristik dasar dan keuntungan relatif dari berbagai jenis antarmuka akan memberikan ide yang adil kepada perancang mengenai perintah mana yang harus didukung menggunakan jenis antarmuka apa. Beberapa karakteristik penting, keuntungan, dan kerugiannya dalam menggunakan setiap jenis antarmuka pengguna.

Antarmuka Berbasis Bahasa Perintah

Antarmuka berbasis bahasa perintah—seperti namanya, didasarkan pada perancangan bahasa perintah yang dapat digunakan pengguna untuk mengeluarkan perintah. Pengguna diharapkan untuk membingkai perintah yang sesuai dalam bahasa dan mengetiknya dengan tepat kapan pun diperlukan. Antarmuka berbasis bahasa perintah sederhana mungkin hanya menetapkan nama unik untuk perintah yang berbeda. Namun, antarmuka berbasis bahasa perintah yang lebih canggih memungkinkan pengguna untuk membuat perintah kompleks dengan menggunakan seperangkat perintah primitif. Fasilitas untuk membuat perintah seperti itu secara dramatis mengurangi jumlah nama perintah yang harus diingat. Dengan demikian, antarmuka berbasis bahasa perintah dapat dibuat ringkas yang membutuhkan pengetikan minimal oleh pengguna. Antarmuka berbasis bahasa perintah memungkinkan interaksi cepat dengan komputer dan menyederhanakan input perintah yang kompleks.

Di antara tiga kategori antarmuka, antarmuka bahasa perintah memungkinkan prosedur masalah perintah yang paling efisien yang membutuhkan pengetikan minimal. Selanjutnya, antarmuka berbasis bahasa perintah dapat diimplementasikan bahkan pada terminal alfanumerik yang murah. Juga, antarmuka berbasis bahasa perintah lebih mudah untuk dikembangkan dibandingkan dengan antarmuka berbasis menu atau manipulasi langsung karena teknik penulisan kompiler dikembangkan dengan baik. Seseorang dapat secara sistematis mengembangkan antarmuka bahasa perintah dengan menggunakan alat penulisan kompiler standar Lex dan Yacc.

Namun, antarmuka berbasis bahasa perintah menderita beberapa kelemahan. Biasanya, antarmuka berbasis bahasa perintah sulit dipelajari dan mengharuskan pengguna untuk menghafal kumpulan perintah primitif. Selain itu, sebagian besar pengguna membuat kesalahan saat merumuskan perintah dalam bahasa perintah dan juga saat mengetiknya. Selanjutnya, dalam antarmuka berbasis bahasa perintah, semua interaksi dengan sistem dilakukan melalui papan tombol dan tidak dapat memanfaatkan perangkat interaksi yang efektif seperti mouse. Jelas, untuk pengguna biasa dan tidak berpengalaman, antarmuka berbasis bahasa perintah tidak cocok.

Masalah dalam merancang antarmuka berbasis bahasa perintah

Dua masalah desain perintah yang berlebihan adalah mengurangi jumlah perintah primitif yang harus diingat pengguna dan meminimalkan total pengetikan yang diperlukan. Pertimbangan tersebut diuraikan sebagai berikut:

- Perancang harus memutuskan mnemonik (nama perintah) apa yang akan digunakan untuk perintah yang berbeda. Perancang harus mencoba mengembangkan mnemonik yang bermakna namun tetap ringkas untuk meminimalkan jumlah pengetikan yang diperlukan. Misalnya, mnemonic terpendek harus diberikan ke perintah yang paling sering digunakan.
- Perancang harus memutuskan apakah pengguna akan diizinkan untuk mendefinisikan kembali nama perintah agar sesuai dengan preferensi mereka

sendiri. Membiarkan pengguna menentukan mnemoniknya sendiri untuk berbagai perintah adalah fitur yang berguna, tetapi ini meningkatkan kompleksitas pengembangan antarmuka pengguna.

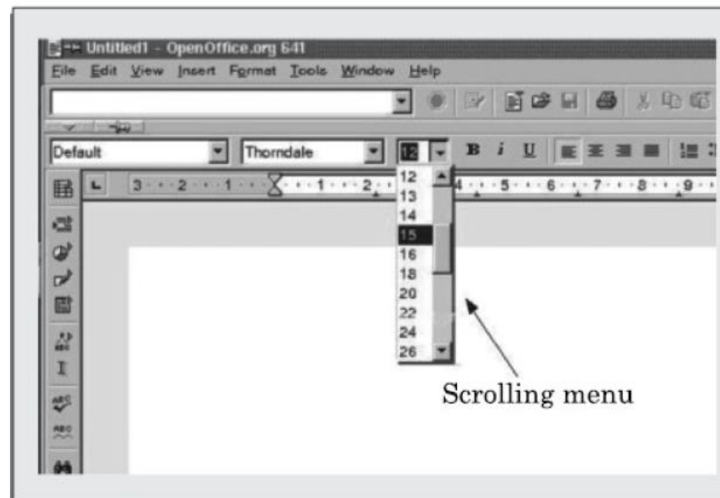
- Perancang harus memutuskan apakah mungkin untuk membuat perintah primitif untuk membentuk perintah yang lebih kompleks. Fasilitas komposisi perintah yang canggih akan membutuhkan sintaks dan semantik dari berbagai opsi komposisi perintah untuk ditentukan dengan jelas dan tidak ambigu. Kemampuan untuk menggabungkan perintah adalah fasilitas yang kuat di tangan pengguna yang berpengalaman, tetapi sangat tidak perlu bagi pengguna yang tidak berpengalaman.

Antarmuka berbasis menu

Keuntungan penting dari antarmuka berbasis menu di atas antarmuka berbasis bahasa perintah adalah bahwa antarmuka berbasis menu tidak mengharuskan pengguna untuk mengingat sintaks perintah yang tepat. Antarmuka berbasis menu didasarkan pada pengenalan nama perintah, bukan ingatan. Manusia jauh lebih baik dalam mengenali sesuatu daripada mengingatnya. Selanjutnya, dalam antarmuka berbasis menu, upaya pengetikan minimal karena sebagian besar interaksi dilakukan melalui pemilihan menu menggunakan perangkat penunjuk. Faktor ini merupakan pertimbangan penting bagi pengguna sesekali yang tidak dapat mengetik dengan cepat.

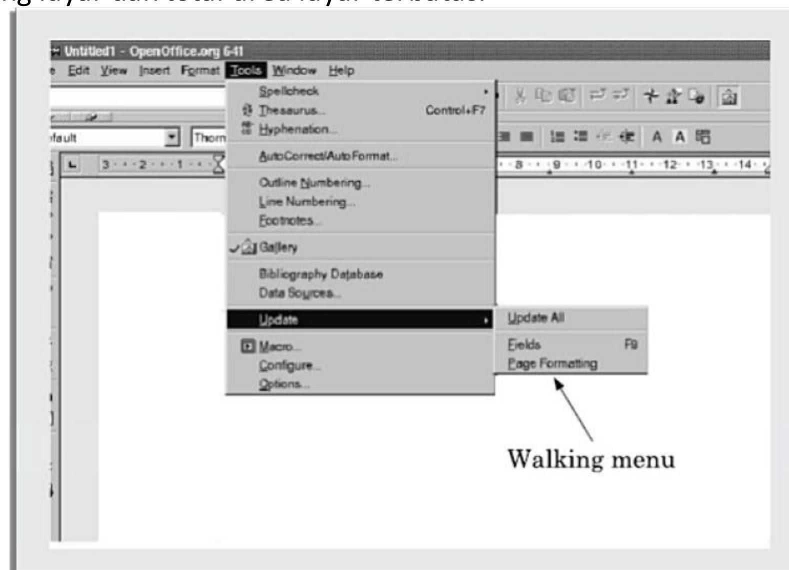
Namun, pengguna berpengalaman menemukan antarmuka pengguna berbasis menu lebih lambat daripada antarmuka berbasis bahasa perintah karena pengguna yang berpengalaman dapat mengetik dengan cepat dan bisa mendapatkan keuntungan kecepatan dengan menyusun perintah primitif yang berbeda untuk mengekspresikan perintah yang kompleks. Menulis perintah dalam antarmuka berbasis menu tidak dimungkinkan. Ini karena fakta bahwa tindakan yang melibatkan penghubung logis (dan, atau, dll.) canggung untuk ditentukan dalam sistem berbasis menu. Juga, jika jumlah pilihannya banyak, sulit untuk merancang antarmuka berbasis menu. Perangkat lunak berukuran sedang mungkin memerlukan ratusan atau ribuan pilihan menu yang berbeda. Faktanya, tantangan utama dalam desain antarmuka berbasis menu adalah menyusun sejumlah besar pilihan menu ke dalam bentuk yang dapat dikelola. Berikut ini adalah beberapa teknik yang tersedia untuk menyusun sejumlah besar item menu:

Menu bergulir: Terkadang daftar pilihan lengkap berukuran besar dan tidak dapat ditampilkan di dalam area menu, diperlukan pengguliran item menu. Ini akan memungkinkan pengguna untuk melihat dan memilih item menu yang tidak dapat diakomodasi di layar. Namun, dalam menu bergulir semua perintah harus berkorelasi tinggi, sehingga pengguna dapat dengan mudah menemukan perintah yang dia butuhkan. Ini penting karena pengguna tidak dapat melihat semua perintah pada satu waktu. Contoh situasi di mana menu gulir sering digunakan adalah pemilihan ukuran font dalam prosesor dokumen (lihat Gambar 9.1). Di sini, pengguna mengetahui bahwa daftar perintah hanya berisi ukuran font yang diatur dalam beberapa urutan dan dia dapat menggulir ke atas atau ke bawah untuk menemukan ukuran yang dia cari. Namun, jika perintah tidak memiliki hubungan pemesanan yang pasti, maka pengguna harus dalam kasus terburuk, menggulir semua perintah untuk menemukan perintah yang tepat yang dia cari, membuat organisasi ini tidak efisien.



Gambar 9.1 Pemilihan ukuran font menggunakan menu scrolling.

Menu berjalan: Menu berjalan sangat umum digunakan untuk menyusun banyak koleksi item menu. Dalam teknik ini, ketika item menu dipilih, item menu selanjutnya akan ditampilkan berdekatan dengannya dalam sub-menu. Contoh menu berjalan ditunjukkan pada Gambar 9.2. Menu berjalan dapat berhasil digunakan untuk menyusun perintah hanya jika ada puluhan daripada ratusan pilihan karena setiap menu yang ditampilkan di sebelahnya mengambil ruang layar dan total area layar terbatas.



Gambar 9.2 Contoh menu jalan kaki.

Menu hierarkis: Jenis menu ini cocok untuk layar kecil dengan area tampilan terbatas seperti di ponsel. Dalam menu hierarkis, item menu diatur dalam hierarki atau struktur pohon. Memilih item menu menyebabkan tampilan menu saat ini digantikan oleh submenu yang sesuai. Jadi dalam hal ini, seseorang dapat mempertimbangkan menu dan berbagai submenunya untuk membentuk struktur seperti pohon hierarkis. Menu berjalan dapat dianggap sebagai bentuk menu hierarkis yang dapat dipraktikkan ketika pohonnya dangkal. Menu hierarkis dapat digunakan untuk mengelola sejumlah besar pilihan, tetapi pengguna cenderung menghadapi masalah navigasi karena mereka mungkin kehilangan jejak di mana mereka berada di pohon menu. Ini mungkin adalah alasan utama mengapa jenis antarmuka ini sangat jarang digunakan.

Antarmuka Manipulasi Langsung

Antarmuka manipulasi langsung menyajikan antarmuka kepada pengguna dalam bentuk model visual (yaitu, ikon atau objek). Untuk alasan ini, antarmuka manipulasi langsung kadang-kadang disebut sebagai antarmuka ikonik. Dalam jenis antarmuka ini, pengguna mengeluarkan perintah dengan melakukan tindakan pada representasi visual objek, misalnya, menarik ikon yang mewakili file ke dalam ikon yang mewakili kotak sampah, untuk menghapus file.

Keuntungan penting dari antarmuka ikonik termasuk fakta bahwa ikon dapat dikenali oleh pengguna dengan sangat mudah, dan ikon tidak bergantung pada bahasa. Namun, pengguna berpengalaman juga menemukan antarmuka manipulasi langsung. Juga, sulit untuk memberikan perintah kompleks menggunakan antarmuka manipulasi langsung. Misalnya, jika seseorang harus menyeret ikon yang mewakili file ke ikon kotak sampah untuk menghapus file, maka untuk menghapus semua file dalam direktori, seseorang harus melakukan operasi ini satu per satu untuk semua file —yang bisa sangat mudah dilakukan dengan mengeluarkan perintah seperti delete *.*.

9.4 DASAR-DASAR PENGEMBANGAN GUI BERBASIS KOMPONEN

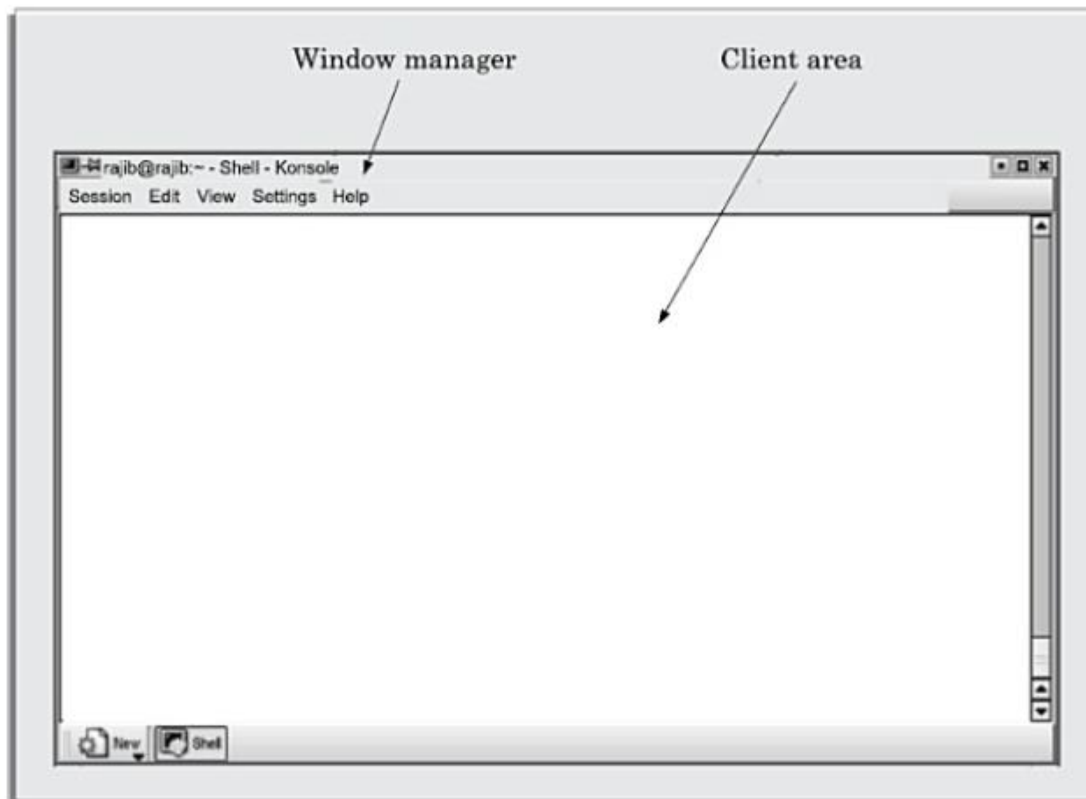
Antarmuka pengguna grafis menjadi populer pada 1980-an. Alasan utama mengapa hanya ada sedikit aplikasi berbasis GUI sebelum tahun delapan puluhan adalah karena terminal grafis terlalu mahal. Misalnya, harga terminal grafis pada masa itu jauh lebih mahal daripada harga komputer pribadi kelas atas saat ini. Juga, terminal grafis bertipe tabung penyimpanan dan tidak memiliki kemampuan raster.

Salah satu komputer pertama yang mendukung aplikasi berbasis GUI adalah komputer Apple Macintosh. Faktanya, popularitas komputer Apple Macintosh di awal tahun delapan puluhan secara langsung disebabkan oleh GUI-nya. Pada hari-hari awal desain GUI, programmer antarmuka pengguna biasanya memulai antarmukanya pengembangan dari awal. Dia akan mulai dari rutinitas tampilan piksel sederhana, menulis program untuk menggambar garis, lingkaran, teks, dll. Dia kemudian akan mengembangkan rutinitasnya sendiri untuk menampilkan item menu, membuat pilihan menu, dll. Gaya antarmuka pengguna saat ini telah mengalami banyak perubahan dibandingkan ke gaya awal. Gaya pengembangan antarmuka pengguna saat ini adalah berbasis komponen. Ia mengakui bahwa setiap antarmuka pengguna dapat dengan mudah dibangun dari beberapa komponen yang telah ditentukan sebelumnya seperti menu, kotak dialog, formulir, dll. Selain komponen standar, dan fasilitas untuk membuat antarmuka yang baik darinya, salah satu dukungan dasar yang tersedia untuk developer antarmuka pengguna adalah sistem jendela. Sistem jendela memungkinkan programmer aplikasi membuat dan memanipulasi jendela tanpa harus menulis fungsi dasar pembuatan jendela. Gambaran umum tentang sistem manajemen window, gaya pengembangan berbasis komponen, dan pemrograman visual adalah sebagai berikut:

Sistem Jendela

Sebagian besar antarmuka pengguna grafis modern dikembangkan menggunakan beberapa sistem jendela. Sistem jendela dapat menghasilkan tampilan melalui serangkaian jendela. Karena jendela adalah entitas dasar dalam antarmuka pengguna grafis seperti itu, pertama-tama kita perlu membahas apa sebenarnya jendela itu.

Window: Window adalah area persegi panjang di layar. Sebuah window (jendela) dapat dianggap sebagai layar virtual, dalam arti menyediakan antarmuka kepada pengguna untuk melakukan aktivitas independen, misalnya, satu window dapat digunakan untuk mengedit program dan lainnya untuk menggambar gambar, dll.



Gambar 9.3 Jendela dengan area klien dan pengguna ditandai.

Sebuah jendela dapat dibagi menjadi dua bagian — bagian klien, dan bagian non-klien. Area klien membentuk seluruh jendela, kecuali untuk batas dan bilah gulir. Area klien adalah area yang tersedia untuk aplikasi klien untuk ditampilkan. Bagian non-klien dari jendela menentukan tampilan dan nuansa jendela. Tampilan dan nuansa mendefinisikan perilaku dasar untuk semua jendela, seperti membuat, memindahkan, mengubah ukuran, membuat ikon jendela. Manajer jendela bertanggung jawab untuk mengelola dan memelihara area non-klien dari sebuah jendela. Sebuah jendela dasar dengan bagian-bagian yang berbeda ditunjukkan pada Gambar 9.3.

Window management system (WMS)

Antarmuka pengguna grafis biasanya terdiri dari sejumlah besar jendela. Oleh karena itu, diperlukan suatu cara yang sistematis untuk mengelola jendela-jendela tersebut. Sebagian besar lingkungan pengembangan antarmuka pengguna grafis melakukan ini melalui sistem manajemen jendela (WMS). Sistem manajemen jendela pada dasarnya adalah manajer sumber daya. Itu melacak sumber daya area layar dan mengalokasikannya ke berbagai jendela yang berusaha menggunakan layar. Dari perspektif yang lebih luas, WMS dapat dianggap sebagai sistem manajemen antarmuka pengguna (UIMS) —yang tidak hanya melakukan manajemen sumber daya, tetapi juga menyediakan perilaku dasar ke windows dan menyediakan beberapa rutinitas utilitas untuk programmer aplikasi untuk pengembangan antarmuka pengguna. Sebuah WMS menyederhanakan tugas desainer GUI untuk sebagian besar dengan menyediakan perilaku dasar untuk berbagai jendela seperti memindahkan, mengubah ukuran, ikon, dll segera setelah mereka dibuat dan dengan menyediakan rutinitas dasar untuk memanipulasi jendela dari program aplikasi seperti membuat, menghancurkan, mengubah atribut yang berbeda dari jendela, dan menggambar teks, garis, dll.

Sebuah WMS terdiri dari dua bagian (lihat Gambar 9.4):

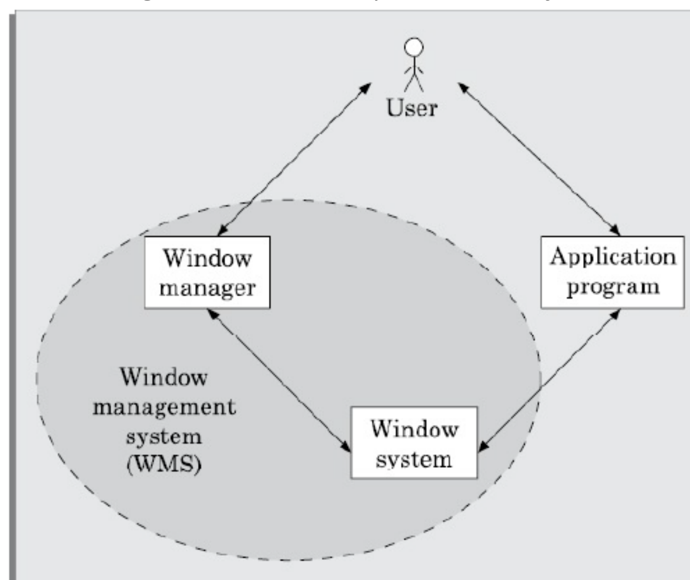
- pengelola jendela, dan

- sistem jendela.

Manajer jendela dan sistem jendela: Manajer jendela dibangun di atas sistem jendela dalam artian memanfaatkan berbagai layanan yang disediakan oleh sistem jendela. Manajer jendela dan bukan sistem jendela menentukan bagaimana tampilan dan perilaku jendela. Bahkan, beberapa jenis window manager dapat dikembangkan berdasarkan sistem window yang sama. Manajer jendela dapat dianggap sebagai jenis klien khusus yang menggunakan layanan (panggilan fungsi) yang didukung oleh sistem jendela. Programmer aplikasi juga dapat secara langsung memanggil layanan sistem jendela untuk mengembangkan antarmuka pengguna. Hubungan antara window manager, window system, dan program aplikasi ditunjukkan pada Gambar 9.4. Gambar ini menunjukkan bahwa pengguna akhir dapat berinteraksi dengan aplikasi itu sendiri atau dengan pengelola jendela (mengubah ukuran, memindahkan, dll.) dan aplikasi dan pengelola jendela memanggil layanan pengelola jendela. Manajer jendela adalah komponen WMS yang dengannya pengguna akhir berinteraksi untuk melakukan berbagai operasi terkait jendela seperti pemosisian ulang jendela, pengubahan ukuran jendela, ikonifikasi, dll.

Biasanya rumit untuk mengembangkan antarmuka pengguna menggunakan serangkaian besar rutinitas yang disediakan oleh sistem jendela dasar. Oleh karena itu, sebagian besar sistem pengembangan antarmuka pengguna biasanya menyediakan abstraksi tingkat tinggi yang disebut widget untuk pengembangan antarmuka pengguna. Widget adalah bentuk pendek dari objek jendela. Kita tahu bahwa suatu objek pada dasarnya adalah kumpulan data terkait dengan beberapa operasi yang ditentukan pada data ini yang tersedia secara eksternal untuk beroperasi pada data ini. Data objek jendela adalah atribut geometris (seperti ukuran, lokasi, dll.) dan atribut lain seperti latar belakang dan warna latar depan, dll. Operasi yang didefinisikan pada data ini meliputi, mengubah ukuran, memindahkan, menggambar, dll.

Widget adalah komponen antarmuka pengguna standar. Antarmuka pengguna biasanya dibuat dengan mengintegrasikan beberapa widget. Beberapa jenis widget penting yang biasanya disediakan dengan sistem *developer user interface*.



Gambar 9.4 Sistem manajemen jendela.

Pengembangan berbasis komponen

Gaya pengembangan berdasarkan widget disebut gaya pengembangan GUI berbasis komponen (atau berbasis widget). Ada beberapa keuntungan penting menggunakan gaya

desain berbasis widget. Salah satu alasan terpenting untuk menggunakan widget sebagai blok penyusun adalah karena widget membantu pengguna mempelajari antarmuka dengan cepat. Dalam gaya pengembangan ini, antarmuka pengguna untuk aplikasi yang berbeda dibangun dari komponen dasar yang sama. Oleh karena itu, pengguna dapat memperluas pengetahuannya tentang perilaku komponen standar dari satu aplikasi ke aplikasi lainnya. Selain itu, gaya pengembangan antarmuka pengguna berbasis komponen mengurangi pekerjaan programmer aplikasi secara signifikan karena ia lebih merupakan integrator komponen antarmuka pengguna daripada programmer dalam pengertian tradisional.

Pemrograman visual

Pemrograman visual adalah gaya drag and drop dari pengembangan program. Dalam gaya pengembangan antarmuka pengguna ini, sejumlah objek visual (ikon) yang mewakili komponen GUI disediakan oleh lingkungan pemrograman. Programmer aplikasi dapat dengan mudah mengembangkan antarmuka pengguna dengan menyeret jenis komponen yang diperlukan (misalnya, menu, formulir, dll.) dari ikon yang ditampilkan dan menempatkannya di mana pun diperlukan. Dengan demikian, pemrograman visual dapat dianggap sebagai pengembangan program melalui manipulasi beberapa objek visual. Penggunaan kembali komponen program dalam bentuk objek visual merupakan aspek penting dari gaya pemrograman ini. Meskipun populer untuk pengembangan antarmuka pengguna, gaya pemrograman ini dapat digunakan untuk aplikasi lain seperti aplikasi Computer-Aided Design (misalnya, desain pabrik), simulasi, dll. Pengembangan antarmuka pengguna menggunakan bahasa pemrograman visual sangat mengurangi upaya yang diperlukan untuk mengembangkan antarmuka.

Contoh bahasa pemrograman visual yang populer adalah Visual Basic, Visual C++, dll. Visual C++ menyediakan alat untuk membangun program dengan antarmuka pengguna berbasis jendela untuk lingkungan Microsoft Windows. Dalam visual C++ Anda biasanya mendesain bilah menu, ikon, dan kotak dialog, dll. sebelum menambahkannya ke program Anda. Objek-objek ini disebut sebagai sumber daya. Anda dapat mendesain bentuk, lokasi, jenis, dan ukuran kotak dialog sebelum menulis kode C++ untuk aplikasi.

Jenis Widget

Paket pemrograman antarmuka yang berbeda mendukung set widget yang berbeda.

Namun, jumlah yang mengejutkan dari mereka berisi jenis widget yang serupa, sehingga orang dapat memikirkan kumpulan widget umum yang berlaku untuk sebagian besar antarmuka. Widget berikut dipilih sebagai perwakilan dari kelas generik ini.

Widget label: Ini mungkin salah satu widget paling sederhana. Widget label tidak melakukan apa pun kecuali menampilkan label, yaitu, tidak memiliki kemampuan interaksi lain dan tidak sensitif terhadap klik mouse. Widget label sering digunakan sebagai bagian dari widget lainnya.

Widget kontainer: Widget ini tidak berdiri sendiri, tetapi ada hanya untuk memuat widget lain. Widget lain dibuat sebagai anak dari widget container. Saat widget container dipindahkan atau diubah ukurannya, widget turunannya juga dipindahkan atau diubah ukurannya. Widget penampung tidak memiliki rutinitas panggilan balik yang terkait dengannya.

Menu pop-up: Ini bersifat sementara dan khusus untuk tugas. Menu pop-up muncul saat menekan tombol mouse, terlepas dari posisi mouse.

Menu pull-down : Ini lebih permanen dan umum. Anda harus memindahkan kursor ke lokasi tertentu dan menarik menu jenis ini ke bawah.

Kotak dialog: Kita sering perlu memilih beberapa elemen dari daftar pilihan. Kotak dialog tetap terlihat hingga pengguna secara eksplisit diberhentikan. Kotak dialog dapat

menyertakan area untuk memasukkan teks serta nilai. Jika perintah terapkan didukung dalam kotak dialog, nilai yang baru dimasukkan dapat dicoba tanpa mengabaikan kotak. Meskipun sebagian besar kotak dialog meminta Anda untuk memasukkan beberapa informasi, ada beberapa kotak dialog yang hanya informatif, memperingatkan Anda tentang masalah dengan sistem Anda atau kesalahan yang Anda buat. Umumnya, kotak-kotak ini meminta Anda untuk membaca informasi yang disajikan dan kemudian klik OK untuk mengabaikan kotak tersebut.

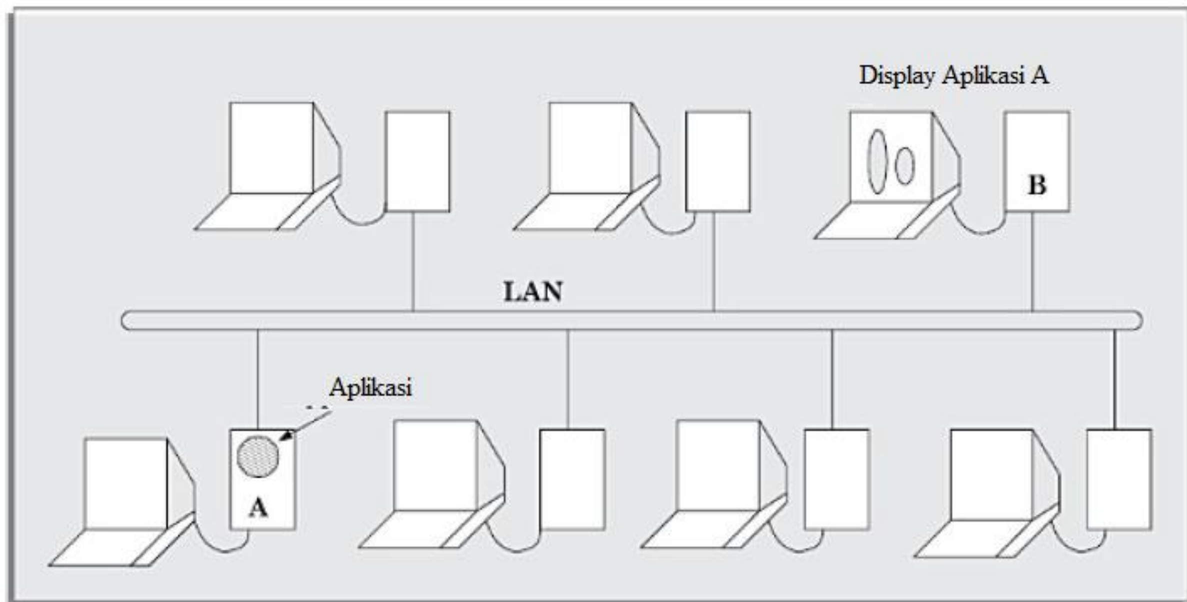
Tombol tekan: Tombol tekan berisi kata kunci atau gambar yang menjelaskan tindakan yang dipicu saat Anda mengaktifkan tombol. Biasanya, tindakan yang terkait dengan tombol tekan terjadi segera saat Anda mengklik tombol tekan kecuali jika berisi elipsis (. . .). Tombol tekan dengan elipsis umumnya menunjukkan bahwa kotak dialog lain akan muncul.

Tombol radio: Satu set tombol radio digunakan ketika hanya satu opsi yang harus dipilih dari banyak opsi. Tombol radio adalah lingkaran berongga yang diikuti dengan teks yang menjelaskan opsi yang dimaksud. Ketika tombol radio dipilih, tombol itu tampak terisi dan tombol radio yang dipilih sebelumnya dari grup tidak dipilih. Hanya satu tombol radio dari grup yang dapat dipilih setiap saat. Operasi ini mirip dengan tombol pemilihan pita yang tersedia di radio lama.

Kotak kombo: Kotak kombo terlihat seperti tombol hingga pengguna berinteraksi dengannya. Saat pengguna menekan atau mengkliknya, kotak kombo menampilkan menu item untuk dipilih. Biasanya kotak kombo digunakan untuk menampilkan salah satu dari banyak pilihan ketika ruang terbatas, jumlah pilihan besar, atau ketika item menu dihitung saat run-time.

Sekilas tentang X-Window/MOTIF

Salah satu alasan penting di balik popularitas ekstrim sistem X-window mungkin karena fakta bahwa ia memungkinkan pengembangan GUI portabel. Aplikasi yang dikembangkan menggunakan sistem X-window tidak bergantung pada perangkat. Selain itu, aplikasi yang dikembangkan menggunakan sistem X-window menjadi jaringan independen dalam arti bahwa antarmuka akan bekerja dengan baik pada terminal yang terhubung di mana saja di jaringan yang sama dengan komputer yang menjalankan aplikasi tersebut. Operasi GUI jaringan-independen telah secara skematis diwakili pada Gambar 9.5. Di sini, A adalah aplikasi komputer tempat aplikasi tersebut berjalan. B dapat berupa komputer mana pun di jaringan tempat Anda dapat berinteraksi dengan aplikasi. GUI independen jaringan dipelopori oleh sistem X-window pada pertengahan tahun delapan puluhan di MIT (Massachusetts Institute of Technology) dengan dukungan dari DEC (Digital Equipment Corporation). Saat ini banyak sistem pengembangan antarmuka pengguna mendukung pengembangan GUI yang tidak bergantung pada jaringan, misalnya, komponen AWT dan Swing dari Java.

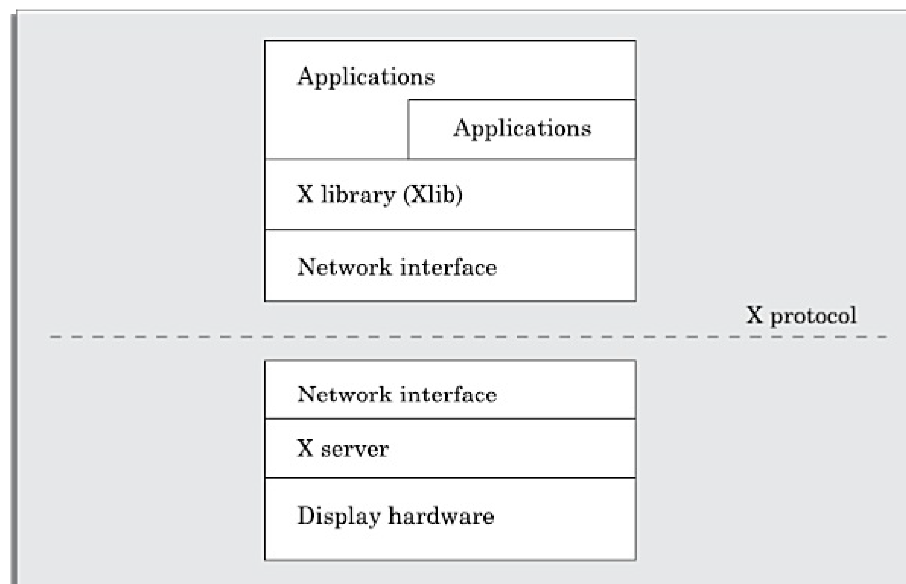


Gambar 9.5 GUI independen jaringan.

Fungsi X-window adalah fungsi tingkat rendah yang ditulis dalam bahasa C yang dapat dipanggil dari program aplikasi. Tetapi hanya perancang aplikasi yang sangat serius yang akan memprogram secara langsung menggunakan rutinitas perpustakaan X-windows. Dibangun di atas X-windows adalah fungsi tingkat yang lebih tinggi secara kolektif disebut Xtoolkit, yang terdiri dari satu set widget dasar dan satu set rutinitas untuk memanipulasi widget ini. Salah satu set widget yang paling banyak digunakan adalah X/Motif. Digital Equipment Corporation (DEC) menggunakan fungsi X-window dasar untuk mengembangkan tampilan dan nuansanya sendiri untuk desain antarmuka yang disebut DECWindows. Bagian berikut ini, akan memberikan gambaran yang sangat singkat tentang sistem X-window dan arsitekturnya dan pembaca yang tertarik dirujuk ke Scheifler et al. [1988] untuk studi lebih lanjut tentang pengembangan antarmuka pengguna grafis menggunakan X-windows dan Motif.

Arsitektur X

Arsitektur X digambarkan dalam Gambar 9.6. Berbagai istilah yang digunakan dalam diagram ini dijelaskan sebagai berikut:



Gambar 9.6 Arsitektur Sistem X.

Xserver: Server X berjalan pada perangkat keras yang disambungkan dengan layar dan papan kunci. Server X melakukan grafik tingkat rendah, mengelola jendela, dan fungsi input pengguna. Server X mengontrol akses ke sumber daya tampilan grafis yang dipetakan sedikit dan mengelolanya.

Protokol X. Protokol X mendefinisikan format permintaan antara aplikasi klien dan server tampilan melalui jaringan. Protokol X dirancang untuk tidak bergantung pada perangkat keras, sistem operasi, protokol jaringan yang mendasarinya, dan bahasa pemrograman yang digunakan.

Pustaka X (Xlib). Xlib menyediakan satu set sekitar 300 rutinitas utilitas untuk panggilan aplikasi. Rutinitas ini mengubah panggilan prosedur menjadi permintaan yang dikirimkan ke server. Xlib menyediakan primitif tingkat rendah untuk mengembangkan antarmuka pengguna, seperti menampilkan jendela, menggambar karakter dan grafik di jendela, menunggu acara tertentu, dll.

Xtoolkit (Xt). Xtoolkit terdiri dari dua bagian: intrinsik dan widget. Kita telah melihat bahwa widget adalah komponen antarmuka pengguna yang telah ditentukan sebelumnya seperti bilah gulir, bilah menu, tombol tekan, dll. untuk mendesain GUI. Intrinsik adalah satu set sekitar selusin rutinitas perpustakaan yang memungkinkan seorang programmer untuk menggabungkan satu set widget ke dalam antarmuka pengguna. Untuk mengembangkan antarmuka pengguna, perancang harus mengumpulkan kumpulan komponen (widget) yang dia butuhkan, dan kemudian dia perlu mendefinisikan karakteristik (disebut sumber daya) dan perilaku widget ini dengan menggunakan rutinitas intrinsik untuk menyelesaikan pengembangan. dari antarmuka. Oleh karena itu, mengembangkan antarmuka menggunakan Xtoolkit jauh lebih mudah daripada mengembangkan antarmuka yang sama hanya menggunakan pustaka X.

Pengukuran Ukuran GUI berbasis Komponen

Baris kode (LOC) bukanlah metrik yang tepat untuk memperkirakan dan mengukur ukuran GUI berbasis komponen. Hal ini dikarenakan, antarmuka dikembangkan dengan mengintegrasikan beberapa komponen yang telah dibangun sebelumnya. Komponen berbeda yang membentuk antarmuka mungkin ditulis menggunakan kode dengan ukuran yang sangat berbeda. Namun, sejauh upaya developer GUI yang mengembangkan antarmuka dengan mengintegrasikan komponen mungkin tidak terpengaruh oleh ukuran kode komponen yang dia integrasikan.

Cara untuk mengukur ukuran antarmuka pengguna modern adalah poin widget (wp). Ukuran antarmuka pengguna (dalam unit wp) hanyalah jumlah total widget yang digunakan dalam antarmuka. Ukuran antarmuka dalam unit wp adalah ukuran kerumitan antarmuka dan kurang lebih tidak tergantung pada lingkungan implementasi. Ukuran wp membuka peluang untuk kontrak pada jumlah fungsionalitas antarmuka pengguna yang terukur, alih-alih definisi yang tidak jelas dari sistem yang lengkap. Namun, hingga saat ini belum ada hasil yang dilaporkan untuk memperkirakan upaya pengembangan dalam hal metrik wp. Cara alternatif untuk menghitung ukuran GUI adalah dengan menghitung jumlah layar. Namun, ini akan menjadi tidak akurat karena kompleksitas layar dapat berkisar dari yang sangat sederhana hingga yang sangat kompleks.

9.5 METODOLOGI DESAIN ANTARMUKA PENGGUNA

Saat ini, tidak ada metodologi langkah-demi-langkah yang tersedia yang dapat diikuti dengan hafalan untuk menghasilkan antarmuka pengguna yang baik. Apa yang disajikan di bagian ini adalah serangkaian rekomendasi yang dapat Anda gunakan untuk melengkapi kecerdikan Anda. Meskipun hampir semua metodologi desain GUI yang populer berpusat

pada pengguna, konsep ini harus dibedakan dengan jelas dari desain antarmuka pengguna oleh pengguna. Sebelum kita mulai membahas tentang metodologi desain antarmuka pengguna, mari kita bedakan antara desain yang berpusat pada pengguna dan desain oleh pengguna.

- Desain yang berpusat pada pengguna adalah tema dari hampir semua teknik desain antarmuka pengguna modern. Namun, desain yang berpusat pada pengguna tidak berarti desain oleh pengguna. Seseorang tidak boleh membuat pengguna mendesain antarmuka, juga tidak boleh berasumsi bahwa pendapat pengguna tentang alternatif desain mana yang lebih unggul selalu benar. Meskipun pengguna mungkin memiliki pengetahuan yang baik tentang tugas yang harus mereka lakukan menggunakan GUI, tetapi mereka mungkin tidak mengetahui masalah desain GUI.
- Pengguna memiliki pengetahuan yang baik tentang tugas yang harus mereka lakukan, mereka juga tahu apakah mereka menemukan antarmuka yang mudah dipelajari dan digunakan tetapi mereka memiliki pemahaman dan pengalaman yang kurang dalam desain GUI daripada developer GUI.

Implikasi Kemampuan Kognisi Manusia pada Desain Antarmuka Pengguna

Area interaksi manusia-komputer di mana penelitian ekstensif telah dilakukan adalah bagaimana kemampuan dan keterbatasan kognitif manusia memengaruhi cara antarmuka harus dirancang. Beberapa masalah utama yang paling sering dilaporkan dalam literatur.

Memori terbatas: Manusia dapat mengingat paling banyak tujuh item informasi yang tidak terkait untuk waktu yang singkat. Oleh karena itu, perancang GUI seharusnya tidak mengharuskan pengguna untuk mengingat terlalu banyak item informasi dalam satu waktu. Merupakan tanggung jawab perancang GUI untuk mengantisipasi informasi apa yang akan dibutuhkan pengguna pada titik mana dari setiap tugas dan untuk memastikan bahwa informasi yang relevan ditampilkan untuk dilihat pengguna. Menampilkan beberapa informasi kepada pengguna di beberapa titik, dan kemudian memintanya untuk mengingat kembali informasi tersebut di layar yang berbeda di mana mereka tidak lagi melihat informasi tersebut, menempatkan beban memori pada pengguna dan harus dihindari sedapat mungkin.

Penutupan tugas yang sering: Melakukan tugas (kecuali untuk tugas yang sangat sepele) memerlukan beberapa subtugas. Ketika sistem memberikan umpan balik yang jelas kepada pengguna bahwa tugas telah berhasil diselesaikan, pengguna mendapatkan rasa pencapaian dan kelegaan. Pengguna dapat menghapus informasi mengenai tugas yang diselesaikan dari memori. Ini dikenal sebagai t penutupan sk. Ketika tugas keseluruhan cukup besar dan kompleks, itu harus dibagi menjadi subtugas, yang masing-masing memiliki subtujuan yang jelas yang dapat menjadi titik penutupan.

Mengenali daripada mengingat. Penarikan informasi menimbulkan beban memori yang lebih besar pada pengguna dan harus dihindari sejauh mungkin. Di sisi lain, pengakuan informasi dari alternatif yang ditunjukkan kepadanya lebih dapat diterima.

Prosedural versus berorientasi objek: Desain prosedural fokus pada tugas, mendorong pengguna di setiap langkah tugas, memberi mereka sedikit pilihan untuk hal lain. Pendekatan ini paling baik diterapkan dalam situasi di mana tugas-tugasnya sempit dan terdefinisi dengan baik atau di mana pengguna tidak berpengalaman, seperti ATM bank. Antarmuka berorientasi objek di sisi lain berfokus pada objek. Hal ini memungkinkan pengguna berbagai pilihan.

Metodologi Desain GUI

Metodologi desain GUI yang disajikan di sini didasarkan pada karya seminal Frank Ludolph [Frank1998]. Metodologi desain antarmuka pengguna terdiri dari langkah-langkah penting berikut:

- Periksa model kasus penggunaan perangkat lunak. Wawancara, diskusikan, dan tinjau masalah GUI dengan pengguna akhir.
- Pemodelan tugas dan objek.
- Seleksi metafora.
- Desain interaksi dan tata letak kasar.
- Presentasi rinci dan desain grafis.
- konstruksi GUI.
- Evaluasi kegunaan.

Memeriksa model kasus penggunaan

Titik awal untuk desain GUI adalah model use case. Ini menangkap tugas-tugas penting yang perlu dilakukan pengguna menggunakan perangkat lunak. Sejauh mungkin, antarmuka pengguna harus dikembangkan menggunakan satu atau lebih metafora. Metafora membantu dalam pengembangan antarmuka dengan upaya yang lebih rendah dan mengurangi biaya untuk melatih pengguna. Seiring waktu, orang telah mengembangkan metode yang efisien untuk menangani beberapa situasi yang umum terjadi. Solusi ini adalah tema dari metafora. Metafora juga dapat didasarkan pada objek fisik seperti buku pengunjung, katalog, pena, kuas, gunting, dll. Solusi berdasarkan metafora mudah dipahami oleh pengguna, mengurangi waktu belajar dan biaya pelatihan. Beberapa metafora yang umum digunakan adalah sebagai berikut:

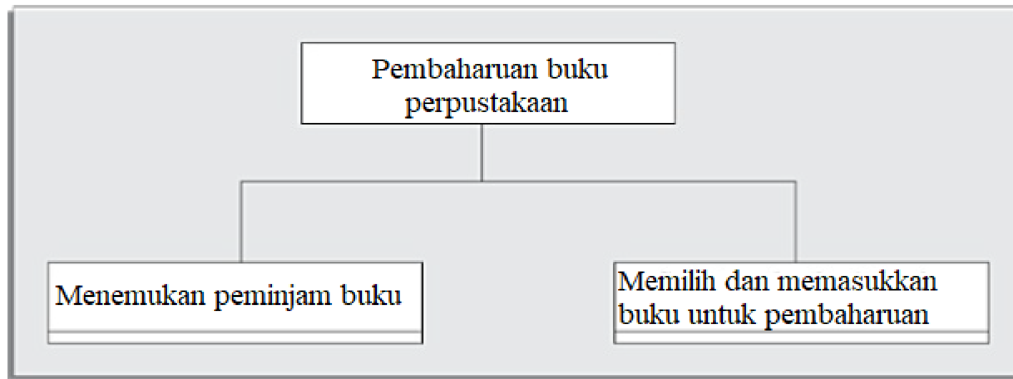
- Papan putih
- Kereta Belanja
- Desktop
- Meja kerja editor
- halaman putih
- Halaman Kuning
- lemari kantor
- Kotak pos
- papan pengumuman
- Buku Pengunjung

Pemodelan tugas dan objek

Tugas adalah kegiatan manusia yang dimaksudkan untuk mencapai beberapa tujuan. Contoh tujuan tugas dapat sebagai berikut:

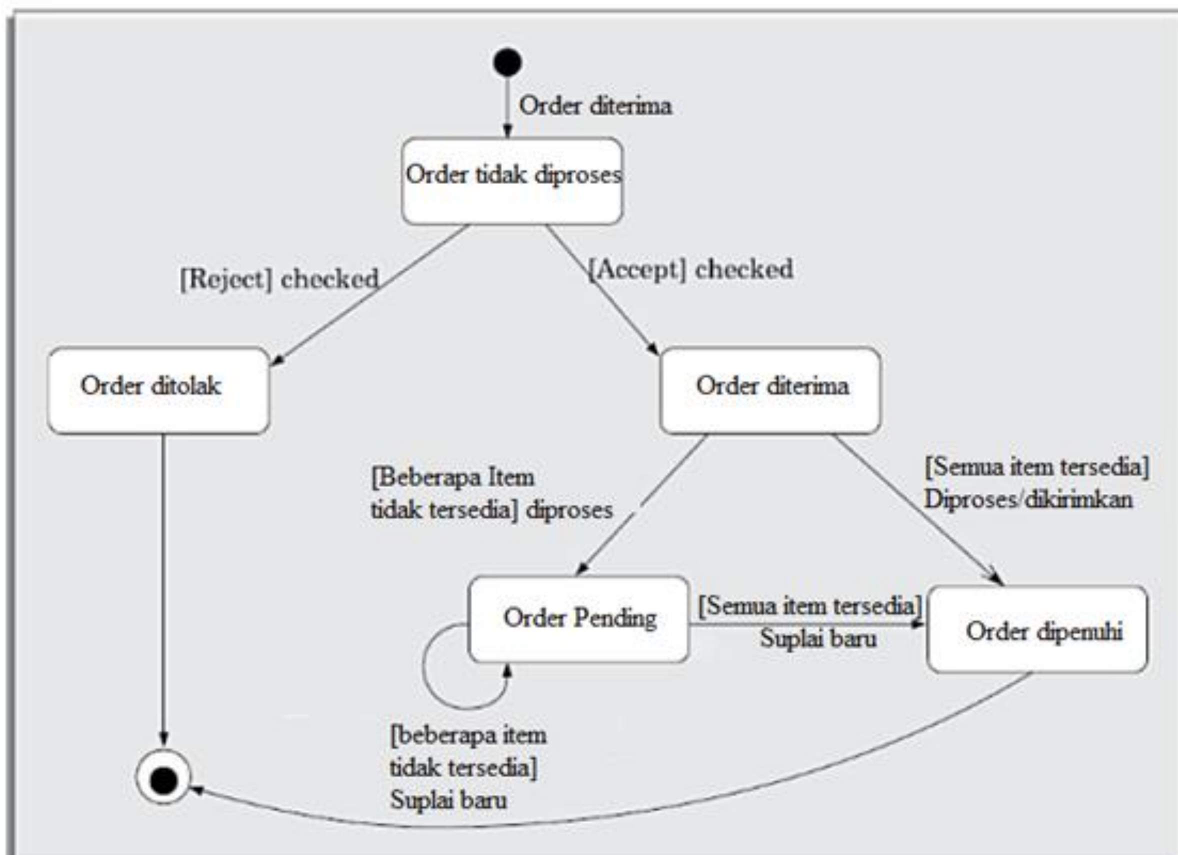
- Pesan kursi maskapai
- Beli barang
- Transfer uang dari satu rekening ke rekening lainnya
- Pesan kargo untuk dikirim ke alamat

Model tugas adalah model abstrak dari struktur tugas. Model tugas harus menunjukkan struktur subtugas yang perlu dilakukan pengguna untuk mencapai tujuan tugas secara keseluruhan. Setiap tugas dapat dimodelkan sebagai hierarki subtugas. Model tugas dapat digambar menggunakan notasi grafis yang mirip dengan model jaringan aktivitas yang kita bahas di Bab 3. Tugas dapat digambar sebagai kotak dengan garis yang menunjukkan bagaimana tugas dipecah menjadi subtugas. Kotak tugas yang digarisbawahi berarti bahwa tidak diperlukan dekomposisi tugas lebih lanjut. Contoh dekomposisi tugas menjadi subtugas ditunjukkan pada Gambar 9.7.



Gambar 9.7 Dekomposisi tugas menjadi subtugas.

Identifikasi objek pengguna membentuk dasar dari desain berbasis objek. Model objek pengguna adalah model objek bisnis yang diyakini oleh pengguna akhir bahwa mereka berinteraksi. Objek dalam perangkat lunak perpustakaan dapat berupa buku, jurnal, anggota, dll. Objek dalam perangkat lunak otomasi supermarket dapat berupa item, tagihan, indentasi, daftar belanja, dll. Diagram keadaan untuk suatu objek dapat digambar menggunakan notasi yang mirip dengan yang digunakan oleh UML. Diagram keadaan model objek dapat digunakan untuk menentukan item menu mana yang harus diredupkan dalam keadaan. Contoh diagram diagram status untuk objek pesanan ditunjukkan pada Gambar 9.8.



Gambar 9.8 Diagram diagram keadaan untuk objek pesanan.

Pilihan metafora

Tempat pertama yang harus dicari ketika mencoba mengidentifikasi kandidat metafora adalah himpunan paralel dengan objek, tugas, dan terminologi dari kasus penggunaan. Jika tidak ada metafora yang jelas dapat ditemukan, maka perancang dapat

kembali pada metafora dunia fisik benda-benda konkret. Kesesuaian setiap kandidat metafora harus diuji dengan menyatakan kembali objek dan tugas dari model antarmuka pengguna dalam hal metafora. Kriteria lain yang dapat digunakan untuk menilai metafora adalah bahwa metafora harus sesederhana mungkin, operasi yang menggunakan metafora harus jelas dan koheren dan harus sesuai dengan pengetahuan 'akal sehat' pengguna. Misalnya, akan sangat canggung dan merepotkan pengguna jika metafora gunting digunakan untuk merekatkan barang yang berbeda.

Contoh 9.1 Kita perlu mengembangkan antarmuka untuk toko pesanan berbayar berbasis web, di mana pengguna dapat memeriksa isi toko melalui browser web dan dapat memesannya. Beberapa metafora yang mungkin untuk bagian yang berbeda dari masalah ini sebagai berikut:

- Item yang berbeda dapat diambil dari rak dan diperiksa. Pengguna dapat meminta katalog yang terkait dengan item dengan mengklik item tersebut.
- Item terkait dapat diambil dari laci lemari item.
- Item dapat diatur dalam bentuk buku, mirip dengan cara informasi tentang komponen elektronik diatur dalam buku pegangan semikonduktor.

Setelah pengguna memutuskan tentang item yang ingin mereka beli, mereka dapat memasukkannya ke dalam keranjang belanja.

Desain interaksi dan tata letak kasar

Desain interaksi melibatkan pemetaan subtugas ke dalam kontrol yang sesuai, dan widget lain seperti formulir, kotak teks, dll. Ini melibatkan pembuatan pilihan dari sekumpulan komponen yang tersedia yang paling sesuai dengan subtugas. Tata letak kasar menyangkut bagaimana kontrol, widget lain untuk diatur di windows.

Presentasi terperinci dan desain grafis

Setiap jendela harus mewakili satu objek atau banyak objek yang memiliki hubungan yang jelas satu sama lain. Pada satu ekstrem, setiap tampilan objek bisa berada di jendelanya sendiri. Namun, ini kemungkinan akan menyebabkan terlalu banyak pembukaan, penutupan, pemindahan, dan pengubahan ukuran jendela. Di sisi lain, semua pemandangan dapat ditempatkan dalam satu jendela secara berdampingan, menghasilkan jendela yang sangat besar. Ini akan memaksa pengguna untuk memindahkan kursor di sekitar jendela untuk mencari objek yang berbeda.

Konstruksi GUI

Beberapa jendela harus didefinisikan sebagai dialog modal. Saat jendela adalah dialog modal, tidak ada jendela lain dalam aplikasi yang dapat diakses hingga jendela saat ini ditutup. Ketika dialog modal ditutup, pengguna dikembalikan ke jendela dari mana dialog modal dipanggil. Dialog modal biasanya digunakan ketika konfirmasi eksplisit atau langkah otorisasi diperlukan untuk suatu tindakan (mis., Konfirmasi penghapusan). Meskipun penggunaan dialog modal sangat penting dalam beberapa situasi, penggunaan dialog modal yang berlebihan mengurangi fleksibilitas pengguna. Secara khusus, urutan dialog modal harus dihindari.

Inspeksi antarmuka pengguna

Nielson [Niel94] mempelajari masalah kegunaan umum dan membangun daftar periksa poin yang dapat dengan mudah diperiksa untuk sebuah antarmuka. Daftar periksa berikut didasarkan pada karya Nielson [Niel94]:

Visibilitas status sistem: Sistem harus sejauh mungkin memberi informasi kepada pengguna tentang status sistem dan apa yang sedang terjadi. Misalnya, pengguna tidak boleh memberikan perintah dan terus menunggu, bertanya-tanya apakah sistem mogok dan dia harus mem-boot ulang sistem atau bahwa hasilnya akan muncul setelah beberapa waktu.

Kesesuaian antara sistem dan dunia nyata: Sistem harus berbicara dalam bahasa pengguna dengan kata-kata, frasa, dan konsep yang akrab dengan yang digunakan oleh pengguna, daripada menggunakan istilah yang berorientasi pada sistem.

Membatalkan kesalahan: Pengguna harus merasa bahwa dia memegang kendali daripada merasa tidak berdaya atau berada di kendali sistem. Langkah penting menuju ini adalah bahwa pengguna harus dapat membatalkan dan mengulang operasi.

Konsistensi: Pengguna tidak perlu bertanya-tanya apakah kata, konsep, dan operasi yang berbeda memiliki arti yang sama dalam situasi yang berbeda.

Pengakuan daripada mengingat: Pengguna tidak harus mengingat informasi yang disajikan di layar lain. Semua data dan instruksi harus terlihat di layar untuk dipilih oleh pengguna.

Dukungan untuk berbagai tingkat keahlian: Penyediaan akselerator untuk pengguna berpengalaman memungkinkan mereka melakukan tindakan yang paling sering mereka perlukan secara efisien.

Desain estetis dan minimalis: Dialog dan layar tidak boleh berisi informasi yang tidak relevan dan jarang dibutuhkan. Setiap unit informasi tambahan dalam dialog atau layar bersaing dengan unit terkait dan mengurangi visibilitasnya.

Pesan bantuan dan kesalahan: Ini harus diungkapkan dalam bahasa sederhana (tanpa kode), menunjukkan masalah secara tepat, dan menyarankan solusi secara konstruktif.

Pencegahan kesalahan: Kemungkinan kesalahan harus diminimalkan. Prinsip utama dalam hal ini adalah untuk mencegah pengguna memasukkan nilai yang salah. Dalam situasi di mana pilihan harus dibuat dari antara kumpulan nilai yang berbeda, kontrol harus menyajikan hanya nilai yang valid menggunakan daftar drop-down, satu set tombol opsi atau kontrol multichoice serupa. Ketika format tertentu diperlukan untuk data atribut, data yang dimasukkan harus divalidasi saat pengguna mencoba mengirimkan data.

9.6 RINGKASAN

- Konsep penting yang terkait dengan desain antarmuka pengguna dan menyarankan beberapa properti yang diinginkan yang harus dimiliki antarmuka pengguna yang baik.
- Gaya pengembangan antarmuka pengguna saat ini berbasis komponen. Antarmuka pengguna berbasis komponen meningkatkan kecepatan belajar dan juga mengurangi upaya pengembangan antarmuka. Pengembangan antarmuka berbasis komponen membuat pengguna terbiasa dengan cara standar berinteraksi dengan antarmuka. Pengguna dapat dengan mudah memperluas pengetahuan mereka tentang berinteraksi dengan satu antarmuka ke antarmuka lain, sehingga mengurangi waktu belajar untuk sebagian besar.
- Konsep dasar yang terkait dengan sistem manajemen jendela dan memberikan gambaran singkat tentang pemrograman X-Window/Motif dan Visual.

9.7 LATIHAN

1. Pilih opsi yang benar:
    a. Manakah dari berikut ini yang dapat dianggap memberikan ukuran paling akurat dari ukuran antarmuka pengguna:
        i. LOC komponen GUI
        ii. Jumlah skenario
        iii. Jumlah jendela
        iv. Ukuran data input dan output

   b. Manakah dari jenis antarmuka berikut yang menurut pengguna pemula paling mudah digunakan?
      i. Antarmuka manipulasi langsung
      ii. Antarmuka berbasis menu
      iii. Antarmuka berbasis perintah
      iv. Kombinasi menu dan antarmuka perintah
   c. Widget dalam terminologi antarmuka pengguna berarti:
      i. Objek jendela
      ii. Jendela yatim piatu
      iii. Apa yang Anda lihat adalah apa yang Anda dapatkan
      iv. Cebol yang cerdik
2. Sebutkan lima karakteristik yang diinginkan yang harus dimiliki oleh antarmuka pengguna yang baik.
3. Bandingkan keuntungan relatif dari antarmuka pengguna teks dan grafis.
4. Apa perbedaan antara panduan pengguna dan sistem bantuan online dalam antarmuka pengguna sistem perangkat lunak? Diskusikan berbagai cara di mana bantuan online dapat diberikan kepada pengguna saat dia menjalankan perangkat lunak.
   a. Bandingkan keuntungan relatif dari bahasa perintah, berbasis menu, dan antarmuka manipulasi langsung.
   b. Misalkan Anda telah diminta untuk merancang antarmuka pengguna dari produk perangkat lunak yang besar. Apakah Anda akan memilih berbasis menu, manipulasi langsung, berbasis bahasa perintah, atau campuran dari semua jenis antarmuka ini untuk mengembangkan antarmuka untuk produk Anda? Membenarkan pilihan Anda.
5. Nyatakan BENAR atau SALAH dari berikut ini. Dukung jawaban Anda dengan alasan yang tepat:
   a. Gaya pemrograman visual dibatasi untuk pengembangan antarmuka pengguna saja.
   b. Antarmuka pengguna tanpa mode lebih disukai untuk produk perangkat lunak yang perlu mendukung sejumlah besar fungsi bagi pengguna.
   c. Pengguna pemula biasanya lebih menyukai antarmuka bahasa perintah daripada antarmuka berbasis menu dan ikonik.
   d. Untuk antarmuka pengguna modern, LOC adalah ukuran akurat dari ukuran antarmuka.
   e. Tampilan dan nuansa sistem jendela pada dasarnya ditentukan oleh pengelola jendela.
6. Diskusikan mengapa beberapa paket perangkat lunak populer mendukung bahasa perintah selain antarmuka pengguna berbasis menu dan ikonik.
7. Buat daftar keuntungan dan kerugian penting dari antarmuka bahasa perintah.
8. Buat daftar keuntungan dan kerugian penting dari antarmuka berbasis menu.
9. Saat mengembangkan antarmuka pengguna untuk produk perangkat lunak, bagaimana Anda dapat mengakomodasi pengguna dengan tingkat keterampilan yang berbeda.
10. Bandingkan keuntungan relatif dari menu bergulir, menu hierarkis, dan menu berjalan sebagai teknik untuk mengatur perintah pengguna.
11. Apa yang Anda pahami dengan antarmuka ikonik? Jelaskan bagaimana Anda dapat mengeluarkan perintah menggunakan antarmuka ikonik.

12. Buat daftar keuntungan dan kerugian penting dari antarmuka manipulasi langsung.
13. Buat daftar komponen GUI penting yang dapat digunakan untuk mengembangkan antarmuka pengguna grafis untuk aplikasi apa pun.
14. Apa perbedaan antara antarmuka pengguna berbasis mode dan tanpa mode? Apa keuntungan relatif mereka? Mana yang akan Anda gunakan untuk mengembangkan antarmuka produk perangkat lunak yang mendukung sejumlah besar perintah? Justifikasi jawaban Anda.
15. Apa itu pengelola jendela? Sebutkan setidaknya dua tanggung jawab penting seorang window manager. Sebutkan beberapa window manager yang biasa digunakan pada sistem Linux.
16. Apa itu sistem manajemen jendela (WMS)? Mewakili komponen utama WMS dalam diagram skematik dan menjelaskan secara singkat peran mereka.
17. Apa keuntungan menggunakan sistem manajemen jendela untuk desain GUI? Sebutkan beberapa sistem manajemen jendela yang tersedia secara komersial.
18. Diskusikan secara singkat arsitektur sistem X window. Apa keuntungan penting menggunakan sistem X window untuk mengembangkan antarmuka pengguna grafis?
19. Tulis lima kalimat untuk menyoroti fitur-fitur penting Visual C++?
20. Apa yang Anda pahami dengan bahasa visual? Bagaimana bahasa seperti Visual C++ dan Visual Basic memungkinkan Anda melakukan pengembangan antarmuka pengguna berbasis komponen?
21. Apa yang Anda pahami dengan pengembangan antarmuka pengguna berbasis komponen? Apa keuntungan dari pengembangan antarmuka pengguna berbasis komponen?
22. Mengapa hitungan layar GUI aplikasi yang berbeda mungkin bukan ukuran yang akurat dari ukuran antarmuka pengguna? Sarankan ukuran yang lebih akurat dari ukuran antarmuka pengguna aplikasi. Jelaskan bagaimana mengatasi kesulitan dengan jumlah layar ukuran.
23. antara "desain yang berpusat pada pengguna" dan "desain oleh pengguna." Periksa pro dan kontra dari dua pendekatan ini untuk desain antarmuka pengguna.
24. Misalkan umpan balik pelanggan untuk produk yang Anda kembangkan adalah "terlalu sulit untuk dipelajari". Jelaskan bagaimana kecepatan pembelajaran antarmuka pengguna produk dapat ditingkatkan.
25. Bedakan antara antarmuka pengguna prosedural dan berorientasi objek. Jenis antarmuka mana yang lebih bermanfaat? Justifikasi jawaban Anda.
26. Apa yang Anda pahami dengan "tampilan dan nuansa" dari sebuah jendela. Komponen sistem operasi mana yang menentukan tampilan dan nuansa jendela. Jelaskan jawaban Anda.
27. Rancang dan kembangkan antarmuka pengguna grafis untuk perangkat lunak editor grafis yang dijelaskan dalam Latihan 6.18.
28. Siapkan daftar periksa setidaknya lima item inspeksi untuk inspeksi yang efektif dari setiap antarmuka pengguna.
29. Pelajari antarmuka pengguna dari beberapa produk perangkat lunak populer seperti Word, Powerpoint, Excel, OpenOffice, Gimp, dll. dan identifikasi metafora yang digunakan. Juga, periksa bagaimana tugas dipecah menjadi subtugas dan penutupan dicapai.
30. Apa yang Anda pahami dengan metafora dalam konteks desain antarmuka pengguna? Mengapa menguntungkan untuk merancang antarmuka pengguna berdasarkan

metafora? Daftar beberapa metafora yang dapat digunakan untuk desain antarmuka pengguna.

31. Apa yang Anda pahami dengan dialog modal? Mengapa ini diperlukan? Mengapa penggunaan terlalu banyak dialog modal dalam desain antarmuka harus dihindari?
32. Bagaimana kemampuan dan keterbatasan kognisi manusia memengaruhi perancangan antarmuka pengguna komputer-manusia?
33. Bedakan antara desain antarmuka yang berpusat pada pengguna dan desain antarmuka oleh pengguna.
34. Apakah ada perbedaan antara merancang perangkat lunak dan membangun model berdasarkan persyaratan perangkat lunak?
35. Tentukan metrik yang dapat digunakan untuk mengukur ukuran bagian antarmuka pengguna dari perangkat lunak.

BAB 10
CODING DAN TESTING F

Dalam bab ini, kita akan membahas fase pengkodean dan pengujian dari siklus hidup perangkat lunak. Pengkodean dilakukan setelah fase desain selesai dan dokumen desain telah berhasil ditinjau. Pada tahap pengkodean, setiap modul yang ditentukan dalam dokumen desain dikodekan dan unit diuji. Selama pengujian unit, setiap modul diuji secara terpisah dari modul lain. Artinya, modul diuji secara independen saat dan ketika pengkodeannya selesai. Setelah semua modul sistem telah dikodekan dan unit diuji, tahap integrasi dan pengujian sistem dilakukan.

Integrasi dan pengujian modul dilakukan sesuai dengan rencana integrasi. Rencana integrasi, di mana modul yang berbeda diintegrasikan bersama, biasanya membayangkan integrasi modul melalui sejumlah langkah. Selama setiap langkah integrasi, sejumlah modul ditambahkan ke sistem yang terintegrasi sebagian dan sistem yang dihasilkan diuji. Produk lengkap terbentuk hanya setelah semua modul terintegrasi bersama. Pengujian sistem dilakukan pada produk lengkap. Selama pengujian sistem, produk diuji terhadap persyaratannya seperti yang tercatat dalam dokumen SRS.

Pengujian adalah fase penting dalam pengembangan perangkat lunak dan biasanya membutuhkan upaya maksimal di antara semua fase pengembangan. Biasanya, pengujian perangkat lunak profesional dilakukan dengan menggunakan sejumlah besar kasus uji. Biasanya banyak kasus uji yang berbeda dapat dieksekusi secara paralel oleh anggota tim yang berbeda. Oleh karena itu, untuk mengurangi waktu pengujian, selama fase pengujian tenaga kerja terbesar (dibandingkan dengan semua fase siklus hidup lainnya) dikerahkan. Dalam organisasi pengembangan yang khas, setiap saat, jumlah maksimum insinyur perangkat lunak dapat ditemukan untuk terlibat dalam kegiatan pengujian. Maka tidak mengherankan jika dalam industri perangkat lunak selalu ada permintaan besar untuk insinyur pengujian perangkat lunak. Namun, banyak insinyur pemula memiliki kesan yang salah bahwa pengujian adalah aktivitas sekunder dan secara intelektual tidak merangsang seperti aktivitas yang terkait dengan fase pengembangan lainnya.

Selama bertahun-tahun, persepsi umum tentang pengujian sebagai monyet yang mengetik data acak dan mencoba merusak sistem telah berubah. Sekarang penguji dipandang sebagai ahli konsep, teknik, dan alat khusus. Seperti yang akan segera kita sadari, menguji produk perangkat lunak sama menantangnya dengan aktivitas pengembangan awal seperti spesifikasi, desain, dan pengkodean. Selain itu, pengujian melibatkan banyak pemikiran kreatif. Dalam Bab ini, pertama-tama kita membahas beberapa masalah penting yang terkait dengan kegiatan yang dilakukan dalam fase pengkodean. Selanjutnya, fokus pada berbagai jenis teknik pengujian program untuk program prosedural dan berorientasi objek.

10.1 KODING

Masukan pada tahap coding adalah dokumen desain yang dihasilkan pada akhir tahap desain. Harap diingat bahwa dokumen desain tidak hanya berisi desain sistem tingkat tinggi dalam bentuk struktur modul (misalnya, bagan struktur), tetapi juga desain detail. Desain rinci biasanya didokumentasikan dalam bentuk spesifikasi modul di mana struktur data dan algoritma untuk setiap modul ditentukan. Selama fase pengkodean, modul berbeda yang diidentifikasi dalam dokumen desain dikodekan sesuai dengan spesifikasi modul masing-masing, yang harus dapat menggambarkan tujuan keseluruhan dari fase pengkodean sebagai

berikut. Tujuan dari fase pengkodean adalah untuk mengubah desain sistem menjadi kode dalam bahasa tingkat tinggi, dan kemudian menguji unit kode ini.

Biasanya, organisasi pengembangan perangkat lunak yang baik mengharuskan programmer mereka untuk mematuhi beberapa gaya pengkodean yang terdefinisi dengan baik dan standar yang disebut standar pengkodean mereka. Organisasi pengembangan perangkat lunak ini merumuskan standar pengkodean mereka sendiri yang paling sesuai untuk mereka, dan mengharuskan developer mereka untuk mengikuti standar secara ketat karena keuntungan bisnis signifikan yang ditawarkannya. Keuntungan utama mengikuti gaya pengkodean standar adalah sebagai berikut:

- Standar pengkodean memberikan tampilan yang seragam pada kode yang ditulis oleh insinyur yang berbeda.
- Ini memfasilitasi pemahaman kode dan penggunaan kembali kode.
- Ini mempromosikan praktik pemrograman yang baik.

Standar pengkodean mencantumkan beberapa aturan yang harus diikuti selama pengkodean, seperti cara variabel diberi nama, cara kode ditata, konvensi pengembalian kesalahan, dll. Selain standar pengkodean, beberapa pedoman pengkodean juga ditentukan oleh perusahaan perangkat lunak. Tapi, apa perbedaan antara pedoman pengkodean dan standar pengkodean? Adalah wajib bagi programmer untuk mengikuti standar pengkodean. Kepatuhan kode mereka terhadap standar pengkodean diverifikasi selama inspeksi kode. Setiap kode yang tidak sesuai dengan standar pengkodean ditolak selama peninjauan kode dan kode tersebut dikerjakan ulang oleh programmer terkait. Sebaliknya, pedoman pengkodean memberikan beberapa saran umum mengenai gaya pengkodean yang harus diikuti tetapi meninggalkan implementasi aktual dari pedoman ini pada kebijaksanaan masing-masing pengembang.

Setelah sebuah modul dikodekan, biasanya dilakukan review kode untuk memastikan bahwa standar pengkodean diikuti dan juga untuk mendeteksi kesalahan sebanyak mungkin sebelum pengujian. Penting untuk mendeteksi kesalahan sebanyak mungkin selama tinjauan kode, karena tinjauan adalah cara yang efisien untuk menghilangkan kesalahan dari kode dibandingkan dengan penghapusan cacat menggunakan pengujian. Pertama membahas beberapa standar dan pedoman pengkodean yang representatif. Selanjutnya membahas teknik review kode.

Standar dan Pedoman Pengkodean

Organisasi pengembangan perangkat lunak yang baik biasanya mengembangkan standar dan pedoman pengkodean mereka sendiri tergantung pada apa yang paling sesuai dengan organisasi mereka dan berdasarkan jenis perangkat lunak tertentu yang mereka kembangkan. Untuk memberikan gambaran tentang jenis standar pengkodean yang digunakan, kita hanya akan membuat daftar beberapa standar pengkodean umum dan pedoman yang umumnya diadopsi oleh banyak organisasi pengembangan perangkat lunak, daripada mencoba memberikan daftar lengkap.

Standar pengkodean representatif

Aturan untuk membatasi penggunaan global: Aturan ini mencantumkan tipe data apa yang dapat dideklarasikan secara global dan apa yang tidak, dengan maksud untuk membatasi data yang perlu didefinisikan dengan cakupan global. Header standar untuk modul yang berbeda: Header modul yang berbeda harus memiliki format dan informasi standar untuk kemudahan pemahaman dan pemeliharaan. Berikut ini adalah contoh format header yang digunakan di beberapa perusahaan:

- Nama modul.
- Tanggal pembuatan modul.

- Nama penulis.
- Sejarah modifikasi.
- Sinopsis modul. Ini adalah artikel kecil tentang apa yang dilakukan modul.
- Berbagai fungsi yang didukung dalam modul, bersama dengan parameter input/outputnya.
- Variabel global diakses/dimodifikasi oleh modul.

Konvensi penamaan untuk variabel global, variabel lokal, dan pengidentifikasi konstan: Konvensi penamaan yang populer adalah bahwa variabel diberi nama menggunakan huruf besar-kecil. Nama variabel global akan selalu dimulai dengan huruf kapital (misalnya, GlobalData) dan nama variabel lokal dimulai dengan huruf kecil (misalnya, localData). Nama konstanta harus dibentuk hanya menggunakan huruf kapital (mis., CONSTDATA).

Konvensi mengenai nilai pengembalian kesalahan dan mekanisme penanganan pengecualian: Cara kondisi kesalahan dilaporkan oleh fungsi yang berbeda dalam suatu program harus menjadi standar dalam suatu organisasi. Misalnya, semua fungsi saat menghadapi kondisi kesalahan harus mengembalikan 0 atau 1 secara konsisten, terlepas dari programmer mana yang telah menulis kodenya. Ini **memfasilitasi penggunaan kembali dan debugging.**

Panduan pengkodean representatif: Berikut ini adalah beberapa panduan pengkodean representatif yang direkomendasikan oleh banyak organisasi pengembangan perangkat lunak. Bila perlu, alasan di balik pedoman ini juga disebutkan.

Jangan gunakan gaya pengkodean yang terlalu pintar atau terlalu sulit untuk dipahami: Kode harus mudah dipahami. Banyak insinyur yang tidak berpengalaman benar-benar bangga dalam menulis kode yang samar dan tidak dapat dipahami. Pengkodean yang cerdas dapat mengaburkan makna kode dan mengurangi pemahaman kode; sehingga membuat pemeliharaan dan debugging menjadi sulit dan mahal.

Hindari efek samping yang tidak jelas: Efek samping dari pemanggilan fungsi mencakup modifikasi parameter yang diteruskan dengan referensi, modifikasi variabel global, dan operasi I/O. Efek samping yang tidak jelas adalah salah satu yang tidak terlihat dari pemeriksaan biasa terhadap kode tersebut. Efek samping yang tidak jelas membuat sulit untuk memahami sepotong kode. Sebagai contoh, misalkan nilai variabel global diubah atau beberapa file I/O dijalankan secara tidak jelas dalam modul yang dipanggil. Artinya, ini sulit untuk disimpulkan dari nama fungsi dan informasi header. Kemudian, akan sangat sulit untuk memahami kodenya.

Jangan gunakan pengenal untuk beberapa tujuan: Programmer sering menggunakan pengenal yang sama untuk menunjukkan beberapa entitas sementara. Misalnya, beberapa programmer menggunakan variabel loop sementara untuk juga menghitung dan menyimpan hasil akhir. Alasan yang mereka berikan untuk beberapa penggunaan variabel tersebut adalah efisiensi memori, misalnya, tiga variabel menggunakan tiga lokasi memori, sedangkan ketika variabel yang sama digunakan untuk tiga tujuan berbeda, hanya satu lokasi memori yang digunakan. Namun, ada beberapa hal yang salah dengan pendekatan ini dan karenanya harus dihindari. Beberapa masalah yang disebabkan oleh penggunaan variabel untuk berbagai tujuan adalah sebagai berikut:

- Setiap variabel harus diberi nama deskriptif yang menunjukkan tujuannya. Ini tidak mungkin jika pengenal digunakan untuk berbagai tujuan. Penggunaan variabel untuk berbagai tujuan dapat menyebabkan kebingungan dan mempersulit seseorang yang mencoba membaca dan memahami kode.
- Penggunaan variabel untuk berbagai tujuan biasanya membuat perangkat tambahan di masa depan lebih sulit. Misalnya, saat mengubah hasil akhir yang

dihitung dari integer ke tipe float, programmer selanjutnya mungkin memperhatikan bahwa itu juga telah digunakan sebagai variabel loop sementara yang tidak bisa menjadi tipe float.

Kode harus didokumentasikan dengan baik: Sebagai aturan praktis, harus ada setidaknya satu baris komentar rata-rata untuk setiap tiga baris kode sumber.

Panjang fungsi apa pun tidak boleh melebihi 10 baris sumber: Fungsi yang panjang biasanya sangat sulit dipahami karena mungkin memiliki banyak variabel dan melakukan berbagai jenis perhitungan. Untuk alasan yang sama, fungsi yang panjang cenderung memiliki jumlah bug yang jauh lebih besar.

Jangan gunakan pernyataan GO TO: Penggunaan pernyataan GO TO membuat program tidak terstruktur. Hal ini membuat program sangat sulit untuk dipahami, di-debug, dan dipelihara.

10.2 KODE REVIEW

Pengujian adalah mekanisme penghilangan cacat yang efektif. Namun, pengujian hanya berlaku untuk kode yang dapat dieksekusi. Review adalah teknik yang sangat efektif untuk menghilangkan cacat dari kode sumber. Faktanya, peninjauan telah diakui lebih hemat biaya dalam menghilangkan cacat dibandingkan dengan pengujian. Selama bertahun-tahun, teknik peninjauan telah menjadi sangat populer dan telah digeneralisasi untuk digunakan dengan produk kerja lainnya.

Review kode untuk sebuah modul dilakukan setelah modul berhasil dikompilasi. Artinya, semua kesalahan sintaks telah dihilangkan dari modul. Jelas, tinjauan kode tidak menargetkan untuk merancang kesalahan sintaks dalam suatu program, tetapi dirancang untuk mendeteksi kesalahan logis, algoritmik, dan pemrograman. Peninjauan kode telah diakui sebagai strategi yang sangat hemat biaya untuk menghilangkan kesalahan pengkodean dan untuk menghasilkan kode berkualitas tinggi. Alasan di balik mengapa tinjauan kode adalah strategi yang jauh lebih hemat biaya untuk menghilangkan kesalahan dari kode dibandingkan dengan pengujian adalah bahwa tinjauan langsung mendeteksi kesalahan. Di sisi lain, pengujian hanya membantu mendeteksi kegagalan dan upaya yang signifikan diperlukan untuk menemukan kesalahan selama debugging.

Alasan di balik pernyataan di atas dijelaskan sebagai berikut. Menghilangkan kesalahan dari kode melibatkan tiga aktivitas utama—pengujian, debugging, dan kemudian memperbaiki kesalahan. Pengujian dilakukan untuk mendeteksi jika sistem gagal bekerja secara memuaskan untuk jenis input tertentu dan dalam keadaan tertentu. Setelah kegagalan terdeteksi, debugging dilakukan untuk menemukan kesalahan yang menyebabkan kegagalan dan untuk menghapusnya. Dari tiga aktivitas pengujian, debugging mungkin merupakan aktivitas yang paling melelahkan dan memakan waktu. Dalam pemeriksaan kode, kesalahan terdeteksi secara langsung, sehingga menghemat upaya signifikan yang diperlukan untuk menemukan kesalahan.

Biasanya, dua jenis tinjauan berikut dilakukan pada kode modul:

- Pemeriksaan kode.
- Panduan kode.

Prosedur pelaksanaan dan tujuan akhir dari kedua teknik peninjauan ini sangat berbeda.

Panduan Kode

Panduan kode adalah teknik analisis kode informal. Dalam teknik ini, sebuah modul diambil untuk ditinjau setelah modul dikodekan, berhasil dikompilasi, dan semua kesalahan sintaks telah dihilangkan. Beberapa anggota tim pengembangan diberi kode beberapa hari

sebelum rapat panduan. Setiap anggota memilih beberapa kasus uji dan mensimulasikan eksekusi kode dengan tangan (yaitu, melacak eksekusi melalui pernyataan dan fungsi kode yang berbeda). Tujuan utama dari panduan kode adalah untuk menemukan kesalahan algoritmik dan logis dalam kode. Para anggota mencatat temuan mereka dari panduan mereka dan mendiskusikannya dalam pertemuan panduan di mana pembuat kode modul hadir. Meskipun panduan kode adalah teknik analisis informal, beberapa pedoman telah berkembang selama bertahun-tahun untuk membuat teknik analisis yang naif tetapi berguna ini lebih efektif. Pedoman ini didasarkan pada pengalaman pribadi, akal sehat, beberapa faktor subjektif lainnya. Oleh karena itu, pedoman ini harus dianggap sebagai contoh daripada sebagai aturan yang diterima untuk diterapkan secara dogmatis. Beberapa pedoman tersebut adalah sebagai berikut:

- Tim yang melakukan penelusuran kode tidak boleh terlalu besar atau terlalu kecil. Idealnya, harus terdiri dari tiga hingga tujuh anggota.
- Diskusi harus fokus pada penemuan kesalahan dan menghindari pertimbangan tentang cara memperbaiki kesalahan yang ditemukan.
- Untuk mendorong kerjasama dan untuk menghindari perasaan di antara para insinyur bahwa mereka sedang diawasi dan dievaluasi dalam pertemuan panduan kode, manajer tidak boleh menghadiri pertemuan panduan.

Inspeksi Kode

Selama pemeriksaan kode, kode diperiksa untuk mengetahui adanya beberapa kesalahan pemrograman umum. Ini berbeda dengan simulasi tangan dari eksekusi kode yang dilakukan selama penelusuran kode. Tujuan utama dari pemeriksaan kode dapat dinyatakan seperti berikut ini:

Tujuan utama dari pemeriksaan kode adalah untuk memeriksa keberadaan beberapa jenis kesalahan umum yang biasanya menyusup ke dalam kode karena kesalahan dan kelalaian programmer dan untuk memeriksa apakah standar pengkodean telah dipatuhi.

Proses pemeriksaan memiliki beberapa efek samping yang menguntungkan, selain menemukan kesalahan. Programmer biasanya menerima umpan balik tentang gaya pemrograman, pilihan algoritma, dan teknik pemrograman. Peserta lain mendapatkan keuntungan dengan terkena kesalahan programmer lain.

Sebagai contoh jenis kesalahan yang terdeteksi selama pemeriksaan kode, pertimbangkan kesalahan klasik dalam menulis prosedur yang memodifikasi parameter formal dan kemudian menyebutnya dengan parameter aktual yang konstan. Kemungkinan besar kesalahan seperti itu dapat ditemukan dengan secara khusus mencari kesalahan semacam ini dalam kode, daripada hanya dengan mensimulasikan eksekusi kode secara manual. Selain kesalahan yang sering dibuat, kepatuhan terhadap standar pengkodean juga diperiksa selama pemeriksaan kode.

Perusahaan pengembangan perangkat lunak yang baik mengumpulkan statistik mengenai berbagai jenis kesalahan yang biasanya dilakukan oleh para insinyur mereka dan mengidentifikasi jenis kesalahan yang paling sering dilakukan. Daftar kesalahan yang sering dilakukan seperti itu dapat digunakan sebagai daftar periksa selama pemeriksaan kode untuk mencari kemungkinan kesalahan. Berikut adalah daftar beberapa kesalahan pemrograman klasik yang dapat diperiksa selama pemeriksaan kode:

- Penggunaan variabel yang tidak diinisialisasi.
- Melompat ke loop.
- Loop yang tidak berakhir.
- Tugas yang tidak kompatibel.
- Indeks array di luar batas.

- Alokasi penyimpanan dan dealokasi yang tidak tepat.
- Ketidakcocokan antara parameter aktual dan formal dalam panggilan prosedur.
- Penggunaan operator logika yang salah atau prioritas yang salah antar operator.
- Modifikasi variabel loop yang tidak tepat.
- Perbandingan kesetaraan nilai floating point.
- Referensi menggantung disebabkan ketika memori yang direferensikan belum dialokasikan.

Pengujian Kamar Bersih

Pengujian kamar bersih dirintis di IBM. Jenis pengujian ini sangat bergantung pada penelusuran, inspeksi, dan verifikasi formal. Programmer tidak diperbolehkan untuk menguji kode mereka dengan mengeksekusi kode selain melakukan beberapa pengujian sintaks menggunakan kompiler. Sangat menarik untuk dicatat bahwa istilah cleanroom pertama kali diciptakan di IBM dengan menggambar analogi ke unit fabrikasi semikonduktor di mana cacat dihindari dengan manufaktur di atmosfer ultra-bersih.

Teknik ini dilaporkan menghasilkan dokumentasi dan kode yang lebih andal dan dapat dipelihara daripada metode pengembangan lain yang sangat bergantung pada pengujian berbasis eksekusi kode. Masalah utama dengan pendekatan ini adalah bahwa upaya pengujian meningkat karena penelusuran, inspeksi, dan verifikasi memakan waktu lama untuk mendeteksi semua kesalahan sederhana. Deteksi kesalahan berbasis pengujian juga efisien untuk mendeteksi kesalahan tertentu yang luput dari pemeriksaan manual.

10.3 DOKUMENTASI PERANGKAT LUNAK

Ketika perangkat lunak dikembangkan, selain file yang dapat dieksekusi dan kode sumber, beberapa jenis dokumen seperti manual pengguna, dokumen spesifikasi persyaratan perangkat lunak (SRS), dokumen desain, dokumen uji, manual instalasi, dll., Dikembangkan sebagai bagian dari proses rekayasa perangkat lunak. Semua dokumen ini dianggap sebagai bagian penting dari praktik pengembangan perangkat lunak yang baik. Dokumen yang baik sangat membantu dengan cara berikut:

- Dokumen yang baik membantu meningkatkan pemahaman kode. Akibatnya, ketersediaan dokumen yang baik membantu mengurangi upaya dan waktu yang diperlukan untuk pemeliharaan.
- Dokumen membantu pengguna untuk memahami dan menggunakan sistem secara efektif.
- Dokumen yang baik membantu mengatasi masalah pergantian tenaga kerja1 secara efektif. Bahkan ketika seorang insinyur meninggalkan organisasi, dan seorang insinyur baru masuk, dia dapat membangun pengetahuan yang dibutuhkan dengan mudah dengan mengacu pada dokumen.
- Produksi dokumen yang baik membantu manajer untuk secara efektif melacak kemajuan proyek. Manajer proyek akan mengetahui bahwa beberapa kemajuan terukur telah dicapai, jika hasil dari beberapa bagian pekerjaan telah didokumentasikan dan hal yang sama telah ditinjau.

Berbagai jenis dokumen perangkat lunak secara luas dapat diklasifikasikan menjadi berikut:

- **Dokumentasi internal**: Ini disediakan dalam kode sumber itu sendiri.
- **Dokumentasi eksternal**: Ini adalah dokumen pendukung seperti dokumen SRS, dokumen instalasi, manual pengguna, dokumen desain, dan dokumen uji.

Dua jenis dokumentasi ini akan dibahas di bagian selanjutnya.

Dokumentasi Internal

Dokumentasi internal adalah fitur pemahaman kode yang disediakan dalam kode sumber itu sendiri. Dokumentasi internal dapat disediakan dalam kode dalam beberapa bentuk. Jenis penting dari dokumentasi internal adalah sebagai berikut:

- Komentar tertanam dalam kode sumber.
- Penggunaan nama variabel yang bermakna.
- Header modul dan fungsi.
- Indentasi kode.
- Penataan kode (yaitu, kode didekomposisi menjadi modul dan fungsi).
- Penggunaan tipe enumerasi.
- Penggunaan pengidentifikasi konstan.
- Penggunaan tipe data yang ditentukan pengguna.

Dari berbagai jenis dokumentasi internal ini, mana yang paling berharga untuk memahami sepotong kode? Eksperimen yang cermat menunjukkan bahwa dari semua jenis dokumentasi internal, nama variabel yang bermakna paling berguna saat mencoba memahami sepotong kode. Pernyataan di atas, tentu saja, bertentangan dengan harapan umum bahwa komentar kode akan menjadi yang paling berguna. Temuan penelitian jelas benar ketika komentar ditulis tanpa banyak berpikir. Misalnya, gaya komentar kode berikut tidak banyak membantu dalam memahami kode.

```
a=10; /* dibuat 10 */
```

Gaya komentar kode yang baik adalah menulis untuk mengklarifikasi aspek tertentu yang tidak jelas dari cara kerja kode, daripada mengacaukan kode dengan komentar sepele. Organisasi pengembangan perangkat lunak yang baik biasanya memastikan dokumentasi internal yang baik dengan merumuskan standar pengkodean dan pedoman pengkodean secara tepat. Bahkan ketika sepotong kode dikomentari dengan hati-hati, nama variabel yang bermakna ternyata paling membantu dalam memahami kode.

Dokumentasi Eksternal

Dokumentasi eksternal disediakan melalui berbagai jenis dokumen pendukung seperti manual pengguna, dokumen spesifikasi kebutuhan perangkat lunak, dokumen desain, dokumen uji, dll. Gaya pengembangan perangkat lunak yang sistematis memastikan bahwa semua dokumen ini berkualitas baik dan diproduksi secara teratur .

Fitur penting yang diperlukan untuk dokumentasi eksternal yang baik adalah konsistensi dengan kode. Jika dokumen yang berbeda tidak konsisten, banyak kebingungan dibuat untuk seseorang yang mencoba memahami perangkat lunak. Dengan kata lain, semua dokumen yang dikembangkan untuk suatu produk harus mutakhir dan setiap perubahan yang dilakukan pada kode harus tercermin dalam dokumen eksternal yang relevan. Bahkan jika hanya beberapa dokumen yang tidak mutakhir, mereka menciptakan inkonsistensi dan menyebabkan kebingungan. Fitur penting lainnya yang diperlukan untuk dokumen eksternal adalah pemahaman yang tepat berdasarkan kategori pengguna untuk siapa dokumen tersebut dirancang. Untuk mencapai ini, indeks kabut Gunning sangat berguna. Kita bahas ini selanjutnya.

Indeks kabut Gunning

Indeks kabut Gunning (dikembangkan oleh Robert Gunning pada tahun 1952) adalah metrik yang telah dirancang untuk mengukur keterbacaan dokumen. Nilai metrik yang dihitung (indeks kabut) dari suatu dokumen menunjukkan jumlah tahun pendidikan formal yang harus dimiliki seseorang, agar dapat memahami dokumen itu dengan nyaman. Artinya, jika dokumen tertentu memiliki indeks kabut 12, siapa pun yang telah menyelesaikan kelas 12-nya tidak akan mengalami banyak kesulitan dalam memahami dokumen itu. Indeks kabut Gunning dari dokumen D dapat dihitung sebagai berikut:

$$Kabut\ (D) = 0.4\ x\ \left(\frac{kat}{kalimat}\right) - persen\ kata\ yang\ memiliki\ silabel\ 3\ atau\ lebih$$

Perhatikan bahwa indeks kabut dihitung sebagai jumlah dari dua faktor yang berbeda.

Faktor pertama menghitung jumlah rata-rata kata per kalimat (jumlah total kata dalam dokumen dibagi dengan jumlah total kalimat). Oleh karena itu, faktor ini menjelaskan pengamatan umum bahwa kalimat yang panjang sulit dipahami. Faktor kedua mengukur persentase kata kompleks dalam dokumen. Perhatikan bahwa suku kata adalah sekelompok kata yang dapat diucapkan secara independen. Misalnya, kata "kalimat" memiliki tiga suku kata ("sen", "sepuluh", dan "ce"). Kata-kata yang memiliki lebih dari tiga suku kata adalah kata-kata yang kompleks dan kehadiran banyak kata seperti itu menghambat keterbacaan dokumen.

10.4 PENGUJIAN

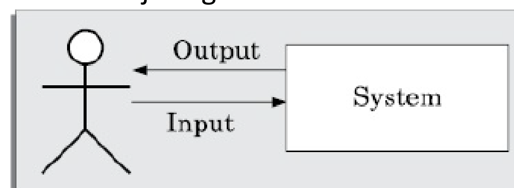
Tujuan dari pengujian program adalah untuk membantu mengidentifikasi semua cacat dalam program. Namun, dalam praktiknya, bahkan setelah penyelesaian fase pengujian yang memuaskan, tidak mungkin untuk menjamin bahwa suatu program bebas dari kesalahan. Ini karena domain data input dari sebagian besar program sangat besar, dan tidak praktis untuk menguji program secara mendalam sehubungan dengan setiap nilai yang dapat diasumsikan oleh input. Pertimbangkan fungsi yang mengambil angka floating point sebagai argumen. Jika seorang penguji membutuhkan waktu 1 detik untuk mengetikkan suatu nilai, maka sejuta penguji pun tidak akan dapat mengujinya secara mendalam setelah mencoba selama jutaan tahun. Bahkan dengan batasan yang jelas dari proses pengujian ini, kita tidak boleh meremehkan pentingnya pengujian. Kita harus ingat bahwa pengujian yang cermat dapat mengekspos sebagian besar cacat yang ada dalam suatu program, dan karena itu menyediakan cara praktis untuk mengurangi cacat dalam suatu sistem.

Konsep Dasar dan Terminologi

Pada bagian ini, kita akan membahas beberapa konsep dasar dalam pengujian program yang menjadi dasar diskusi selanjutnya tentang pengujian program.

Bagaimana cara menguji sebuah program?

Pengujian suatu program melibatkan pelaksanaan program dengan serangkaian input pengujian dan mengamati apakah program berperilaku seperti yang diharapkan. Jika program gagal berperilaku seperti yang diharapkan, maka data input dan kondisi kegagalannya dicatat untuk debugging nanti dan koreksi kesalahan. Tampilan pengujian program yang sangat disederhanakan secara skematis ditunjukkan pada Gambar 10.1. Penguji telah ditampilkan sebagai ikon tongkat, yang memasukkan beberapa data uji ke sistem dan mengamati output yang dihasilkannya untuk memeriksa apakah sistem gagal pada beberapa input tertentu. Kecuali kondisi di mana perangkat lunak gagal dicatat, menjadi sulit bagi developer untuk mereproduksi kegagalan yang diamati oleh penguji. Misalnya, perangkat lunak mungkin gagal untuk kasus uji hanya ketika koneksi jaringan diaktifkan.



Gambar 10.1 Tampilan pengujian program yang disederhanakan.

Terminologi

Seperti yang berlaku untuk domain khusus apa pun, area pengujian perangkat lunak telah dikaitkan dengan serangkaian terminologinya sendiri. Berikut ini akan dibahas beberapa terminologi penting yang telah dibakukan oleh IEEE Standard Glossary of Software Engineering Terminology [IEEE90]:

- Kesalahan pada dasarnya adalah setiap tindakan programmer yang kemudian muncul sebagai hasil yang salah selama eksekusi program. Seorang programmer dapat melakukan kesalahan di hampir semua aktivitas pengembangan. Misalnya, selama pengkodean, seorang programmer mungkin melakukan kesalahan dengan tidak menginisialisasi variabel tertentu, atau mungkin mengabaikan kesalahan yang mungkin muncul dalam beberapa situasi luar biasa seperti pembagian dengan nol dalam operasi aritmatika. Kedua kesalahan ini dapat menyebabkan hasil yang salah.
- Kesalahan adalah hasil dari kesalahan yang dilakukan oleh developer dalam setiap kegiatan pengembangan. Di antara berbagai kesalahan yang sangat besar yang bisa ada dalam suatu program. Salah satu contoh kesalahan adalah panggilan yang dilakukan ke fungsi yang salah.

Istilah kesalahan, kesalahan, bug, dan cacat dianggap sinonim di bidang pengujian program. Meskipun istilah error, fault, bug, dan defect semuanya digunakan secara bergantian oleh komunitas pengujian program. Harap dicatat bahwa dalam domain pengujian perangkat keras, istilah kesalahan digunakan dengan konotasi yang sedikit berbeda [IEEE90] dibandingkan dengan istilah kesalahan dan bug.

Contoh 10.2 Dapatkah kesalahan perancang menimbulkan kesalahan program? Berikan contoh kesalahan perancang dan kesalahan program yang sesuai.

Jawaban: Ya, kesalahan desainer menimbulkan kesalahan program. Misalnya, suatu persyaratan mungkin diabaikan oleh perancang, yang dapat menyebabkannya diabaikan juga dalam kode.

- Kegagalan suatu program pada dasarnya menunjukkan perilaku yang salah yang ditunjukkan oleh program selama eksekusinya. Perilaku yang salah diamati baik sebagai hasil yang salah yang dihasilkan atau sebagai aktivitas yang tidak sesuai yang dilakukan oleh program. Setiap kegagalan disebabkan oleh beberapa bug yang ada dalam program. Dengan kata lain, kita dapat mengatakan bahwa setiap kegagalan perangkat lunak dapat dilacak ke beberapa bug atau hal lain yang ada dalam kode. Jumlah kemungkinan cara sebuah program bisa gagal sangat besar. Dari sekian banyak cara di mana sebuah program bisa gagal, berikut ini merupakan tiga contoh yang dipilih secara acak:
    - Hasil yang dihitung oleh program adalah 0, ketika hasil yang benar adalah 10.
    - Sebuah program crash pada input.
    - Robot gagal menghindari rintangan dan bertabrakan dengannya.

Perlu dicatat bahwa hanya adanya kesalahan dalam kode program belum tentu menyebabkan kegagalan selama eksekusi.

Contoh 10.3 Berikan contoh kesalahan program yang mungkin tidak menyebabkan kegagalan.

Jawaban: Perhatikan segmen program C berikut:

```
int markList[1:10]; /* mark list of 10 students*/
int roll;           /* student roll number*/
        ...
if(roll>0)
        markList[roll]=mark;
else
        markList[roll]=0;
```

Dalam kode di atas, jika gulungan variabel mengasumsikan nol atau beberapa nilai negatif dalam beberapa keadaan, maka indeks array dari jenis kesalahan terikat akan dihasilkan. Namun, mungkin saja untuk semua nilai input yang diizinkan, gulungan variabel selalu diberi nilai positif. Kemudian, klausa else tidak dapat dijangkau dan tidak ada kegagalan yang akan terjadi. Jadi, bahkan jika ada kesalahan dalam kode, itu tidak muncul sebagai kesalahan karena tidak dapat dijangkau untuk nilai input normal.

Penjelasan: Sebuah indeks array dari jenis kesalahan terikat dikatakan terjadi, ketika variabel indeks array mengasumsikan nilai di luar batas array.

- **Kasus uji** adalah triplet [I , S, R], di mana I adalah input data ke program yang diuji, S adalah status program di mana data akan dimasukkan, dan R adalah hasil yang diharapkan. dihasilkan oleh program. Status program juga disebut mode eksekusi. Sebagai contoh, pertimbangkan mode eksekusi yang berbeda dari perangkat lunak editor teks tertentu. Editor teks dapat setiap saat selama eksekusinya mengambil salah satu mode eksekusi berikut—edit, lihat, buat, dan tampilkan. Dengan kata sederhana, kita dapat mengatakan bahwa test case adalah satu set input tes, mode di mana input akan diterapkan, dan hasil yang diharapkan selama dan setelah eksekusi kasus uji.

Contoh kasus uji adalah [input: "abc", state: edit, result: abc ditampilkan], yang pada dasarnya berarti bahwa input abc perlu diterapkan dalam mode edit, dan hasil yang diharapkan adalah string a b c akan ditampilkan.

- **Skenario pengujian** adalah kasus pengujian abstrak dalam arti hanya mengidentifikasi aspek program yang akan diuji tanpa mengidentifikasi input, status, atau output. Kasus uji dapat dikatakan sebagai implementasi dari skenario uji. Dalam kasus uji, input, output, dan status di mana input akan diterapkan dirancang sedemikian rupa sehingga skenario dapat dieksekusi. Strategi desain kasus uji otomatis yang penting adalah pertama-tama merancang skenario pengujian melalui analisis beberapa abstraksi program (model) dan kemudian mengimplementasikan skenario pengujian sebagai kasus uji.
- **Skrip pengujian** adalah pengkodean kasus uji sebagai program pendek. Skrip pengujian dikembangkan untuk eksekusi otomatis kasus uji.
- Kasus uji dikatakan sebagai kasus uji positif jika dirancang untuk menguji apakah perangkat lunak melakukan fungsionalitas yang diperlukan dengan benar. Test case dikatakan negatif test case, jika dirancang untuk menguji apakah perangkat lunak melakukan sesuatu, yang tidak diperlukan dari sistem. Sebagai salah satu contoh masing-masing kasus uji positif dan kasus uji negatif, pertimbangkan sebuah program untuk mengelola login pengguna. Kasus uji positif dapat dirancang untuk memeriksa apakah sistem login memvalidasi pengguna dengan nama pengguna dan kata sandi yang benar. Kasus uji negatif dalam kasus ini dapat berupa kasus uji yang memeriksa apakah fungsi login memvalidasi dan menerima pengguna dengan nama pengguna atau sandi login yang salah atau palsu.

- **Test suite** adalah kumpulan semua tes yang telah dirancang oleh penguji untuk menguji program yang diberikan.
- **Testabilitas** persyaratan menunjukkan sejauh mana dimungkinkan untuk menentukan apakah implementasi persyaratan sesuai dengan itu baik dalam fungsionalitas dan kinerja. Dengan kata lain, testabilitas suatu persyaratan adalah sejauh mana implementasinya dapat diuji secara memadai untuk menentukan kesesuaiannya dengan persyaratan.

Contoh 10.4 Misalkan dua program telah ditulis untuk mengimplementasikan fungsionalitas yang pada dasarnya sama. Bagaimana Anda bisa menentukan mana yang lebih dapat diuji?

Jawaban: Suatu program lebih dapat diuji, jika dapat diuji secara memadai dengan jumlah kasus uji yang lebih sedikit. Jelas, program yang kurang kompleks lebih dapat diuji. Kompleksitas suatu program dapat diukur dengan menggunakan beberapa jenis metrik seperti jumlah pernyataan keputusan yang digunakan dalam program tersebut. Dengan demikian, program yang lebih dapat diuji harus memiliki metrik kompleksitas struktural yang lebih rendah.

- **Mode kegagalan** perangkat lunak menunjukkan cara yang dapat diamati di mana ia bisa gagal. Dengan kata lain, semua kegagalan yang memiliki gejala serupa yang dapat diamati, merupakan mode kegagalan. Sebagai contoh mode kegagalan perangkat lunak, pertimbangkan perangkat lunak pemesanan tiket kereta api yang memiliki tiga mode kegagalan—gagal memesan tempat duduk yang tersedia, salah memesan tempat duduk (misalnya, memesan tempat duduk yang sudah dipesan), dan sistem macet.
- **Kesalahan setara** menunjukkan dua atau lebih bug yang mengakibatkan sistem gagal dalam mode kegagalan yang sama. Sebagai contoh kesalahan yang setara, pertimbangkan dua kesalahan berikut dalam bahasa C — pembagian dengan nol dan kesalahan akses memori ilegal. Kedua kesalahan ini setara, karena masing-masing menyebabkan crash program.

Verifikasi versus validasi

Tujuan dari teknik verifikasi dan validasi sangat mirip karena kedua teknik ini dirancang untuk membantu menghilangkan kesalahan dalam perangkat lunak. Terlepas dari kesamaan yang tampak antara tujuannya, prinsip-prinsip yang mendasari kedua teknik deteksi bug ini dan penerapannya sangat berbeda. Perbedaan utama antara kedua teknik ini adalah sebagai berikut:

- Verifikasi adalah proses menentukan apakah output dari satu fase pengembangan perangkat lunak sesuai dengan fase sebelumnya; sedangkan validasi adalah proses menentukan apakah perangkat lunak yang dikembangkan sepenuhnya sesuai dengan spesifikasi persyaratannya. Jadi, tujuan verifikasi adalah untuk memeriksa apakah produk kerja yang dihasilkan setelah suatu fase sesuai dengan yang dimasukkan ke fase tersebut. Misalnya, langkah verifikasi dapat untuk memeriksa apakah dokumen desain yang dihasilkan setelah langkah desain sesuai dengan spesifikasi persyaratan. Di sisi lain, validasi diterapkan pada perangkat lunak yang sepenuhnya dikembangkan dan terintegrasi untuk memeriksa apakah itu memenuhi kebutuhan pelanggan.
- Teknik utama yang digunakan untuk verifikasi meliputi review, simulasi, verifikasi formal, dan pengujian. Tinjauan, simulasi, dan pengujian biasanya dianggap sebagai teknik verifikasi informal. Verifikasi formal biasanya melibatkan penggunaan teknik pembuktian teorema atau penggunaan alat otomatis seperti

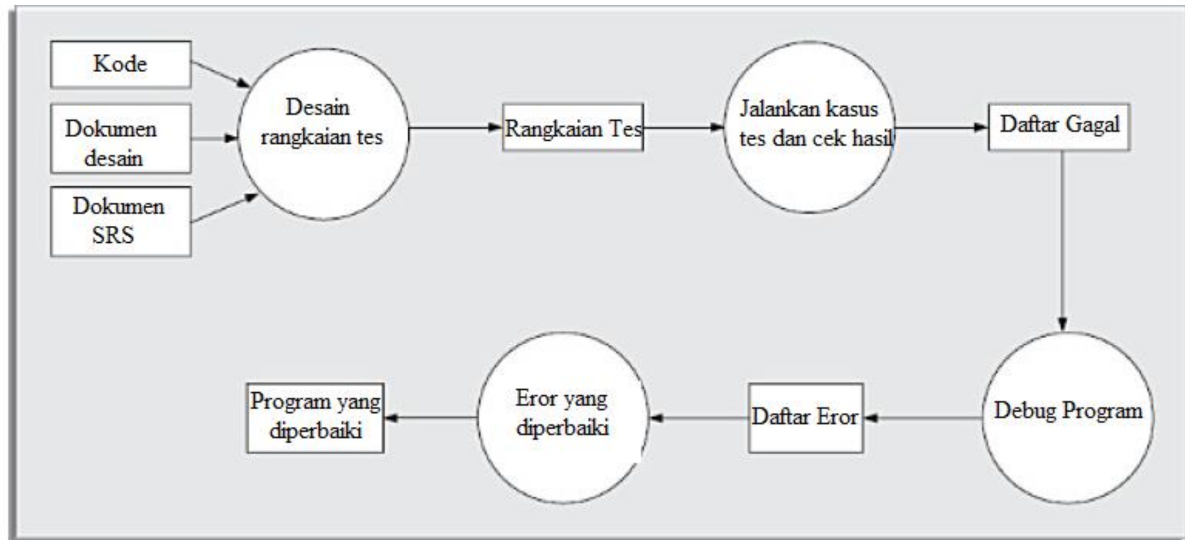
pemeriksa model. Di sisi lain, teknik validasi terutama didasarkan pada pengujian produk. Perhatikan bahwa kita mengkategorikan pengujian baik di bawah verifikasi dan validasi program. Alasannya adalah bahwa pengujian unit dan integrasi dapat dianggap sebagai langkah verifikasi di mana diverifikasi apakah kode sesuai dengan spesifikasi modul dan antarmuka modul. Di sisi lain, pengujian sistem dapat dianggap sebagai langkah validasi di mana ditentukan apakah kode yang dikembangkan sepenuhnya sesuai dengan spesifikasi persyaratannya.

- Verifikasi tidak memerlukan eksekusi perangkat lunak, sedangkan validasi membutuhkan eksekusi perangkat lunak.
- Verifikasi dilakukan selama proses pengembangan untuk memeriksa apakah kegiatan pengembangan berjalan dengan baik, sedangkan validasi dilakukan untuk memeriksa apakah hak yang diminta oleh pelanggan telah dikembangkan. Oleh karena itu kita dapat mengatakan bahwa tujuan utama dari langkah-langkah verifikasi adalah untuk menentukan apakah langkah-langkah dalam pengembangan produk dilakukan dengan baik, sedangkan validasi dilakukan menjelang akhir proses pengembangan untuk menentukan apakah produk yang tepat telah dikembangkan.
- Teknik verifikasi dapat dilihat sebagai upaya untuk mencapai fase penahanan kesalahan. Fase penahanan kesalahan telah diakui sebagai cara yang hemat biaya untuk menghilangkan bug program, dan merupakan prinsip rekayasa perangkat lunak yang penting. Prinsip mendeteksi kesalahan sedekat mungkin dengan titik komitmen mereka dikenal sebagai fase penahanan kesalahan. Fase penahanan kesalahan dapat mengurangi upaya yang diperlukan untuk memperbaiki bug. Misalnya, jika masalah desain terdeteksi dalam fase desain itu sendiri, maka masalah tersebut dapat ditangani dengan lebih mudah daripada jika kesalahan diidentifikasi, katakanlah, pada akhir fase pengujian. Dalam kasus selanjutnya, akan diperlukan tidak hanya untuk mendesain ulang, tetapi juga untuk mengulang pengkodean yang relevan serta aktivitas pengujian sistem, sehingga menimbulkan biaya yang lebih tinggi.

Sementara verifikasi berkaitan dengan fase penahanan kesalahan, tujuan validasi adalah untuk memeriksa apakah perangkat lunak yang dapat dikirimkan bebas dari kesalahan. Kita dapat menganggap teknik verifikasi dan validasi sebagai jenis filter bug yang berbeda. Untuk mencapai keandalan produk yang tinggi dengan cara yang hemat biaya, tim pengembangan perlu melakukan aktivitas verifikasi dan validasi. Aktivitas yang terlibat dalam dua jenis teknik deteksi bug ini bersama-sama disebut aktivitas “V dan V”. Berdasarkan pembahasan di atas, dapat disimpulkan bahwa: *Teknik deteksi kesalahan = Teknik verifikasi + Teknik validasi*

Contoh 10.5 Apakah mungkin untuk mengembangkan perangkat lunak yang sangat andal, menggunakan teknik validasi saja? Jika demikian, dapatkah kita mengatakan bahwa semua teknik verifikasi berlebihan?

Jawaban: Dimungkinkan untuk mengembangkan perangkat lunak yang sangat andal hanya dengan menggunakan teknik validasi. Namun, ini akan menyebabkan biaya pengembangan meningkat secara drastis. Teknik verifikasi membantu mencapai fase penahanan kesalahan dan menyediakan sarana untuk menghilangkan bug dengan biaya yang efektif.



Gambar 10.2 Proses pengujian.

Aktivitas Pengujian

Pengujian melibatkan melakukan kegiatan utama berikut:

- **Desain rangkaian uji:** Kumpulan kasus uji yang menggunakan program yang akan diuji dirancang mungkin menggunakan beberapa teknik desain kasus uji.
- **Menjalankan test case dan memeriksa hasilnya untuk mendeteksi kegagalan**: Setiap test case dijalankan dan hasilnya dibandingkan dengan hasil yang diharapkan. Ketidaksesuaian antara hasil aktual dan hasil yang diharapkan menunjukkan kegagalan. Kasus uji di mana sistem gagal dicatat untuk debugging nanti.
- **Temukan kesalahan**: Dalam aktivitas ini, gejala kegagalan dianalisis untuk menemukan kesalahan. Untuk setiap kegagalan yang diamati selama aktivitas sebelumnya, pernyataan yang salah diidentifikasi.
- **Koreksi kesalahan**: Setelah kesalahan ditemukan selama debugging, kode diubah dengan tepat untuk memperbaiki kesalahan.

Kegiatan pengujian telah ditunjukkan secara skematis pada Gambar 10.2. Seperti yang dapat dilihat, kasus uji dirancang terlebih dahulu, kasus uji dijalankan untuk mendeteksi kegagalan. Bug yang menyebabkan kegagalan diidentifikasi melalui debugging, dan kesalahan yang diidentifikasi diperbaiki. Dari semua aktivitas pengujian yang disebutkan di atas, debugging sering kali menjadi aktivitas yang paling memakan waktu.

Mengapa Desain Kasus Uji?

Sebelum membahas berbagai teknik desain kasus uji, kita perlu meyakinkan diri pada pertanyaan berikut. Apakah tidak cukup untuk menguji perangkat lunak menggunakan sejumlah besar nilai input acak? Mengapa merancang kasus uji? Jawaban atas pertanyaan ini—ini akan sangat mahal dan pada saat yang sama cara pengujian yang sangat tidak efektif karena alasan berikut:

Ketika kasus uji dirancang berdasarkan data input acak, banyak kasus uji tidak berkontribusi pada pentingnya rangkaian uji, Artinya, kasus uji tidak membantu mendeteksi cacat tambahan yang belum terdeteksi oleh kasus uji lain dalam rangkaian.

Menguji perangkat lunak menggunakan kumpulan besar kasus uji yang dipilih secara acak tidak menjamin bahwa semua (atau bahkan sebagian besar) kesalahan dalam sistem akan terungkap. Mari kita coba memahami mengapa jumlah kasus uji acak dalam rangkaian uji, secara umum, tidak menunjukkan efektivitas pengujian. Perhatikan contoh segmen kode

berikut yang menentukan nilai terbesar dari dua bilangan bulat x dan y. Segmen kode ini memiliki kesalahan pemrograman sederhana:

```
if (x>y) max = x;
else max = x
```

Untuk segmen kode yang diberikan, rangkaian pengujian {(x=3,y=2);(x=2,y=3)} dapat mendeteksi kesalahan, sedangkan rangkaian pengujian yang lebih besar {(x=3,y=2) ;(x=4,y=3); (x=5,y=1)} tidak mendeteksi kesalahan. Semua kasus pengujian dalam rangkaian pengujian yang lebih besar membantu mendeteksi kesalahan yang sama, sementara kesalahan lain dalam kode tetap tidak terdeteksi. Jadi, tidak benar untuk mengatakan bahwa test suite yang lebih besar akan selalu mendeteksi lebih banyak kesalahan daripada yang lebih kecil, kecuali tentu saja test suite yang lebih besar juga telah dirancang dengan hati-hati. Ini menyiratkan bahwa untuk pengujian yang efektif, rangkaian pengujian harus dirancang dengan hati-hati daripada dipilih secara acak.

Pengujian menyeluruh dari hampir semua sistem nontrivial tidak praktis karena fakta bahwa domain nilai data input untuk sebagian besar sistem perangkat lunak praktis sangat besar atau tak terbatas. Oleh karena itu, untuk menguji perangkat lunak secara memuaskan dengan biaya minimum, kita harus merancang rangkaian pengujian minimal yang berukuran wajar dan dapat mengungkap sebanyak mungkin kesalahan yang ada dalam sistem. Untuk mengurangi biaya pengujian dan pada saat yang sama membuat pengujian lebih efektif, pendekatan sistematis telah dikembangkan untuk merancang rangkaian pengujian kecil yang dapat mendeteksi sebagian besar, jika tidak semua kegagalan. Test suite minimal adalah kumpulan test case yang dirancang dengan hati-hati sehingga setiap test case membantu mendeteksi kesalahan yang berbeda. Ini berbeda dengan pengujian menggunakan beberapa nilai input acak. Pada dasarnya ada dua pendekatan utama untuk merancang kasus uji secara sistematis:

- Pendekatan Black-box
- Pendekatan White-box (atau glass-box)

Dalam pendekatan kotak hitam, kasus uji dirancang hanya dengan menggunakan spesifikasi fungsional perangkat lunak. Artinya, kasus uji dirancang semata-mata berdasarkan analisis perilaku input/out (yaitu, perilaku fungsional) dan tidak memerlukan pengetahuan tentang struktur internal program. Untuk alasan ini, pengujian kotak hitam juga dikenal sebagai pengujian fungsional. Di sisi lain, merancang kasus uji kotak putih memerlukan pengetahuan menyeluruh tentang struktur internal suatu program, dan oleh karena itu pengujian kotak putih juga disebut pengujian struktural. Kasus uji kotak hitam dirancang semata-mata berdasarkan perilaku input-output suatu program. Sebaliknya, kasus uji kotak putih didasarkan pada analisis kode. Kedua pendekatan untuk desain kasus uji ini saling melengkapi. Artinya, sebuah program harus diuji menggunakan kasus uji yang dirancang oleh kedua pendekatan, dan satu pengujian menggunakan satu pendekatan tidak menggantikan pengujian menggunakan yang lain.

Pengujian di Besar versus Pengujian di Kecil

Sebuah produk perangkat lunak biasanya diuji dalam tiga tingkat atau tahap:

- Pengujian unit
- Tes integrasi
- Pengujian sistem

Selama pengujian unit, fungsi individu (atau unit) dari suatu program diuji. Pengujian unit disebut sebagai pengujian dalam skala kecil, sedangkan pengujian integrasi dan sistem disebut sebagai pengujian dalam skala besar. Setelah menguji semua unit secara individual, unit secara perlahan diintegrasikan dan diuji setelah setiap langkah integrasi (pengujian

integrasi). Akhirnya, sistem yang terintegrasi penuh diuji (pengujian sistem). Integrasi dan pengujian sistem dikenal sebagai pengujian dalam skala besar.

Seringkali pemula mengajukan pertanyaan—"Mengapa menguji setiap modul (unit) secara terpisah terlebih dahulu, kemudian mengintegrasikan modul-modul ini dan menguji, dan sekali lagi menguji set modul yang terintegrasi—mengapa tidak menguji set modul yang terintegrasi sekali saja secara menyeluruh?" Jawaban atas pertanyaan ini adalah sebagai berikut—Ada dua alasan utama untuk itu. Pertama saat menguji sebuah modul, modul lain yang membutuhkan antarmuka modul ini mungkin belum siap. Selain itu, selalu merupakan ide yang baik untuk menguji modul terlebih dahulu secara terpisah sebelum integrasi karena itu membuat proses debug lebih mudah. Jika kegagalan terdeteksi ketika satu set modul terintegrasi sedang diuji, akan sulit untuk menentukan modul mana yang benar-benar memiliki kesalahan.

Pada bagian berikut, kita membahas berbagai tingkat pengujian. Harus diingat dalam semua diskusi kita selanjutnya bahwa pengujian unit dilakukan dalam fase pengkodean itu sendiri segera setelah pengkodean modul selesai. Di sisi lain, integrasi dan pengujian sistem dilakukan selama fase pengujian.

10.5 PENGUJIAN UNIT

Pengujian unit dilakukan setelah modul dikodekan dan ditinjau. Kegiatan ini biasanya dilakukan oleh pembuat kode modul itu sendiri dalam tahap pengkodean. Sebelum melakukan pengujian unit, kasus pengujian unit harus dirancang dan lingkungan pengujian untuk unit yang diuji harus dikembangkan. Pada bagian ini, pertama-tama kita membahas lingkungan yang diperlukan untuk melakukan pengujian unit.

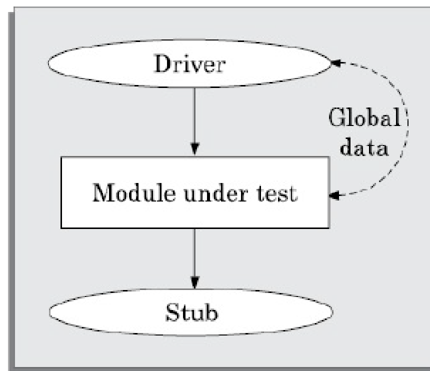
Modul driver dan rintisan

Untuk menguji satu modul, kita membutuhkan lingkungan yang lengkap untuk menyediakan semua kode yang relevan yang diperlukan untuk eksekusi modul. Artinya, selain modul yang diuji, berikut ini diperlukan untuk menguji modul:

- Prosedur milik modul lain yang dipanggil oleh modul yang sedang diuji.
- Struktur data non-lokal yang diakses modul.
- Prosedur untuk memanggil fungsi modul yang sedang diuji dengan parameter yang sesuai.

Modul yang diperlukan untuk menyediakan lingkungan yang diperlukan (yang dipanggil atau dipanggil oleh modul yang sedang diuji) biasanya tidak tersedia sampai mereka juga telah diuji unit. Dalam konteks ini, stub dan driver dirancang untuk menyediakan lingkungan yang lengkap untuk modul sehingga pengujian dapat dilakukan.

Stub: Peran modul rintisan dan driver secara bergambar ditunjukkan pada Gambar 10.3. Prosedur stub adalah prosedur dummy yang memiliki parameter I/O yang sama dengan fungsi yang dipanggil oleh unit yang diuji tetapi memiliki perilaku yang sangat disederhanakan. Misalnya, prosedur rintisan dapat menghasilkan perilaku yang diharapkan menggunakan mekanisme pencarian tabel sederhana.



Gambar 10.3 Pengujian unit dengan bantuan modul driver dan stub.

Driver: Modul driver harus berisi struktur data non-lokal yang diakses oleh modul yang diuji. Selain itu, juga harus memiliki kode untuk memanggil fungsi yang berbeda dari unit yang diuji dengan nilai parameter yang sesuai untuk pengujian.

10.6 PENGUJIAN KOTAK HITAM

Dalam pengujian kotak hitam, kasus uji dirancang dari pemeriksaan nilai input/output saja dan tidak diperlukan pengetahuan tentang desain atau kode. Berikut ini adalah dua pendekatan utama yang tersedia untuk merancang kasus uji kotak hitam:

- Equivalence Class Partitioning (ECP)
- Boundary Value Analysis (BVA)

***Equivalence Class Partitioning* (ECP)**

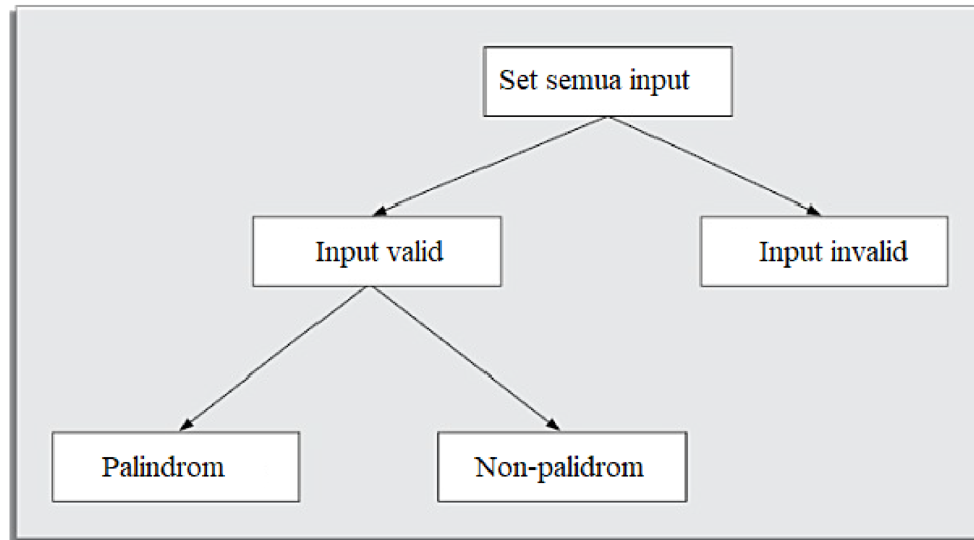
Dalam pendekatan partisi kelas kesetaraan, domain nilai input ke program yang diuji dipartisi menjadi satu set kelas kesetaraan. Partisi dilakukan sedemikian rupa sehingga untuk setiap data input yang termasuk dalam kelas ekivalensi yang sama, program berperilaku serupa. Gagasan utama di balik pendefinisian kelas ekuivalensi dari data input adalah bahwa menguji kode dengan satu nilai apa pun yang termasuk dalam kelas ekuivalensi sama baiknya dengan menguji kode dengan nilai lain apa pun yang termasuk dalam kelas ekuivalensi yang sama.

Contoh 10.6 Untuk perangkat lunak yang menghitung akar kuadrat dari bilangan bulat input yang dapat mengasumsikan nilai dalam kisaran 0 dan 5000. Tentukan kelas ekivalensi dan rangkaian uji kotak hitam.

Jawaban: Ada tiga kelas ekivalen— Himpunan bilangan bulat negatif, himpunan bilangan bulat dalam kisaran 0 dan 5000, dan himpunan bilangan bulat yang lebih besar dari 5000. Oleh karena itu, kasus uji harus menyertakan perwakilan untuk masing-masing dari tiga kelas ekivalensi. Rangkaian pengujian yang mungkin dapat berupa: {-5.500.6000}.

Contoh 10.8 Merancang rangkaian uji partisi kelas kesetaraan untuk fungsi yang membaca string karakter berukuran kurang dari lima karakter dan menampilkan apakah itu palindrom.

Jawaban: Kelas ekivalensi adalah kelas tingkat daun yang ditunjukkan pada Gambar 10.4. Kelas ekivalensinya adalah palindrom, non-palindrom, dan input yang tidak valid. Sekarang, dengan memilih satu nilai representatif dari setiap kelas ekivalensi, kita memiliki rangkaian tes yang diperlukan: {abc,aba,abcdef}.



Gambar 10.4 Kelas kesetaraan untuk Contoh 10.6.

Boundary Value Analysis **(BVA)**

Jenis kesalahan pemrograman yang sering dilakukan oleh programmer kehilangan pertimbangan khusus yang harus diberikan pada nilai pada batas kelas ekivalensi yang berbeda dari input. Alasan di balik programmer melakukan kesalahan seperti itu mungkin murni karena faktor psikologis. Programmer sering gagal untuk menangani dengan benar pemrosesan khusus yang diperlukan oleh nilai input yang terletak pada batas kelas ekivalensi yang berbeda. Misalnya, programmer mungkin salah menggunakan < daripada <=, atau sebaliknya <= untuk <, dll. Desain rangkaian pengujian berbasis analisis nilai batas melibatkan perancangan kasus uji menggunakan nilai pada batas kelas ekivalensi yang berbeda.

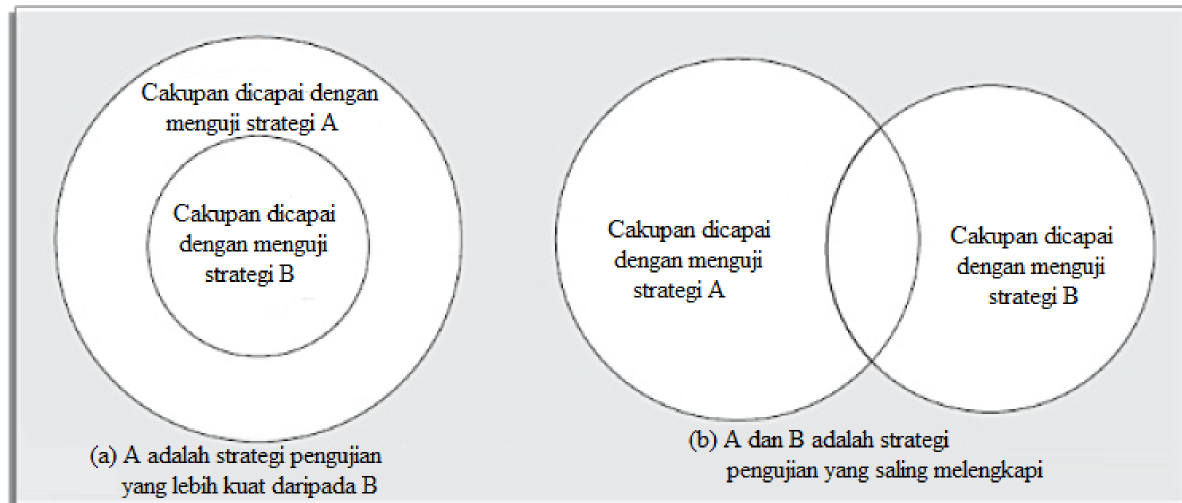
Untuk merancang kasus uji nilai batas, diperlukan untuk memeriksa kelas kesetaraan untuk memeriksa apakah ada kelas kesetaraan yang berisi rentang nilai. Untuk kelas ekivalensi yang bukan rentang nilai (yaitu, terdiri dari kumpulan nilai diskrit) tidak ada kasus uji nilai batas yang dapat didefinisikan. Untuk kelas ekivalensi yang merupakan rentang nilai, nilai batas perlu disertakan dalam rangkaian pengujian. Misalnya, jika kelas ekivalensi berisi bilangan bulat dalam rentang 1 hingga 10, maka rangkaian pengujian nilai batas adalah {0,1,10,11}.

Contoh 10.9 Untuk fungsi yang menghitung akar kuadrat dari nilai bilangan bulat dalam kisaran 0 dan 5000, tentukan rangkaian pengujian nilai batas.

Jawaban: Ada tiga kelas ekivalen— Himpunan bilangan bulat negatif, himpunan bilangan bulat dalam kisaran 0 dan 5000, dan himpunan bilangan bulat yang lebih besar dari 5000. Rangkaian pengujian berbasis nilai batas adalah: {0,-1, 5000,5001}.

Contoh 10.10 Desain rangkaian uji nilai batas untuk fungsi yang dijelaskan dalam Contoh 10.6.

Jawaban: Kelas ekivalensi ditunjukkan pada Gambar 10.5. Ada batas antara kelas kesetaraan yang valid dan tidak valid. Jadi, rangkaian pengujian nilai batas adalah {abcdefg, abcdef}.



Gambar 10.5 CFG untuk (a) urutan, (b) seleksi, dan (c) jenis iterasi konstruksi.

Ringkasan Pendekatan Desain Suite Uji Kotak Hitam

Ini merupakan langkah-langkah penting dalam pendekatan desain suite uji kotak hitam:

- Periksa nilai input dan output dari program.
- Mengidentifikasi kelas kesetaraan.
- Rancang kasus uji kelas ekuivalensi dengan memilih satu nilai representatif dari setiap kelas ekuivalensi.
- Rancang kasus uji nilai batas sebagai berikut. Periksa apakah ada kelas ekivalensi yang merupakan rentang nilai. Sertakan nilai pada batas kelas kesetaraan tersebut dalam rangkaian pengujian.

Strategi untuk pengujian kotak hitam intuitif dan sederhana. Untuk pengujian kotak hitam, langkah yang paling penting adalah identifikasi kelas ekivalensi. Seringkali, identifikasi kelas kesetaraan tidak langsung. Namun, dengan sedikit latihan, seseorang akan dapat mengidentifikasi semua kelas ekivalensi dalam domain data input. Tanpa latihan, seseorang mungkin mengabaikan banyak kelas ekuivalensi dalam kumpulan data input. Setelah kelas ekivalensi diidentifikasi, kelas ekivalensi dan kasus uji nilai batas dapat dipilih hampir secara mekanis.

10.7 PENGUJIAN KOTAK PUTIH

Pengujian kotak putih adalah jenis pengujian unit yang penting. Sejumlah besar strategi pengujian kotak putih ada. Setiap strategi pengujian pada dasarnya merancang kasus uji berdasarkan analisis beberapa aspek kode sumber dan didasarkan pada beberapa heuristik. Pertama membahas beberapa konsep dasar yang terkait dengan pengujian kotak putih, dan menindaklanjutinya dengan diskusi tentang strategi pengujian khusus.

Konsep dasar

Strategi pengujian kotak putih dapat berbasis cakupan atau berbasis kesalahan.

Pengujian berbasis kesalahan

Strategi pengujian berbasis kesalahan menargetkan untuk mendeteksi jenis kesalahan tertentu. Kesalahan ini yang menjadi fokus strategi pengujian merupakan model kesalahan strategi. Contoh strategi berbasis kesalahan adalah pengujian mutasi, yang akan dibahas nanti di bagian ini.

Pengujian berbasis cakupan

Strategi pengujian berbasis cakupan mencoba untuk mengeksekusi (atau menutupi) elemen tertentu dari suatu program. Contoh populer dari strategi pengujian berbasis cakupan adalah cakupan pernyataan, cakupan cabang, cakupan kondisi ganda, dan pengujian berbasis cakupan jalur.

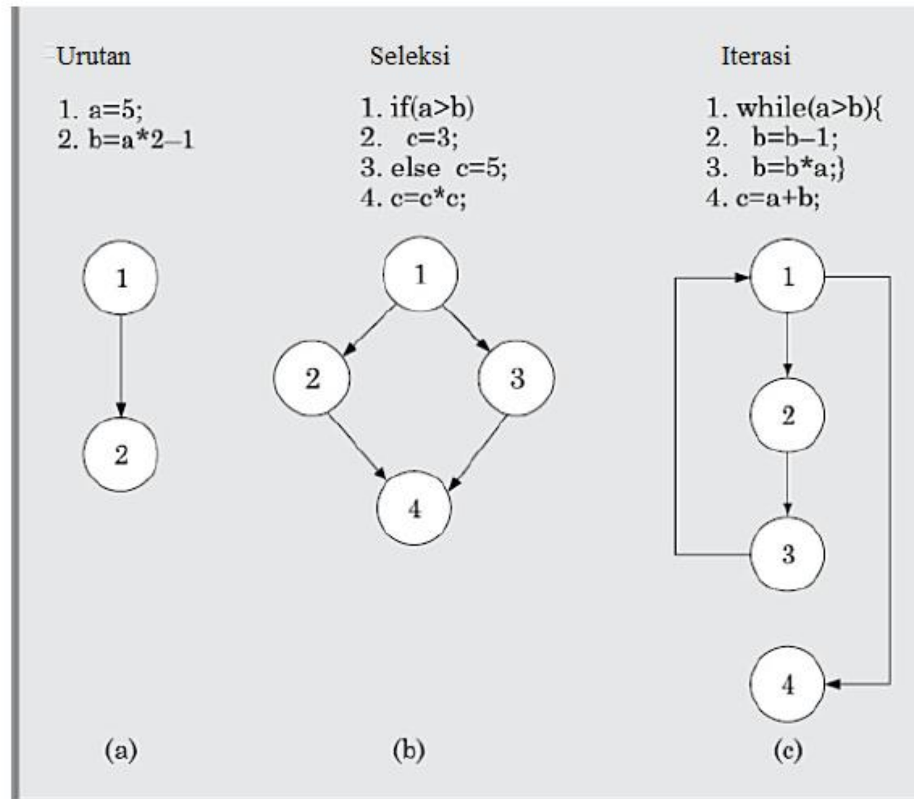
Kriteria pengujian untuk pengujian berbasis cakupan

Strategi pengujian berbasis cakupan biasanya menargetkan untuk mengeksekusi (yaitu, menutupi) elemen program tertentu untuk menemukan kegagalan. Kumpulan elemen program khusus yang ditargetkan untuk dieksekusi oleh strategi pengujian disebut kriteria pengujian strategi. Sebagai contoh, jika strategi pengujian membutuhkan semua pernyataan dari suatu program untuk dieksekusi setidaknya sekali, maka kita katakan bahwa kriteria pengujian dari strategi tersebut adalah cakupan pernyataan. Rangkaian tes memadai sehubungan dengan suatu kriteria, jika mencakup semua elemen domain yang ditentukan oleh kriteria itu.

Pengujian yang lebih kuat versus yang lebih lemah

Sejumlah besar strategi pengujian kotak putih telah diusulkan. Oleh karena itu menjadi perlu untuk membandingkan efektivitas strategi pengujian yang berbeda dalam mendeteksi kesalahan. Kita dapat membandingkan dua strategi pengujian dengan menentukan apakah yang satu lebih kuat, lebih lemah, atau saling melengkapi. Strategi pengujian kotak putih dikatakan lebih kuat daripada strategi lain, jika strategi pengujian yang lebih kuat mencakup semua elemen program yang tercakup oleh strategi pengujian yang lebih lemah, dan strategi yang lebih kuat juga mencakup setidaknya satu elemen program yang tidak tercakup oleh strategi yang lebih lemah. .

Ketika tidak satu pun dari dua strategi pengujian yang sepenuhnya mencakup elemen program yang dijalankan oleh yang lain, maka keduanya disebut strategi pengujian komplementer. Konsep pengujian yang lebih kuat, lebih lemah, dan komplementer secara skematis diilustrasikan pada Gambar 10.6. Perhatikan pada Gambar 10.6(a) bahwa strategi pengujian A lebih kuat dari B karena B hanya mencakup subset elemen yang tepat yang dicakup oleh B. Di sisi lain, Gambar 10.6(b) menunjukkan A dan B adalah strategi pengujian komplementer karena beberapa elemen dari A tidak tercakup oleh B dan sebaliknya. Jika pengujian yang lebih kuat telah dilakukan, maka pengujian yang lebih lemah tidak perlu diselesaikan.



Gambar 10.6 Ilustrasi strategi pengujian yang lebih kuat, lebih lemah, dan saling melengkapi.

Namun, rangkaian pengujian harus diperkaya dengan menggunakan berbagai strategi pengujian pelengkap. Pengujian berbasis cakupan sering digunakan untuk memeriksa kualitas pengujian yang dicapai oleh rangkaian pengujian perlu ditunjukkan. Sulit untuk secara manual merancang rangkaian pengujian untuk mencapai cakupan khusus untuk program non-sepele.

Cakupan Pernyataan

Strategi cakupan pernyataan bertujuan untuk merancang kasus uji sehingga dapat mengeksekusi setiap pernyataan dalam suatu program setidaknya satu kali. Gagasan utama yang mengatur strategi cakupan pernyataan adalah bahwa kecuali pernyataan dijalankan, tidak ada cara untuk menentukan apakah ada kesalahan dalam pernyataan itu.

Jelas bahwa tanpa mengeksekusi pernyataan, sulit untuk menentukan apakah itu menyebabkan kegagalan karena akses memori ilegal, perhitungan hasil yang salah karena operasi aritmatika yang tidak tepat, dll. Namun dapat ditunjukkan bahwa kelemahan dari cakupan pernyataan Strateginya adalah mengeksekusi pernyataan sekali dan mengamati bahwa pernyataan itu berperilaku dengan benar untuk satu nilai input tidak menjamin bahwa pernyataan itu akan berperilaku benar untuk semua nilai input. Namun demikian, cakupan pernyataan adalah teknik pengujian yang sangat intuitif dan menarik. Berikut ini adalah ilustrasi rangkaian pengujian yang mencapai cakupan pernyataan.

Contoh 10.11 Merancang rangkaian pengujian berbasis cakupan pernyataan untuk program komputasi GCD Euclid berikut:

```
int computeGCD(x,y)
    int x,y;
{
    1 while (x != y){
    2 if (x>y) then
    3 x=x-y;
```

```
        4 else y=y-x;
        5 }
        6 return x;
}
```

Jawaban: Untuk merancang kasus uji untuk cakupan pernyataan, ekspresi kondisional dari pernyataan while perlu dibuat benar dan ekspresi kondisional dari pernyataan if perlu dibuat benar dan salah. Dengan memilih set pengujian {(x = 3, y = 3), (x = 4, y = 3), (x = 3, y = 4)}, semua pernyataan program akan dieksekusi setidaknya sekali.

Cakupan Cabang

Test suite memenuhi cakupan cabang, jika membuat setiap kondisi cabang dalam program untuk mengasumsikan nilai benar dan salah secara bergantian. Dengan kata lain, untuk cakupan cabang, setiap cabang dalam representasi program CFG harus diambil setidaknya satu kali, ketika rangkaian uji dijalankan. Pengujian cabang juga dikenal sebagai pengujian tepi, karena dalam skema pengujian ini, setiap tepi grafik aliran kontrol program dilalui setidaknya sekali.

Contoh 10.12 Untuk program Contoh 10.11, tentukan test suite untuk mencapai cakupan cabang.

Jawaban: Rangkaian pengujian {(x = 3, y = 3), (x = 3, y = 2), (x = 4, y = 3), (x = 3, y = 4)} mencapai cakupan cabang. Sangat mudah untuk menunjukkan bahwa pengujian berbasis cakupan cabang adalah pengujian yang lebih kuat daripada pengujian berbasis cakupan pernyataan. Kita dapat membuktikan ini dengan menunjukkan bahwa cakupan cabang memastikan cakupan pernyataan, tetapi tidak sebaliknya.

Teorema 10.1 Pengujian berbasis cakupan cabang lebih kuat dari pengujian berbasis cakupan pernyataan.

Bukti: Kita perlu menunjukkan bahwa (a) cakupan cabang memastikan cakupan pernyataan, dan (b) cakupan pernyataan tidak memastikan cakupan cabang.

a. Pengujian cabang akan menjamin cakupan pernyataan karena setiap pernyataan harus dimiliki oleh beberapa cabang (dengan asumsi bahwa tidak ada kode yang tidak dapat dijangkau).
b. Untuk menunjukkan bahwa cakupan pernyataan tidak menjamin cakupan cabang, akan cukup untuk memberikan contoh rangkaian uji yang mencapai cakupan pernyataan, tetapi tidak mencakup setidaknya satu cabang. Pertimbangkan kode berikut, dan rangkaian pengujian {5}. jika(x>2) x+1;

Test suite akan mencapai cakupan pernyataan. Namun, itu tidak mencapai cakupan cabang, karena kondisi (x > 2) tidak dibuat salah oleh kasus uji apa pun di suite.

Cakupan Berbagai Kondisi

Dalam pengujian berbasis cakupan kondisi ganda (MC), kasus uji dirancang untuk membuat setiap komponen ekspresi kondisional komposit mengasumsikan nilai benar dan salah. Misalnya, pertimbangkan ekspresi kondisional gabungan ((c1 .and.c2).or.c3). Rangkaian uji akan mencapai cakupan MC, jika semua kondisi komponen c1, c2 dan c3 masing-masing dibuat untuk mengasumsikan nilai benar dan salah. Pengujian cabang dapat dianggap sebagai strategi pengujian kondisi yang sederhana di mana hanya kondisi gabungan yang muncul dalam pernyataan cabang yang berbeda yang dibuat untuk mengasumsikan nilai benar dan salah. Sangat mudah untuk membuktikan bahwa pengujian kondisi adalah strategi pengujian yang lebih kuat daripada pengujian cabang. Untuk ekspresi kondisional komposit dari n komponen, 2n kasus uji diperlukan untuk cakupan kondisi ganda. Jadi, untuk cakupan kondisi ganda, jumlah kasus uji meningkat secara eksponensial dengan jumlah kondisi

komponen. Oleh karena itu, teknik pengujian berbasis cakupan kondisi ganda hanya praktis jika n (jumlah kondisi) kecil.

Contoh 10.13 Berikan contoh kesalahan yang dideteksi oleh cakupan kondisi ganda, tetapi tidak oleh cakupan cabang.

Jawaban: Perhatikan segmen program C berikut:

```
if(temperature>150 || temperature>50)
setWarningLightOn()
```

Segmen program memiliki bug dalam kondisi komponen kedua, seharusnya suhu <50. Rangkaian uji {temperature=160, temperature=40} mencapai cakupan cabang. Tapi, itu tidak dapat memeriksa setWarningLightOn(); tidak boleh disebut untuk nilai suhu dalam 150 dan 50.

Cakupan Jalur

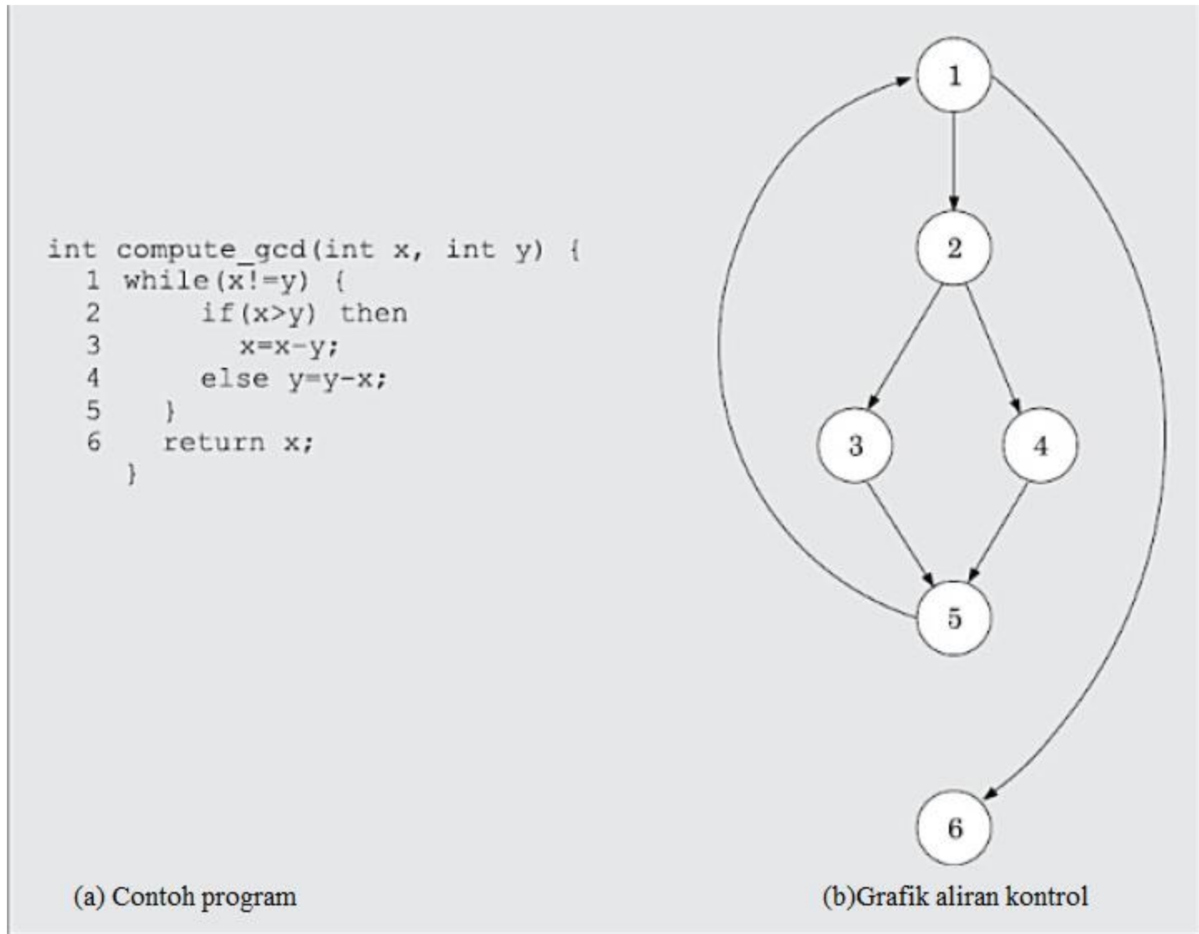
Test suite mencapai cakupan jalur jika mengeksekusi setiap jalur independen linier (atau jalur basis) setidaknya sekali. Jalur bebas linier dapat didefinisikan dalam bentuk grafik aliran kontrol (CFG) dari suatu program. Oleh karena itu, untuk memahami strategi pengujian berbasis cakupan jalur, pertama-tama kita perlu memahami bagaimana CFG suatu program dapat ditarik.

Grafik aliran kontrol (CFG)

Grafik aliran kontrol menggambarkan bagaimana kontrol mengalir melalui program. Kita dapat mendefinisikan grafik aliran kontrol sebagai berikut: *Grafik aliran kontrol menggambarkan urutan di mana instruksi yang berbeda dari suatu program dieksekusi.* Untuk menggambar grafik aliran kontrol suatu program, pertama-tama kita perlu memberi nomor pada semua pernyataan dari suatu program. Pernyataan bernomor yang berbeda berfungsi sebagai simpul dari grafik aliran kontrol (lihat Gambar 10.5). Terdapat edge dari satu node ke node lainnya, jika eksekusi pernyataan yang mewakili node pertama dapat mengakibatkan transfer kontrol ke node lainnya.

Secara lebih formal, kita dapat mendefinisikan CFG sebagai berikut. CFG adalah graf berarah yang terdiri dari sekumpulan node dan edge (N, E), sehingga setiap node n x N sesuai dengan pernyataan program yang unik dan edge ada di antara dua node jika kontrol dapat mentransfer dari satu node ke node lainnya.

Kita dapat dengan mudah menggambar CFG untuk program apa pun, jika kita tahu cara merepresentasikan urutan, seleksi, dan jenis pernyataan dalam CFG. Bagaimanapun, setiap program dibangun dengan menggunakan tiga jenis konstruksi ini saja. Gambar 10.5 merangkum bagaimana CFG untuk ketiga jenis konstruksi ini dapat ditarik. Representasi CFG dari urutan dan jenis keputusan dari pernyataan adalah lurus ke depan. Harap perhatikan dengan seksama bagaimana CFG untuk konstruksi loop (iterasi) dapat digambar. Untuk konstruksi tipe iterasi seperti konstruksi while, kondisi loop diuji hanya pada awal loop dan oleh karena itu selalu mengontrol aliran dari pernyataan terakhir loop ke atas loop. Artinya, konstruk loop berakhir dari pernyataan pertama (setelah loop ditemukan salah) dan tidak setiap saat keluar dari loop pada pernyataan terakhir dari loop. Dengan menggunakan ide-ide dasar ini, CFG dari program yang diberikan pada Gambar 10.7(a) dapat digambarkan seperti yang ditunjukkan pada Gambar 10.7(b).



(a) Contoh program (b)Grafik aliran kontrol

Gambar 10.7 Diagram alir kontrol dari contoh program.

Jalur

Jalur melalui program adalah setiap simpul dan urutan tepi dari simpul awal ke simpul terminal dari grafik aliran kontrol suatu program. Harap dicatat bahwa sebuah program dapat memiliki lebih dari satu simpul terminal ketika berisi beberapa jenis pernyataan keluar atau kembali. Menulis kasus uji untuk mencakup semua jalur program tipikal tidak praktis karena mungkin ada jumlah jalur tak terbatas melalui program dengan adanya loop. Sebagai contoh, pada Gambar 10.5(c), mungkin terdapat jumlah jalur yang tidak terbatas seperti 12314, 12312314, 12312312314, dll. Jika cakupan semua jalur dicoba, maka jumlah kasus uji yang diperlukan akan menjadi sangat besar. Untuk alasan ini, pengujian cakupan jalur tidak mencoba untuk mencakup semua jalur, tetapi hanya sebagian dari jalur yang disebut jalur independen linier (atau jalur basis). Sekarang mari kita bahas apa itu jalur bebas linier dan bagaimana menentukannya dalam sebuah program.

Himpunan jalur independen linier (atau set jalur basis)

Sekumpulan jalur untuk program tertentu disebut himpunan jalur bebas linier (atau himpunan jalur basis atau hanya himpunan basis), jika setiap jalur dalam himpunan tersebut memperkenalkan setidaknya satu sisi baru yang tidak termasuk dalam jalur lain dalam set. Harap dicatat bahwa meskipun kita menemukan bahwa suatu jalur memiliki satu simpul baru dibandingkan dengan semua jalur bebas linier lainnya, maka jalur ini juga harus dimasukkan dalam himpunan jalur bebas linier. Ini karena, setiap jalur yang memiliki simpul baru akan secara otomatis memiliki tepi baru. Definisi alternatif dari himpunan jalur bebas linier [McCabe76] adalah sebagai berikut: Jika himpunan jalur bebas linier satu sama lain, maka

tidak ada jalur dalam himpunan yang dapat diperoleh melalui operasi linier apa pun (yaitu, penambahan atau pengurangan) pada jalur lain dalam himpunan.

Menurut definisi di atas dari himpunan jalur bebas linier, untuk setiap jalur dalam himpunan, subjalurnya tidak dapat menjadi anggota himpunan. Faktanya, sembarang jalur program, dapat disintesis dengan melakukan operasi linier pada jalur basis. Mungkin, nama basis set berasal dari pengamatan bahwa jalur di basis set membentuk "basis" untuk semua jalur program. Harap dicatat bahwa mungkin tidak selalu ada basis set unik untuk suatu program dan beberapa set basis untuk program yang sama biasanya dapat ditentukan.

Meskipun lurus ke depan untuk mengidentifikasi jalur independen linier untuk program sederhana, untuk program yang lebih kompleks tidak mudah untuk menentukan jumlah jalur independen. Dalam konteks ini, metrik kompleksitas siklomatik McCabe adalah hasil penting yang memungkinkan kita menghitung jumlah jalur independen linier untuk program arbitrer apa pun. Kompleksitas siklomatik McCabe mendefinisikan batas atas untuk jumlah jalur bebas linier melalui suatu program. Juga, kompleksitas siklomatik McCabe sangat sederhana untuk dihitung. Meskipun metrik McCabe tidak secara langsung mengidentifikasi jalur independen linier, tetapi metrik ini memberi kita cara praktis untuk menentukan kira-kira berapa banyak jalur yang harus dicari.

Metrik Kompleksitas Cyclomatic McCabe

McCabe memperoleh hasil-hasilnya dengan menerapkan teknik-teknik teori-grafik pada grafik aliran kendali suatu program. Kompleksitas siklomatik McCabe mendefinisikan batas atas pada jumlah jalur independen dalam suatu program. Ada tiga cara berbeda untuk menghitung kompleksitas siklomatik. Untuk program terstruktur, hasil yang dihitung dengan ketiga metode tersebut dijamin sesuai.

Metode 1: Diberikan grafik aliran kontrol G dari suatu program, kompleksitas siklomatik V(G) dapat dihitung sebagai:

$$V(G) = E - N + 2$$

di mana, N adalah jumlah node dari grafik aliran kontrol dan E adalah jumlah tepi dalam grafik aliran kontrol. Untuk CFG contoh pada Gambar 10.7, E = 7 dan N = 6. Oleh karena itu, nilai kompleksitas Cyclomatic = 7 – 6 + 2 = 3.

Metode 2: Cara alternatif untuk menghitung kompleksitas siklomatik suatu program didasarkan pada inspeksi visual dari grafik aliran kontrol adalah sebagai berikut —Dalam metode ini, kompleksitas siklomatik V (G) untuk grafik G diberikan oleh ekspresi berikut :

V(G) = Jumlah total area berbatas yang tidak tumpang tindih + 1

Dalam grafik aliran kendali program G, setiap daerah yang dibatasi oleh node dan edge dapat disebut sebagai daerah berbatas. Ini adalah cara mudah untuk menentukan kompleksitas siklomatik McCabe. Tapi, bagaimana jika graf G tidak planar (yaitu, bagaimanapun Anda menggambar grafik, dua atau lebih sisi selalu berpotongan). Sebenarnya, dapat ditunjukkan bahwa representasi aliran kontrol dari program terstruktur selalu menghasilkan grafik planar. Namun, kehadiran GOTO dapat dengan mudah menambahkan tepi yang berpotongan. Oleh karena itu, untuk program tidak terstruktur, cara menghitung kompleksitas siklomatik McCabe ini tidak berlaku. Jumlah daerah yang dibatasi dalam CFG meningkat dengan jumlah pernyataan keputusan dan loop. Oleh karena itu, metrik McCabe memberikan ukuran kuantitatif dari kesulitan pengujian dan keandalan akhir dari suatu program. Perhatikan contoh CFG yang ditunjukkan pada Gambar 10.7. Dari pemeriksaan visual CFG jumlah daerah yang dibatasi adalah 2. Oleh karena itu kompleksitas siklomatik, dihitung dengan metode ini juga 2+1=3. Metode ini memberikan cara yang sangat mudah untuk menghitung kompleksitas siklomatik CFG, hanya dari pemeriksaan visual CFG. Di sisi lain, metode untuk menghitung CFG dapat dengan mudah diotomatisasi. Artinya, metode

perhitungan metrik McCabe 1 dan 3 dapat dengan mudah dikodekan ke dalam program yang dapat digunakan untuk secara otomatis menentukan kompleksitas siklomatik program arbitrer.

Metode 3: Kompleksitas siklomatik suatu program juga dapat dengan mudah dihitung dengan menghitung jumlah pernyataan keputusan dan loop dari program. Jika N adalah jumlah pernyataan keputusan dan loop dari suatu program, maka metrik McCabe sama dengan N + 1.

Bagaimana pengujian jalur dilakukan dengan menggunakan komputer?

Nilai metrik siklomatik McCabe? Mengetahui jumlah jalur dasar dalam suatu program tidak memudahkan untuk merancang kasus uji untuk cakupan jalur, hanya memberikan indikasi jumlah minimum kasus uji yang diperlukan untuk cakupan jalur. Untuk CFG dari segmen program yang cukup kompleks misalnya 20 node dan 25 edge, Anda mungkin memerlukan beberapa hari upaya untuk mengidentifikasi semua jalur independen linier di dalamnya dan untuk merancang kasus uji. Oleh karena itu tidak praktis untuk meminta perancang uji untuk mengidentifikasi semua jalur independen linier dalam kode, dan kemudian merancang kasus uji untuk memaksa eksekusi di sepanjang masing-masing jalur yang diidentifikasi. Dalam praktiknya, untuk pengujian jalur, biasanya penguji terus membentuk kasus uji dengan data acak dan mengeksekusinya hingga cakupan yang diperlukan tercapai. Alat pengujian seperti penganalisis program dinamis digunakan untuk menentukan persentase jalur bebas linier yang dicakup oleh kasus uji yang telah dijalankan sejauh ini. Jika persentase jalur independen linier yang tercakup di bawah 90 persen, lebih banyak kasus uji (dengan input acak) ditambahkan untuk meningkatkan cakupan jalur. Biasanya, tidak praktis untuk menargetkan pencapaian 100 persen cakupan jalur, karena metrik McCabe hanya batas atas dan tidak memberikan jumlah jalur yang tepat.

Langkah-langkah untuk melakukan pengujian berbasis cakupan jalur

Berikut adalah urutan langkah-langkah yang perlu dilakukan untuk menurunkan kasus uji berbasis cakupan jalur untuk suatu program:

1. Gambarkan grafik aliran kendali untuk program tersebut.
2. Tentukan metrik McCabe V(G).
3. Tentukan kompleksitas siklomatik. Ini memberikan jumlah minimum kasus uji yang diperlukan untuk mencapai cakupan jalur.
4. ulangi

Uji menggunakan serangkaian kasus uji yang dirancang secara acak. Lakukan analisis dinamis untuk memeriksa cakupan jalur yang dicapai hingga setidaknya 90 persen cakupan jalur tercapai.

Penggunaan metrik kompleksitas siklomatik McCabe

Selain penggunaannya dalam pengujian jalur, kompleksitas siklomatik program memiliki banyak aplikasi menarik lainnya seperti berikut ini:

Estimasi kompleksitas struktural kode: Kompleksitas siklomatik McCabe adalah ukuran kompleksitas struktural suatu program. Alasan untuk ini adalah bahwa hal itu dihitung berdasarkan struktur kode (jumlah keputusan dan konstruksi iterasi yang digunakan). Secara intuitif, metrik kompleksitas McCabe berkorelasi dengan tingkat kesulitan memahami program, karena seseorang memahami program dengan memahami perhitungan yang dilakukan di sepanjang jalur independen program. Kompleksitas siklomatik suatu program adalah ukuran kompleksitas psikologis atau tingkat kesulitan dalam memahami program.

Mengingat hasil di atas, dari perspektif pemeliharaan, masuk akal untuk membatasi kompleksitas siklomatik dari fungsi yang berbeda hingga beberapa nilai yang masuk akal. Organisasi pengembangan perangkat lunak yang baik biasanya membatasi kompleksitas

siklomatik dari berbagai fungsi hingga nilai maksimum sepuluh atau lebih. Ini berbeda dengan kompleksitas komputasi yang didasarkan pada eksekusi pernyataan program.

Estimasi upaya pengujian: Kompleksitas siklomatik adalah ukuran jumlah maksimum jalur basis. Dengan demikian, ini menunjukkan jumlah minimum kasus uji yang diperlukan untuk mencapai cakupan jalur. Oleh karena itu, upaya pengujian dan waktu yang diperlukan untuk menguji sepotong kode secara memuaskan sebanding dengan kompleksitas siklomatik kode. Untuk mengurangi upaya pengujian, perlu untuk membatasi kompleksitas siklomatik setiap fungsi menjadi tujuh.

Estimasi keandalan program: Studi eksperimental menunjukkan adanya hubungan yang jelas antara metrik McCabe dan jumlah kesalahan laten dalam kode setelah pengujian. Hubungan ini ada mungkin karena korelasi kompleksitas siklomatik dengan kompleksitas struktural kode. Biasanya semakin besar kompleksitas struktural, semakin sulit untuk menguji dan men-debug kode.

Pengujian Berbasis Aliran Data

Metode pengujian berbasis aliran data memilih jalur pengujian suatu program sesuai dengan definisi dan penggunaan variabel yang berbeda dalam suatu program.

Pertimbangkan program P. Untuk pernyataan bernomor S dari P , misalkan

DEF(S) = {X /pernyataan S berisi definisi X } dan

USES(S)= {X /pernyataan S berisi penggunaan X }

Untuk pernyataan S: a=b+c;, DEF(S)={a}, USES(S)={b, c}. Definisi variabel X pada pernyataan S dikatakan hidup pada pernyataan S1 , jika terdapat lintasan dari pernyataan S ke pernyataan S1 yang tidak mengandung definisi X .

Semua kriteria definisi adalah kriteria cakupan pengujian yang mensyaratkan bahwa set pengujian yang memadai harus mencakup semua kemunculan definisi dalam arti bahwa, untuk setiap kemunculan definisi, jalur pengujian harus mencakup jalur di mana definisi mencapai penggunaan definisi. Semua kriteria penggunaan mensyaratkan bahwa semua penggunaan definisi harus tercakup. Jelas, kriteria semua kegunaan lebih kuat daripada kriteria semua definisi. Kriteria yang lebih kuat lagi adalah semua kriteria jalur penggunaan definisi, yang memerlukan cakupan semua jalur penggunaan definisi yang mungkin, baik yang bebas siklus atau hanya memiliki siklus sederhana. Siklus sederhana adalah lintasan di mana hanya simpul akhir dan simpul awal yang sama.

Rantai definisi-penggunaan (atau rantai DU) dari variabel X berbentuk [X, S, S1], di mana S dan S1 adalah nomor pernyataan, sehingga X x DEF(S) dan X x USES(S1), dan definisi X pada pernyataan S adalah langsung pada pernyataan S1. Salah satu strategi pengujian aliran data sederhana adalah mengharuskan setiap rantai DU dicakup setidaknya sekali. Strategi pengujian aliran data sangat berguna untuk menguji program yang berisi pernyataan if dan loop bersarang.

Pengujian Mutasi

Semua strategi pengujian kotak putih yang telah kita bahas sejauh ini, adalah teknik pengujian berbasis cakupan. Sebaliknya, pengujian mutasi adalah teknik pengujian berbasis kesalahan dalam arti bahwa kasus uji mutasi dirancang untuk membantu mendeteksi jenis kesalahan tertentu dalam suatu program. Dalam pengujian mutasi, sebuah program pertama kali diuji dengan menggunakan rangkaian pengujian awal yang dirancang dengan menggunakan berbagai strategi pengujian kotak putih yang telah kita bahas. Setelah pengujian awal selesai, pengujian mutasi dapat dilakukan. Gagasan di balik pengujian mutasi adalah membuat beberapa perubahan sewenang-wenang pada suatu program pada suatu waktu. Setiap kali program diubah, itu disebut program bermutasi dan perubahan yang dilakukan disebut mutan. Asumsi yang mendasari pengujian mutasi adalah bahwa semua

kesalahan pemrograman dapat dinyatakan sebagai kombinasi kesalahan sederhana. Operator mutasi membuat perubahan khusus pada program. Misalnya, satu operator mutasi dapat secara acak menghapus pernyataan program. Sebuah mutan mungkin atau mungkin tidak menyebabkan kesalahan dalam program. Jika mutan tidak menimbulkan kesalahan dalam program, maka program asli dan program yang dimutasi disebut program ekuivalen.

Program yang bermutasi diuji terhadap rangkaian uji asli program. Jika terdapat setidaknya satu test case dalam test suite dimana program bermutasi menghasilkan hasil yang salah, maka mutan dikatakan mati, karena kesalahan yang diperkenalkan oleh operator mutasi telah berhasil dideteksi oleh test suite. Jika mutan tetap hidup bahkan setelah semua kasus uji telah habis, rangkaian uji ditingkatkan untuk membunuh mutan. Namun, tidak semudah ini. Ingatlah bahwa ada kemungkinan program yang bermutasi menjadi program yang setara. Jika ini masalahnya, akan sia-sia mencoba merancang kasus uji yang akan mengidentifikasi kesalahan.

Sebuah keuntungan penting dari pengujian mutasi adalah bahwa hal itu dapat diotomatisasi untuk sebagian besar. Proses generasi mutan dapat diotomatisasi dengan mendefinisikan satu set perubahan primitif yang dapat diterapkan pada program. Perubahan primitif ini dapat berupa perubahan program sederhana seperti—menghapus pernyataan, menghapus definisi variabel, mengubah jenis operator aritmatika (mis., + ke -), mengubah operator logika (dan ke atau) mengubah nilai konstanta , mengubah tipe data variabel, dll. Kelemahan utama dari pendekatan pengujian berbasis mutasi adalah biaya komputasinya sangat mahal, karena sejumlah besar kemungkinan mutan dapat dihasilkan.

Pengujian mutasi melibatkan menghasilkan sejumlah besar mutan. Juga setiap mutan perlu diuji dengan rangkaian uji lengkap. Jelas karena itu, pengujian mutasi tidak cocok untuk pengujian manual. Pengujian mutasi paling cocok untuk digunakan bersama dengan beberapa alat pengujian yang secara otomatis menghasilkan mutan dan menjalankan rangkaian pengujian secara otomatis pada setiap mutan. Saat ini, beberapa alat uji tersedia yang secara otomatis menghasilkan mutan untuk program tertentu.

10.8 DEBUGGING

Setelah kegagalan terdeteksi, pertama-tama perlu mengidentifikasi pernyataan program yang salah dan bertanggung jawab atas kegagalan tersebut, kemudian kesalahan tersebut dapat diperbaiki. Dalam Bagian ini, akan merangkum pendekatan penting yang tersedia untuk mengidentifikasi lokasi kesalahan. Masing-masing pendekatan ini memiliki kelebihan dan kekurangannya sendiri dan oleh karena itu masing-masing akan berguna dalam keadaan yang sesuai.

Pendekatan Debug

Berikut ini adalah beberapa pendekatan yang populer diadopsi oleh programmer untuk debugging:

Metode kekerasan

Ini adalah metode debugging yang paling umum tetapi merupakan metode yang paling tidak efisien. Dalam pendekatan ini, pernyataan cetak disisipkan di seluruh program untuk mencetak nilai antara dengan harapan bahwa beberapa nilai yang dicetak akan membantu mengidentifikasi pernyataan yang salah. Pendekatan ini menjadi lebih sistematis dengan penggunaan debugger simbolis (juga disebut debugger kode sumber), karena nilai variabel yang berbeda dapat dengan mudah diperiksa dan break point serta watch point dapat dengan mudah diatur untuk menguji nilai variabel dengan mudah. Melangkah tunggal menggunakan debugger simbolis adalah bentuk lain dari pendekatan ini, di mana developer secara mental

menghitung hasil yang diharapkan setelah setiap instruksi sumber dan memeriksa apakah hal yang sama dihitung dengan langkah tunggal melalui program.

Mundur

Ini juga merupakan pendekatan yang cukup umum. Dalam pendekatan ini, mulai dari pernyataan di mana gejala kesalahan telah diamati, kode sumber ditelusuri mundur sampai kesalahan ditemukan. Sayangnya, karena jumlah jalur sumber yang akan dilacak kembali meningkat, jumlah jalur mundur potensial meningkat dan mungkin menjadi sangat besar untuk program yang kompleks, membatasi penggunaan pendekatan ini.

Metode eliminasi penyebab

Dalam pendekatan ini, setelah kegagalan diamati, gejala kegagalan (yaitu, variabel tertentu memiliki nilai negatif meskipun seharusnya positif, dll.) dicatat. Berdasarkan gejala kegagalan, penyebab yang mungkin berkontribusi terhadap gejala dikembangkan dan tes dilakukan untuk menghilangkan masing-masing. Teknik terkait identifikasi kesalahan dari gejala kesalahan adalah analisis pohon kesalahan perangkat lunak.

Pengirisan program

Teknik ini mirip dengan back tracking. Dalam pendekatan backtracking, seseorang sering kali harus memeriksa sejumlah besar pernyataan. Namun, ruang pencarian dikurangi dengan mendefinisikan irisan. Sepotong program untuk variabel tertentu dan pada pernyataan tertentu adalah kumpulan baris sumber sebelum pernyataan ini yang dapat mempengaruhi nilai variabel itu [Mund2002]. Pengirisan program memanfaatkan fakta bahwa kesalahan dalam nilai variabel dapat disebabkan oleh pernyataan yang bergantung pada data.

Pedoman Debug

Debugging sering dilakukan oleh programmer berdasarkan kecerdikan dan pengalaman mereka. Berikut ini adalah beberapa panduan umum untuk debugging yang efektif:

- Banyak kali debugging membutuhkan pemahaman menyeluruh tentang desain program. Mencoba men-debug berdasarkan pemahaman parsial tentang desain program mungkin memerlukan banyak upaya untuk melakukan debug bahkan untuk masalah sederhana.
- Debugging kadang-kadang bahkan memerlukan desain ulang penuh dari sistem. Dalam kasus seperti itu, kesalahan umum yang sering dilakukan oleh programmer pemula adalah mencoba untuk tidak memperbaiki kesalahan tetapi gejalanya.
- Seseorang harus waspada terhadap kemungkinan bahwa koreksi kesalahan dapat menimbulkan kesalahan baru. Oleh karena itu setelah setiap putaran perbaikan kesalahan, pengujian regresi harus dilakukan.

10.9 ALAT ANALISIS PROGRAM

Alat analisis program biasanya adalah alat otomatis yang mengambil kode sumber atau kode yang dapat dieksekusi dari suatu program sebagai input dan menghasilkan laporan mengenai beberapa karakteristik penting dari program, seperti ukurannya, kompleksitas, kecukupan komentar, kepatuhan terhadap standar pemrograman. , kecukupan pengujian, dll. Klasifikasi berbagai alat analisis program ke dalam dua kategori besar adalah sebagai berikut:

- Alat analisis statis
- Alat analisis dinamis

Alat Analisis Statis

Alat analisis program statis menilai dan menghitung berbagai karakteristik suatu program tanpa menjalankannya. Biasanya, alat analisis statis menganalisis kode sumber untuk menghitung metrik tertentu yang mencirikan kode sumber (seperti ukuran, kompleksitas

siklomatik, dll.) dan juga melaporkan kesimpulan analitis tertentu. Ini juga memeriksa kesesuaian kode dengan standar pengkodean yang ditentukan. Dalam konteks ini, ini menampilkan hasil analisis berikut:

- Sejauh mana standar pengkodean telah dipatuhi?
- Apakah kesalahan pemrograman tertentu seperti variabel yang tidak diinisialisasi, ketidakcocokan antara parameter aktual dan formal, variabel yang dideklarasikan tetapi tidak pernah digunakan, dll., ada? Daftar semua kesalahan tersebut ditampilkan.

Teknik peninjauan kode seperti penelusuran kode dan inspeksi kode dapat dianggap sebagai metode analisis statis karena target tersebut untuk mendeteksi kesalahan berdasarkan analisis kode sumber. Namun, sebenarnya, ini tidak benar karena menggunakan istilah analisis program statis untuk menunjukkan alat analisis otomatis. Di sisi lain, kompiler dapat dianggap sebagai jenis alat analisis program statis.

Keterbatasan praktis utama dari alat analisis statis terletak pada ketidakmampuannya untuk menganalisis informasi run-time seperti referensi memori dinamis menggunakan variabel pointer dan aritmatika pointer, dll. Dalam bahasa pemrograman tingkat tinggi, variabel pointer dan alokasi memori dinamis menyediakan kemampuan untuk referensi memori dinamis. Namun, referensi memori dinamis adalah sumber utama kesalahan pemrograman dalam suatu program. Alat analisis statis sering merangkum hasil analisis setiap fungsi dalam grafik kutub yang dikenal sebagai Kiviat Chart. Bagan Kiviat biasanya menunjukkan nilai yang dianalisis untuk kompleksitas siklomatik, jumlah baris sumber, persentase baris komentar, metrik Halstead, dll.

Alat Analisis Dinamis

Alat analisis program dinamis dapat digunakan untuk mengevaluasi beberapa karakteristik program berdasarkan analisis perilaku waktu berjalan suatu program. Alat-alat ini biasanya merekam dan menganalisis perilaku sebenarnya dari suatu program saat sedang dieksekusi. Alat analisis program dinamis (juga disebut penganalisis dinamis) biasanya mengumpulkan informasi jejak eksekusi dengan memasukkan kode. Instrumentasi kode biasanya dicapai dengan menyisipkan pernyataan tambahan untuk mencetak nilai variabel tertentu ke dalam file untuk mengumpulkan jejak eksekusi program. Kode yang diinstrumentasi saat dieksekusi, mencatat perilaku perangkat lunak untuk kasus uji yang berbeda. Karakteristik penting dari rangkaian uji yang dihitung oleh alat analisis dinamis adalah tingkat cakupan yang dicapai oleh rangkaian uji.

Setelah perangkat lunak diuji dengan rangkaian pengujian lengkap dan perilakunya dicatat, alat analisis dinamis melakukan analisis pasca eksekusi dan menghasilkan laporan yang menjelaskan cakupan yang telah dicapai oleh rangkaian pengujian lengkap untuk program tersebut. Misalnya, alat analisis dinamis dapat melaporkan pernyataan, cabang, dan cakupan jalur yang dicapai oleh rangkaian pengujian. Jika cakupan yang dicapai tidak memuaskan, lebih banyak kasus uji dapat dirancang, ditambahkan ke rangkaian uji, dan dijalankan. Selanjutnya, hasil analisis dinamis dapat membantu menghilangkan kasus uji yang berlebihan dari rangkaian uji. Biasanya hasil analisis dinamis dilaporkan dalam bentuk histogram atau diagram lingkaran untuk menggambarkan cakupan struktural yang dicapai untuk berbagai modul program. Output dari alat analisis dinamis dapat disimpan dan dicetak dengan mudah untuk memberikan bukti bahwa pengujian menyeluruh telah dilakukan.

10.10 TES INTEGRASI

Pengujian integrasi dilakukan setelah semua (atau setidaknya beberapa) modul telah diuji unit. Penyelesaian pengujian unit yang berhasil, sebagian besar, memastikan bahwa unit

(atau modul) secara keseluruhan berfungsi dengan baik. Dalam konteks ini, tujuan pengujian integrasi adalah untuk mendeteksi kesalahan pada antarmuka modul (parameter panggilan). Misalnya, diperiksa bahwa tidak ada ketidakcocokan parameter yang terjadi ketika satu modul memanggil fungsionalitas modul lain. Jadi, tujuan utama dari pengujian integrasi adalah untuk menguji antarmuka modul, yaitu, tidak ada kesalahan dalam melewati parameter, ketika satu modul memanggil fungsionalitas modul lain. Tujuan dari pengujian integrasi adalah untuk memeriksa apakah modul-modul yang berbeda dari sebuah program berinteraksi satu sama lain dengan benar.

Selama pengujian integrasi, modul yang berbeda dari suatu sistem diintegrasikan secara terencana menggunakan rencana integrasi. Rencana integrasi menentukan langkah-langkah dan urutan modul digabungkan untuk mewujudkan sistem penuh. Setelah setiap langkah integrasi, sistem yang terintegrasi sebagian diuji. Faktor penting yang memandu rencana integrasi adalah grafik ketergantungan modul.

Kita telah membahas di Bab 6 bahwa bagan struktur (atau grafik ketergantungan modul) menentukan urutan di mana modul yang berbeda saling memanggil. Dengan demikian, dengan memeriksa bagan struktur, rencana integrasi dapat dikembangkan. Salah satu (atau campuran) dari pendekatan berikut dapat digunakan untuk mengembangkan rencana pengujian:

- Pendekatan big-bang untuk pengujian integrasi
- Pendekatan top-down untuk pengujian integrasi
- Pendekatan bottom-up untuk pengujian integrasi
- Pendekatan campuran (juga disebut terjepit) untuk pengujian integrasi

Pendekatan big-bang untuk pengujian integrasi

Pengujian big-bang adalah pendekatan yang paling jelas untuk pengujian integrasi. Dalam pendekatan ini, semua modul yang membentuk sistem terintegrasi dalam satu langkah. Dengan kata sederhana, semua modul sistem yang diuji unit hanya dihubungkan bersama dan diuji. Namun, teknik ini hanya dapat digunakan untuk sistem yang sangat kecil. Masalah utama dengan pendekatan ini adalah bahwa setelah kegagalan terdeteksi selama pengujian integrasi, sangat sulit untuk melokalisasi kesalahan karena kesalahan mungkin terletak di salah satu modul. Oleh karena itu, kesalahan debugging yang dilaporkan selama pengujian integrasi big-bang sangat mahal untuk diperbaiki. Akibatnya, pengujian integrasi big-bang hampir tidak pernah digunakan untuk program besar.

Pendekatan bottom-up untuk pengujian integrasi

Produk perangkat lunak yang besar seringkali terdiri dari beberapa subsistem. Sebuah subsistem mungkin terdiri dari banyak modul yang berkomunikasi satu sama lain melalui antarmuka yang terdefinisi dengan baik. Dalam pengujian integrasi bottom-up, pertama modul untuk setiap subsistem diintegrasikan. Dengan demikian, subsistem dapat diintegrasikan secara terpisah dan mandiri.

Tujuan utama melakukan pengujian integrasi subsistem adalah untuk menguji apakah antarmuka di antara berbagai modul yang membentuk subsistem bekerja dengan memuaskan. Kasus uji harus dipilih dengan hati-hati untuk menggunakan antarmuka dalam semua cara yang mungkin. Dalam pengujian bottom-up murni tidak diperlukan stub, dan hanya test-driver yang diperlukan. Sistem perangkat lunak yang besar biasanya memerlukan beberapa tingkat pengujian subsistem, subsistem tingkat yang lebih rendah secara berurutan digabungkan untuk membentuk subsistem tingkat yang lebih tinggi. Keuntungan utama dari pengujian integrasi bottom-up adalah bahwa beberapa subsistem yang terpisah dapat diuji secara bersamaan. Keuntungan lain dari pengujian bottom-up adalah bahwa modul tingkat rendah diuji secara menyeluruh, karena dilakukan di setiap langkah integrasi. Karena modul

tingkat rendah melakukan I/O dan fungsi penting lainnya, pengujian modul tingkat rendah secara menyeluruh meningkatkan keandalan sistem. Kerugian dari pengujian bottom-up adalah kompleksitas yang terjadi ketika sistem terdiri dari sejumlah besar subsistem kecil yang berada pada level yang sama. Kasus ekstrim ini sesuai dengan pendekatan big-bang.

Pendekatan top-down untuk pengujian integrasi

Pengujian integrasi top-down dimulai dengan modul root dalam bagan struktur dan satu atau dua modul bawahan dari modul root. Setelah 'kerangka' tingkat atas diuji, modul yang langsung berada di lapisan 'kerangka' yang lebih rendah digabungkan dengannya dan diuji. Pendekatan pengujian integrasi top-down membutuhkan penggunaan program stub untuk mensimulasikan efek dari rutinitas tingkat yang lebih rendah yang dipanggil oleh rutinitas yang sedang diuji. Integrasi top-down murni tidak memerlukan rutinitas driver apa pun. Keuntungan dari pengujian integrasi top-down adalah hanya memerlukan penulisan stub, dan stub lebih mudah ditulis dibandingkan dengan driver. Kelemahan dari pendekatan pengujian integrasi top-down adalah bahwa dengan tidak adanya rutinitas tingkat yang lebih rendah, menjadi sulit untuk melakukan rutinitas tingkat atas dengan cara yang diinginkan karena rutinitas tingkat yang lebih rendah biasanya melakukan input/output (I/O) operasi.

Pendekatan campuran untuk pengujian integrasi

Pengujian integrasi campuran (juga disebut sandwiched) mengikuti kombinasi pendekatan pengujian top-down dan bottom-up. Dalam pendekatan top-down, pengujian dapat dimulai hanya setelah modul tingkat atas dikodekan dan unit diuji. Demikian pula, pengujian dari bawah ke atas hanya dapat dimulai setelah modul tingkat bawah siap. Pendekatan campuran mengatasi kelemahan pendekatan top-down dan bottom-up ini. Dalam pendekatan pengujian campuran, pengujian dapat dimulai saat dan ketika modul tersedia setelah pengujian unit. Oleh karena itu, ini adalah salah satu pendekatan pengujian integrasi yang paling umum digunakan. Dalam pendekatan ini, baik stub maupun driver harus dirancang.

Pengujian Integrasi Bertahap versus Inkremental

Pengujian integrasi big-bang dilakukan dalam satu langkah integrasi. Sebaliknya, pada strategi lainnya, integrasi dilakukan melalui beberapa langkah. Dalam strategi selanjutnya, modul dapat diintegrasikan baik secara bertahap atau bertahap. Perbandingan kedua strategi ini adalah sebagai berikut:

- Dalam pengujian integrasi inkremental, hanya satu modul baru yang ditambahkan ke sistem yang terintegrasi sebagian setiap kali.
- Dalam integrasi bertahap, sekelompok modul terkait ditambahkan ke sistem parsial setiap kali.

Jelas, integrasi bertahap membutuhkan lebih sedikit langkah integrasi dibandingkan dengan pendekatan integrasi inkremental. Namun, ketika kegagalan terdeteksi, lebih mudah untuk men-debug sistem saat menggunakan pendekatan pengujian tambahan karena kesalahan dapat dengan mudah dilacak ke antarmuka modul yang baru saja terintegrasi. Harap perhatikan bahwa kasus penurunan pendekatan pengujian integrasi bertahap adalah pengujian big-bang.

10.11 MENGUJI PROGRAM BERORIENTASI OBJEK

Selama tahun-tahun awal pemrograman berorientasi objek, diyakini bahwa orientasi objek akan, sebagian besar, mengurangi biaya dan upaya yang dikeluarkan untuk pengujian. Pemikiran ini didasarkan pada pengamatan bahwa orientasi objek menggabungkan beberapa fitur pemrograman yang baik seperti enkapsulasi, abstraksi, penggunaan kembali melalui pewarisan, polimorfisme, dll., sehingga kemungkinan kesalahan dalam kode diminimalkan.

Namun, segera disadari bahwa pengujian memuaskan program berorientasi objek jauh lebih sulit dan membutuhkan lebih banyak biaya dan usaha dibandingkan dengan pengujian program prosedural serupa. Alasan utama di balik situasi ini adalah bahwa berbagai fitur berorientasi objek memperkenalkan komplikasi tambahan dan cakupan jenis bug baru yang ada dalam program prosedural. Oleh karena itu, kasus uji tambahan diperlukan untuk dirancang untuk mendeteksi ini.

Apa Unit yang Cocok untuk Pengujian?

Program berorientasi objek?

Untuk program prosedural, kita melihat bahwa prosedur adalah unit dasar pengujian. Artinya, pertama semua prosedur diuji unit. Kemudian berbagai prosedur yang diuji diintegrasikan bersama dan diuji. Jadi, sejauh menyangkut program prosedural, prosedur adalah unit dasar pengujian. Karena metode dalam program berorientasi objek analog dengan prosedur dalam program prosedural, dapatkah kita mempertimbangkan metode program berorientasi objek sebagai unit dasar pengujian? Weyuker mempelajari masalah ini dan mendalilkan aksioma antikomposisinya sebagai berikut: Pengujian metode individual yang memadai tidak memastikan bahwa suatu kelas telah diuji secara memuaskan.

Pembenaran intuitif utama untuk aksioma antikomposisi adalah sebagai berikut. Sebuah metode beroperasi dalam ruang lingkup data dan metode lain dari objeknya. Artinya, semua metode berbagi data kelas. Oleh karena itu, perlu dilakukan pengujian suatu metode dalam konteks tersebut. Selain itu, objek dapat memiliki jumlah status yang signifikan. Perilaku suatu metode dapat berbeda berdasarkan keadaan objek yang sesuai. Oleh karena itu, tidak cukup untuk menguji semua metode dan memeriksa apakah mereka dapat diintegrasikan dengan memuaskan. Suatu metode harus diuji dengan semua metode dan data lain dari objek yang sesuai. Selain itu, suatu metode perlu diuji pada semua status yang dapat diasumsikan oleh objek. Akibatnya, tidak tepat untuk mempertimbangkan metode sebagai unit dasar pengujian program berorientasi objek.

Objek adalah unit dasar pengujian program berorientasi objek. Jadi, dalam program berorientasi objek, pengujian unit berarti menguji setiap objek secara terpisah. Selama pengujian integrasi (disebut pengujian cluster dalam literatur pengujian berorientasi objek) berbagai objek yang diuji unit diintegrasikan dan diuji. Akhirnya, pengujian tingkat sistem dilakukan.

Apakah Berbagai Fitur Orientasi Objek Membuat Pengujian Mudah?

Pada bagian ini, kita membahas implikasi dari fitur orientasi objek yang berbeda dalam pengujian.

Enkapsulasi: fitur enkapsulasi membantu dalam abstraksi data, isolasi kesalahan, dan pencegahan kesalahan. Namun, sejauh menyangkut pengujian, enkapsulasi bukanlah halangan untuk pengujian, tetapi menyebabkan kesulitan selama debugging. Enkapsulasi mencegah penguji mengakses data internal ke suatu objek. Tentu saja, ada kemungkinan bahwa seseorang dapat meminta kelas untuk mendukung metode pelaporan status untuk mencetak semua data internal ke suatu objek. Dengan demikian, fitur enkapsulasi meskipun membuat pengujian menjadi sulit, kesulitan tersebut dapat diatasi sampai batas tertentu melalui penggunaan metode pelaporan keadaan yang sesuai.

Warisan: Fitur pewarisan membantu dalam penggunaan kembali kode dan diharapkan dapat menyederhanakan pengujian. Diharapkan jika sebuah kelas diuji secara menyeluruh, maka kelas yang diturunkan dari kelas ini hanya membutuhkan pengujian tambahan dari fitur yang ditambahkan. Namun, ini tidak terjadi. Bahkan jika kelas kelas dasar telah diuji secara menyeluruh, metode yang diwarisi dari kelas dasar perlu diuji lagi di kelas turunan.

Alasan untuk ini adalah bahwa metode yang diwarisi akan bekerja dalam konteks baru (data baru dan definisi metode). Akibatnya, perilaku yang benar dari suatu metode di tingkat atas, tidak menjamin perilaku yang benar di tingkat yang lebih rendah. Oleh karena itu, pengujian ulang metode yang diwariskan perlu diikuti sebagai aturan, bukan sebagai pengecualian.

Pengikatan dinamis: Pengikatan dinamis diperkenalkan untuk membuat kode menjadi ringkas, elegan, dan mudah diperluas. Namun, sejauh menyangkut pengujian, semua kemungkinan pengikatan dari pemanggilan metode harus diidentifikasi dan diuji. Ini tidak mudah karena pengikatan terjadi pada saat run-time.

Status objek: Berbeda dengan prosedur dalam program prosedural, objek menyimpan data secara permanen. Akibatnya, objek memang memiliki status signifikan. Perilaku suatu objek biasanya berbeda di negara bagian yang berbeda. Artinya, beberapa metode mungkin tidak aktif di beberapa statusnya. Juga, suatu metode dapat bertindak secara berbeda di negara bagian yang berbeda. Misalnya, ketika sebuah buku telah diterbitkan dalam sistem informasi perpustakaan, buku tersebut mencapai status dikeluarkan. Dalam keadaan ini, jika metode masalah dipanggil, maka mungkin tidak menunjukkan perilaku normalnya.

Mengingat diskusi di atas, menguji objek hanya dalam satu statusnya saja tidak cukup. Objek harus diuji pada semua kemungkinan keadaannya. Juga, apakah semua transisi antar status (seperti yang ditentukan dalam model objek) berfungsi dengan baik atau tidak harus diuji. Selain itu, perlu diuji bahwa tidak ada transisi tambahan (menyelinap), juga tidak ada status tambahan selain yang ditentukan dalam model status. Untuk pengujian berbasis keadaan, oleh karena itu bermanfaat untuk memiliki model keadaan objek, sehingga kesesuaian objek dengan model keadaannya dapat diuji.

Mengapa Teknik Tradisional Dianggap Tidak Memuaskan untuk Menguji Program Berorientasi Objek?

Kita telah melihat bahwa dalam program prosedural tradisional, prosedur adalah unit dasar pengujian. Sebaliknya, objek adalah unit dasar pengujian untuk program berorientasi objek. Selain itu, ada banyak perbedaan signifikan lainnya antara pengujian prosedural dan program berorientasi objek. Misalnya, pengujian berbasis cakupan pernyataan yang populer untuk menguji program prosedural tidak berarti untuk program berorientasi objek. Alasannya adalah bahwa metode yang diwarisi harus diuji ulang di kelas turunan. Faktanya, fitur berorientasi objek yang berbeda (warisan, polimorfisme, pengikatan dinamis, perilaku berbasis keadaan, dll.) memerlukan kasus uji khusus untuk dirancang dibandingkan dengan pengujian tradisional. Berbagai fitur orientasi objek eksplisit dalam model desain, dan biasanya sulit untuk mengekstrak dan menganalisis kode sumber. Akibatnya, model desain adalah artefak yang berharga untuk menguji program berorientasi objek. Kasus uji dirancang berdasarkan model desain. Oleh karena itu, pendekatan ini dianggap sebagai perantara antara pendekatan kotak putih penuh dan pendekatan kotak hitam penuh, dan disebut pendekatan kotak abu-abu. Harap dicatat bahwa pengujian kotak abu-abu dianggap penting untuk program berorientasi objek. Ini berbeda dengan pengujian program prosedural.

Pengujian Kotak Abu-abu untuk Program Berorientasi Objek

Pengujian berbasis model penting untuk program berorientasi objek, karena kasus uji ini membantu mendeteksi bug yang spesifik untuk konstruksi orientasi objek. Untuk program berorientasi objek, beberapa jenis kasus uji dapat dirancang berdasarkan model desain program berorientasi objek. Ini disebut kasus uji kotak abu-abu. Berikut ini adalah beberapa jenis pengujian kotak abu-abu penting yang dapat dilakukan berdasarkan model UML:

Pengujian berbasis model negara

- **Cakupan negara**: Setiap metode objek diuji pada setiap status objek.

- **Cakupan transisi status**: Diuji apakah semua transisi yang digambarkan dalam model status berfungsi dengan baik.
- **Cakupan jalur transisi status**: Semua jalur transisi dalam model status diuji.

Gunakan pengujian berbasis kasus

Cakupan skenario: Setiap use case biasanya terdiri dari skenario arus utama dan beberapa skenario alternatif. Untuk setiap kasus penggunaan, jalur utama dan semua urutan alternatif diuji untuk memeriksa apakah ada kesalahan yang muncul.

Pengujian berbasis diagram kelas

Pengujian kelas turunan: Semua kelas turunan dari kelas dasar harus dipakai dan diuji. Selain menguji metode baru yang didefinisikan dalam turunan. lass, metode yang diwarisi harus diuji ulang.

Pengujian asosiasi: Semua hubungan asosiasi diuji.

Pengujian agregasi: Berbagai objek agregat dibuat dan diuji.

Pengujian berbasis diagram urutan

Cakupan metode: Semua metode yang digambarkan dalam diagram urutan tercakup.

Cakupan jalur pesan: Semua jalur pesan yang dapat dibangun dari diagram urutan tercakup.

Pengujian Integrasi Program Berorientasi Objek

Ada dua pendekatan utama untuk pengujian integrasi program berorientasi objek:

- Berbasis benang
- Gunakan berdasarkan

Pendekatan berbasis utas: Dalam pendekatan ini, semua kelas yang perlu berkolaborasi untuk mewujudkan perilaku kasus penggunaan tunggal diintegrasikan dan diuji. Setelah semua kelas yang diperlukan untuk use case terintegrasi dan diuji, use case lain diambil dan kelas lain (jika ada) yang diperlukan untuk eksekusi use case kedua untuk dijalankan diintegrasikan dan diuji. Ini dilanjutkan sampai semua kasus penggunaan telah dipertimbangkan.

Pendekatan berbasis penggunaan: Integrasi berbasis penggunaan dimulai dengan menguji kelas yang tidak memerlukan layanan dari kelas lain atau membutuhkan layanan dari paling banyak beberapa kelas lain. Setelah kelas-kelas ini terintegrasi dan diuji, kelas-kelas yang menggunakan layanan dari kelas-kelas yang sudah terintegrasi diintegrasikan dan diuji. Hal ini berlanjut sampai semua kelas telah terintegrasi dan diuji.

10.12 PENGUJIAN SISTEM

Setelah semua unit program diintegrasikan dan diuji, pengujian sistem dilakukan. Pengujian sistem dirancang untuk memvalidasi sistem yang dikembangkan sepenuhnya untuk memastikan bahwa sistem tersebut memenuhi persyaratannya. Oleh karena itu, kasus uji dirancang hanya berdasarkan dokumen SRS.

Prosedur pengujian sistem adalah sama untuk program berorientasi objek dan prosedural, karena kasus pengujian sistem dirancang hanya berdasarkan dokumen SRS dan implementasi aktual (prosedural atau berorientasi objek) tidak material. Pada dasarnya ada tiga jenis utama pengujian sistem tergantung pada siapa yang melakukan pengujian:

1. **Pengujian Alfa**: Pengujian alfa mengacu pada pengujian sistem yang dilakukan oleh tim pengujian dalam organisasi yang sedang berkembang.
2. **Pengujian Beta**: Pengujian beta adalah pengujian sistem yang dilakukan oleh sekelompok pelanggan terpilih yang ramah.

3. **Pengujian Penerimaan**: Pengujian penerimaan adalah pengujian sistem yang dilakukan oleh pelanggan untuk menentukan apakah akan menerima pengiriman sistem.

Pada masing-masing jenis pengujian sistem di atas, kasus uji dapat sama, tetapi perbedaannya terletak pada siapa yang merancang kasus uji dan melakukan pengujian. Kasus uji sistem dapat diklasifikasikan ke dalam kasus uji fungsionalitas dan kinerja. Sebelum sistem yang terintegrasi penuh diterima untuk pengujian sistem, pengujian asap dilakukan. Pengujian asap dilakukan untuk memeriksa apakah setidaknya fungsi utama perangkat lunak berfungsi dengan baik. Kecuali jika perangkat lunak stabil dan setidaknya fungsi utama berfungsi dengan baik, pengujian sistem tidak dilakukan.

Tes fungsionalitas dirancang untuk memeriksa apakah perangkat lunak memenuhi persyaratan fungsional seperti yang didokumentasikan dalam dokumen SRS. Tes kinerja, di sisi lain, menguji kesesuaian sistem dengan persyaratan non-fungsional sistem. Kita telah membahas bagaimana merancang kasus uji fungsionalitas dengan menggunakan pendekatan kotak hitam.

Pengujian Asap

Pengujian asap dilakukan sebelum memulai pengujian sistem untuk memastikan bahwa pengujian sistem akan bermakna, atau apakah banyak bagian dari perangkat lunak akan gagal. Gagasan di balik pengujian asap adalah bahwa jika program terintegrasi tidak dapat lulus bahkan pengujian dasar, program tersebut belum siap untuk pengujian yang kuat. Untuk pengujian asap, beberapa kasus uji dirancang untuk memeriksa apakah fungsi dasar berfungsi. Misalnya, untuk sistem otomatisasi perpustakaan, tes asap dapat memeriksa apakah buku dapat dibuat dan dihapus, apakah catatan anggota dapat dibuat dan dihapus, dan apakah buku dapat dipinjamkan dan dikembalikan.

Pengujian Kinerja

Pengujian kinerja adalah jenis pengujian sistem yang penting. Pengujian kinerja dilakukan untuk memeriksa apakah sistem memenuhi persyaratan nonfungsional yang diidentifikasi dalam dokumen SRS. Ada beberapa jenis pengujian kinerja yang sesuai dengan berbagai jenis persyaratan non-fungsional. Untuk sistem tertentu, jenis pengujian kinerja yang akan dilakukan pada suatu sistem tergantung pada persyaratan non-fungsional yang berbeda dari sistem yang didokumentasikan dalam dokumen SRS-nya. Semua tes kinerja dapat dianggap sebagai tes kotak hitam.

Pengujian Stress atau Tegangan

Tes stres juga dikenal sebagai tes daya tahan. Pengujian stres mengevaluasi kinerja sistem ketika ditekan untuk waktu yang singkat. Tes stres adalah tes kotak hitam yang dirancang untuk memaksakan berbagai kondisi input yang tidak normal dan bahkan ilegal untuk menekankan kemampuan perangkat lunak. Volume data input, kecepatan data input, waktu pemrosesan, penggunaan memori, dll., diuji di luar kapasitas yang dirancang. Misalnya, sistem operasi seharusnya mendukung lima belas transaksi bersamaan, maka sistem ditekankan dengan mencoba memulai lima belas atau lebih transaksi secara bersamaan. Sebuah sistem real-time mungkin diuji untuk menentukan efek kedatangan simultan dari beberapa interupsi prioritas tinggi.

Pengujian tegangan sangat penting untuk sistem yang dalam keadaan normal beroperasi di bawah kapasitas maksimumnya tetapi mungkin sangat tertekan pada beberapa jam permintaan puncak. Misalnya, jika persyaratan nonfungsional yang sesuai menyatakan bahwa waktu respons tidak boleh lebih dari dua puluh detik per transaksi saat enam puluh pengguna bersamaan bekerja, maka selama pengujian stres, waktu respons diperiksa dengan tepat enam puluh pengguna yang bekerja secara bersamaan.

Pengujian Volume

Pengujian volume memeriksa apakah struktur data (buffer, array, antrian, tumpukan, dll.) telah dirancang untuk berhasil menangani situasi luar biasa. Misalnya, pengujian volume untuk kompiler mungkin untuk memeriksa apakah tabel simbol meluap ketika program yang sangat besar dikompilasi.

Pengujian konfigurasi

Pengujian konfigurasi digunakan untuk menguji perilaku sistem dalam berbagai konfigurasi perangkat keras dan perangkat lunak yang ditentukan dalam persyaratan. Terkadang sistem dibangun untuk bekerja dalam konfigurasi yang berbeda untuk pengguna yang berbeda. Misalnya, sistem minimal mungkin diperlukan untuk melayani satu pengguna, dan konfigurasi tambahan lainnya mungkin diperlukan untuk melayani pengguna tambahan selama pengujian konfigurasi. Sistem dikonfigurasikan di setiap konfigurasi yang diperlukan dan tergantung pada kebutuhan pelanggan tertentu, sistem akan diperiksa apakah sistem berfungsi dengan benar di semua konfigurasi yang diperlukan.

Pengujian kompatibilitas

Jenis pengujian ini diperlukan ketika sistem berinteraksi dengan sistem eksternal (misalnya, database, server, dll.). Kompatibilitas bertujuan untuk memeriksa apakah antarmuka dengan sistem eksternal bekerja sesuai kebutuhan. Misalnya, jika sistem perlu berkomunikasi dengan sistem database besar untuk mengambil informasi, pengujian kompatibilitas diperlukan untuk menguji kecepatan dan akurasi pengambilan data.

Pengujian regresi

Jenis pengujian ini diperlukan ketika perangkat lunak dipelihara untuk memperbaiki beberapa bug atau meningkatkan fungsionalitas, kinerja, dll.

Pengujian pemulihan

Pengujian pemulihan menguji respons sistem terhadap adanya kesalahan, atau kehilangan daya, perangkat, layanan, data, dll. Sistem mengalami kehilangan sumber daya yang disebutkan (seperti yang dibahas dalam dokumen SRS) dan diperiksa jika sistem pulih dengan memuaskan. Misalnya, printer dapat diputuskan untuk memeriksa apakah sistem hang. Atau, daya mungkin dimatikan untuk memeriksa tingkat kehilangan data dan kerusakan.

Pengujian pemeliharaan

Ini membahas pengujian program diagnostik, dan prosedur lain yang diperlukan untuk membantu pemeliharaan sistem. Diverifikasi bahwa artefak itu ada dan berfungsi dengan baik.

Pengujian dokumentasi

Diperiksa apakah manual pengguna yang diperlukan, manual pemeliharaan, dan manual teknis ada dan konsisten. Jika persyaratan menentukan jenis audiens yang manual khusus harus dirancang, maka manual diperiksa untuk memenuhi persyaratan ini.

Pengujian kegunaan

Pengujian kegunaan menyangkut pemeriksaan antarmuka pengguna untuk melihat apakah itu memenuhi semua persyaratan pengguna mengenai antarmuka pengguna. Selama pengujian kegunaan, layar tampilan, pesan, format laporan, dan aspek lain yang berkaitan dengan persyaratan antarmuka pengguna diuji. GUI yang benar secara fungsional saja tidak cukup. Oleh karena itu, GUI harus diperiksa terhadap daftar periksa yang kita bahas di Sec. 9.5.6.

Pengujian keamanan

Pengujian keamanan sangat penting untuk perangkat lunak yang menangani atau memproses data rahasia yang akan dilindungi dari pencurian. Perlu diuji apakah sistem tersebut aman dari serangan keamanan seperti penyusupan oleh peretas. Selama beberapa

tahun terakhir, sejumlah besar teknik pengujian keamanan telah diusulkan, dan ini termasuk cracking kata sandi, pengujian penetrasi, dan serangan terhadap port tertentu, dll.

Penyemaian Kesalahan

Terkadang pelanggan menentukan jumlah maksimum kesalahan sisa yang dapat ada dalam perangkat lunak yang dikirimkan. Persyaratan ini sering dinyatakan dalam jumlah maksimum kesalahan yang diizinkan per baris kode sumber. Teknik penyemaian kesalahan dapat digunakan untuk memperkirakan jumlah kesalahan sisa dalam suatu perangkat lunak.

Penyemaian kesalahan, seperti namanya, melibatkan penyemaian kode dengan beberapa kesalahan yang diketahui. Dengan kata lain, beberapa kesalahan buatan dimasukkan (diunggulkan) ke dalam program. Jumlah kesalahan yang diunggulkan ini yang terdeteksi selama pengujian standar ditentukan. Nilai-nilai ini dalam hubungannya dengan jumlah kesalahan yang tidak diunggulkan yang terdeteksi selama pengujian dapat digunakan untuk memprediksi aspek-aspek berikut dari suatu program:

- Jumlah kesalahan yang tersisa dalam produk.
- Efektivitas strategi pengujian.

Biarkan N menjadi jumlah total cacat dalam sistem, dan biarkan n cacat ini ditemukan dengan pengujian. Misalkan S adalah jumlah total cacat yang diunggulkan, dan s dari cacat ini ditemukan selama pengujian. Oleh karena itu, kita mendapatkan:

$$\frac{n}{N} = \frac{s}{S}$$

atau

$$N = S\,x\frac{n}{s}$$

Cacat yang masih tersisa dalam program setelah pengujian dapat diberikan oleh:

$$N - n = n\,x\,\frac{(S-1)}{s}$$

Penyemaian kesalahan bekerja dengan memuaskan hanya jika jenis kesalahan yang disemaikan dan frekuensi kemunculannya cocok dengan jenis cacat yang sebenarnya ada. Namun, sulit untuk memprediksi jenis kesalahan yang ada pada suatu perangkat lunak. Sampai batas tertentu, kategori kesalahan yang berbeda yang laten dan frekuensi kemunculannya dapat diperkirakan dengan menganalisis data historis yang dikumpulkan dari proyek serupa. Artinya, data yang dikumpulkan adalah mengenai jenis dan frekuensi kesalahan laten untuk semua proyek terkait sebelumnya. Ini memberikan indikasi jenis (dan frekuensi) kesalahan yang mungkin telah dilakukan dalam program yang sedang dipertimbangkan. Berdasarkan data ini, berbagai jenis kesalahan dengan frekuensi kemunculan yang diperlukan dapat diunggulkan.

10.13 BEBERAPA MASALAH UMUM YANG TERKAIT DENGAN PENGUJIAN

Pada bagian ini, kita akan membahas dua masalah umum yang terkait dengan pengujian. Ini adalah—bagaimana mendokumentasikan hasil pengujian dan bagaimana melakukan pengujian regresi.

Dokumentasi tes

Sepotong dokumentasi yang dihasilkan menjelang akhir pengujian adalah laporan ringkasan pengujian. Laporan ini biasanya mencakup setiap subsistem dan mewakili ringkasan pengujian yang telah diterapkan pada subsistem dan hasilnya. Biasanya menentukan hal berikut:

- Berapa jumlah total tes yang diterapkan pada subsistem.
- Dari jumlah total tes, berapa banyak tes yang berhasil.

- Berapa banyak yang tidak berhasil, dan sejauh mana mereka tidak berhasil, misalnya, apakah suatu tes benar-benar gagal atau apakah beberapa hasil tes yang diharapkan benar-benar diamati.

Pengujian regresi

Pengujian regresi mencakup pengujian unit, integrasi, dan sistem. Sebaliknya, ini adalah dimensi terpisah dari ketiga bentuk pengujian ini. Pengujian regresi adalah praktik menjalankan rangkaian pengujian lama setelah setiap perubahan pada sistem atau setelah setiap perbaikan bug untuk memastikan bahwa tidak ada bug baru yang diperkenalkan karena perubahan atau perbaikan bug. Namun, jika hanya beberapa pernyataan yang diubah, maka seluruh rangkaian pengujian tidak perlu dijalankan — hanya kasus pengujian yang menguji fungsi dan kemungkinan terpengaruh oleh perubahan yang perlu dijalankan. Setiap kali perangkat lunak diubah untuk memperbaiki bug, atau meningkatkan atau menghapus fitur, pengujian regresi dilakukan.

10.14 RINGKASAN

- Sebagian besar organisasi pengembangan perangkat lunak merumuskan standar pengkodean mereka sendiri dan mengharapkan para insinyur mereka untuk mematuhinya. Di sisi lain, pedoman pengkodean berfungsi sebagai saran umum untuk programmer mengenai gaya pemrograman yang baik, tetapi penerapan pedoman diserahkan kepada kebijaksanaan masing-masing insinyur.
- Tinjauan kode adalah cara yang efisien untuk menghilangkan kesalahan dibandingkan dengan pengujian, karena tinjauan kode mengidentifikasi kesalahan sedangkan pengujian mengidentifikasi kegagalan. Oleh karena itu, setelah mengidentifikasi kegagalan, upaya tambahan (debugging) harus dilakukan untuk menemukan dan memperbaiki kesalahan.
- Pengujian menyeluruh dari hampir semua sistem non-sepele tidak praktis. Juga, pemilihan acak kasus uji tidak efisien karena banyak kasus uji menjadi berlebihan karena mendeteksi jenis kesalahan yang sama. Oleh karena itu, kita perlu merancang satu set kasus uji minimal yang akan mengekspos kesalahan sebanyak mungkin.
- Ada dua pendekatan terkenal untuk pengujian
    - pengujian kotak hitam
    - pengujian kotak putih.
- Pengujian kotak hitam juga dikenal sebagai pengujian fungsional. Merancang kasus uji untuk pengujian kotak hitam tidak memerlukan pengetahuan tentang bagaimana fungsi telah dirancang dan diimplementasikan. Di sisi lain, pengujian kotak putih membutuhkan pengetahuan tentang internal perangkat lunak.
- Fitur berorientasi objek memperumit proses pengujian karena kasus uji harus dirancang untuk mendeteksi bug yang terkait dengan jenis fitur baru ini yang khusus untuk program orientasi objek.
- Rangkaian pengujian sistem dirancang berdasarkan dokumen SRS. Dua jenis utama pengujian sistem adalah pengujian fungsionalitas dan pengujian kinerja. Kasus uji fungsionalitas dirancang berdasarkan persyaratan fungsional dan kasus uji kinerja dirancang untuk menguji kepatuhan sistem untuk menguji persyaratan non-fungsional yang didokumentasikan dalam dokumen SRS.

10.15 LATIHAN

1. Untuk setiap pertanyaan berikut, pilih opsi yang benar:

a. Kapan tinjauan kode dilakukan selama siklus hidup perangkat lunak?
    i. Setelah pengujian unit
    ii. Setelah pengkodean dan kompilasi
    iii. Selama pengujian integrasi
    iv. Selama pengujian sistem
b. Manakah dari pernyataan berikut yang benar?
    i. Inspeksi kode dilakukan pada kode yang diuji dan di-debug.
    ii. Inspeksi kode dan penelusuran kode pada dasarnya sama.
    iii. Kepatuhan terhadap standar pengkodean diperiksa selama kode inspeksi.
    iv. Panduan kode membuat inspeksi kode menjadi mubazir.
c. Identifikasi sinonim dari: kesalahan, bug, kesalahan, kegagalan, dan kesalahan:
    i. kesalahan, bug, kesalahan, kegagalan, dan kesalahan
    ii. kesalahan, bug, kegagalan
    iii. kesalahan, kesalahan, kesalahan
    iv. bug, kegagalan, kesalahan
d. Manakah dari berikut ini yang bukan merupakan teknik pengujian perangkat lunak yang dikenal?
    i. Pengujian aliran data
    ii. Pengujian jalur
    iii. Pengujian sintaks
    iv. Pengujian keputusan
e. Pengujian unit modul perangkat lunak TIDAK memerlukan pengujian yang mana dari berikut ini:
    i. Apakah standar pengkodean telah diikuti.
    ii. Apakah fungsi modul bekerja sesuai desain.
    iii. Apakah semua pernyataan aritmatika modul bekerja dengan benar.
    iv. Apakah semua pernyataan kontrol bekerja dengan benar.
f. Manakah dari aktivitas verifikasi dan validasi (V a n d V) berikut yang menargetkan untuk mendeteksi ketidakpatuhan terhadap standar pengkodean?
    i. Pengujian unit
    ii. Inspeksi kode
    iii. Panduan kode
    iv. Pengujian sistem
g. Peninjauan kode tidak menargetkan untuk mendeteksi yang mana dari jenis pengujian berikut:
    i. Kesalahan algoritma
    ii. Kesalahan sintaksis
    iii. Kesalahan pemrograman
    iv. Kesalahan logika
h. Kompleksitas siklomatik McCabe didefinisikan dalam hal berikut ini?
    i. Grafik sintaks
    ii. Diagram aliran data
    iii. Diagram alir kontrol
    iv. Bagan struktur
i. Manakah dari berikut ini yang benar tentang verifikasi program?
    i. Memeriksa bahwa kita sedang membangun sistem yang benar

ii. Memeriksa bahwa kita sedang membangun sistem dengan benar
iii. Dilakukan oleh tim penguji independen
iv. Memastikan bahwa produk yang dikembangkan sesuai dengan keinginan pengguna

j. Manakah dari pernyataan berikut yang bukan merupakan tujuan verifikasi perangkat lunak?
   i. Memastikan bahwa langkah-langkah pengembangan produk dilakukan dengan benar.
   ii. Memastikan bahwa produk yang benar telah dikembangkan.
   iii. Mencapai fase penahanan kesalahan.
   iv. Memastikan bahwa output yang dihasilkan pada suatu tahap sesuai dengan output fase sebelumnya.
k. Manakah dari berikut ini yang tidak membantu dalam mencapai fase penahanan kesalahan?
   i. Pengujian sistem
   ii. Tinjauan
   iii. Pembuatan prototipe
   iv. Simulasi
l. Setelah program dimodifikasi, manakah dari opsi berikut yang mencirikan kasus uji regresi?
   i. Semua kasus uji
   ii. Uji kasus yang mengeksekusi pernyataan yang dimodifikasi
   iii. Uji kasus yang mengeksekusi pernyataan yang terpengaruh atau dimodifikasi
   iv. Uji kasus yang mengeksekusi pernyataan yang tidak terpengaruh
m. Manakah dari berikut ini yang merupakan pendekatan pengujian kotak hitam?
   i. Pengujian jalur
   ii. Pengujian nilai batas
   iii. Pengujian mutasi
   iv. Pengujian cabang
n. Manakah dari berikut ini yang bukan merupakan teknik verifikasi perangkat lunak?
   i. Tinjauan
   ii. Simulasi
   iii. Pengujian unit
   iv. Pembuktian teorema
   v. Pengecekan model
   vi. Inspeksi
   vii. Tes stres
o. Mengapa penting untuk menguji nilai batas saat menguji suatu fungsi?
   i. Ini mengurangi biaya pengujian karena nilai batas mudah dihitung dengan tangan.
   ii. Debugging lebih mudah saat menguji nilai batas.
   iii. Eksekusi yang benar dari suatu fungsi pada semua nilai batas membuktikan bahwa suatu fungsi benar.
   iv. Dalam prakteknya, pemrograman kondisi batas rawan kesalahan.
p. Manakah dari berikut ini yang dapat dianggap sebagai teknik validasi program?
   i. Pengujian unit

ii. Pengujian integrasi
iii. Tinjauan kode
iv. Pengujian penerimaan

q. Jika cakupan cabang telah dicapai pada unit yang diuji, yang mana dari cakupan berikut tersirat secara implisit?
   i. Cakupan jalur
   ii. Cakupan berbagai kondisi
   iii. Cakupan pernyataan
   iv. Cakupan aliran data
r. Manakah dari atribut program berikut yang dapat disimpulkan dari kompleksitas siklomatik suatu program?
   i. Kompleksitas komputasi
   ii. Baris kode (LoC)
   iii. Ukuran kode yang dapat dieksekusi
   iv. Dapat dimengerti

2. Untuk setiap pertanyaan berikut, pilih opsi yang benar:
   a. Manakah dari pernyataan berikut tentang kompleksitas metrik siklomatik program adalah SALAH?
      i. Ini adalah ukuran kesulitan pengujian program.
      ii. Ini adalah ukuran kesulitan memahami program.
      iii. Ini adalah ukuran jalur bebas linier dalam program
      iv. Ini adalah ukuran ukuran program
   b. Pengujian Alfa dan Beta dianggap sebagai salah satu dari jenis pengujian berikut?
      i. Pengujian regresi
      ii. Pengujian unit
      iii. Pengujian integrasi
      iv. Pengujian penerimaan
   c. Tujuan penyemaian kesalahan adalah yang mana dari berikut ini?
      i. Tentukan asal bug
      ii. Tanaman trojan
      iii. Tentukan jumlah bug laten
      iv. Tanam serangga berbahaya sebelum dikirim ke pelanggan
   d. Kapan dalam siklus pengembangan tinjauan kode dilakukan?
      i. Setelah pengkodean selesai dan sebelum kode dikompilasi.
      ii. Setelah pengkodean selesai dan setelah kode dikompilasi.
      iii. Setelah pengujian unit selesai
      iv. Setelah pengujian sistem selesai
   e. Jika ekspresi kondisi dalam pernyataan bersyarat terdiri dari n kondisi atom, berapakah jumlah kasus uji yang diperlukan untuk mencapai cakupan kondisi ganda?
      i. n
      ii. 2n
      iii. 2 × n
      iv. 2 × n + 1
   f. Untuk program besar yang salah satu dari strategi pengujian integrasi berikut ini jarang digunakan:
      i. Big-bang

ii. Atas-bawah
iii. Bawah-atas
iv. Campuran

g. Manakah dari berikut ini yang benar dari proses pengujian integrasi top-down murni?
i. Hanya membutuhkan stub untuk pengujian
ii. Hanya membutuhkan driver untuk pengujian
iii. Membutuhkan stub dan driver untuk pengujian
iv. Tidak memerlukan stub atau driver untuk pengujian

h. Manakah dari jenis pengujian berikut yang tidak dilakukan selama pengujian sistem?
i. Tes stres
ii. Pengujian fungsionalitas
iii. Pengujian pemulihan
iv. Pengujian kotak putih

i. Tujuan utama dari analisis cakupan kode adalah untuk mengevaluasi kualitas dari:
i. Produk
ii. Kasus uji
iii. Pengkodean
iv. Desain

j. Manakah dari jenis model program berikut yang biasanya digunakan untuk merancang rencana uji integrasi?
i. CFG
ii. DFD
iii. Bagan struktur
iv. Bagan negara bagian

k. Manakah dari perangkat lunak berikut yang paling membantu Anda menentukan apakah kasus uji Anda sepenuhnya menjalankan kode Anda?
i. Penganalisis dinamis
ii. Penganalisis statis
iii. Pengurai
iv. Profiler

l. Manakah dari strategi pengujian integrasi berikut yang memerlukan desain stub?
i. Big-bang
ii. Atas-bawah
iii. Boot-up
iv. Bertahap dari bawah ke atas

m. Manakah dari strategi pengujian integrasi berikut yang memerlukan driver untuk dirancang?
i. Big-bang
ii. Atas-bawah
iii. Bawah-atas
iv. Top-down bertahap

n. Tujuan penyemaian kesalahan adalah yang mana dari berikut ini?
i. Tentukan akar penyebab kesalahan
ii. Menggabungkan Trojan dengan tepat

iii. Memasukkan kesalahan Bizantium dengan tepat
iv. Tentukan jumlah kesalahan laten

o. Manakah dari berikut ini yang biasanya menjadi pertimbangan paling penting saat membuat kode?
   i. Produktivitas
   ii. Keterbacaan
   iii. Singkat
   iv. Gunakan ruang memori sesedikit mungkin

3. Bedakan antara kesalahan dan kegagalan dalam konteks pengujian program. Pengujian mendeteksi yang mana dari keduanya? Justifikasi jawaban Anda.
4. Apakah Anda akan mempertimbangkan pendekatan di mana penguji menguji program menggunakan sejumlah besar nilai acak yang memuaskan? Jelaskan jawaban Anda.
5. Apa yang dimaksud dengan modul driver dan stub dalam konteks integrasi dan pengujian unit perangkat lunak? Mengapa modul rintisan dan driver diperlukan?
6. Nyatakan BENAR atau SALAH dari pernyataan berikut. Dukung jawaban Anda dengan alasan yang tepat:
   a. Efektivitas rangkaian uji dalam mendeteksi kesalahan dalam suatu sistem dapat ditentukan dengan menghitung jumlah kasus uji dalam rangkaian.
   b. Setelah kompleksitas siklomatik program McCabe telah ditentukan, sangat mudah untuk mengidentifikasi semua jalur bebas linier dari program tersebut.
   c. Penggunaan alat analisis program statis dan dinamis merupakan pengganti yang efektif untuk pengujian menyeluruh.
   d. Selama peninjauan kode, Anda mendeteksi kesalahan sedangkan selama pengujian kode, Anda mendeteksi kegagalan.
   e. Pengujian integrasi top-down murni tidak memerlukan penggunaan modul rintisan apa pun.
   f. Kepatuhan terhadap standar pengkodean diperiksa selama tahap pengujian sistem.
   g. Sebuah program biasanya tidak memiliki satu set unik jalur bebas linier.
   h. Jumlah minimum kasus uji yang diperlukan untuk pengujian berbasis cakupan cabang dari suatu program dapat lebih besar daripada yang diperlukan untuk pengujian berbasis cakupan jalur dari program yang sama.
   i. Pengujian berbasis cakupan cabang adalah strategi pengujian yang lebih kuat dibandingkan dengan pengujian berbasis cakupan jalur.
   j. Dari semua jenis dokumentasi internal (yaitu, disediakan dalam kode sumber), komentar yang cermat adalah yang paling berguna.
   k. Kesalahan dan kegagalan adalah sinonim dalam terminologi pengujian perangkat lunak.
   l. Pengembangan fungsi driver dan stub yang sesuai sangat penting untuk melaksanakan pengujian sistem yang efektif dari suatu produk.
   m. Pengujian sistem dapat dianggap sebagai pengujian kotak putih dari suatu sistem.
   n. Tujuan utama dari pengujian integrasi adalah untuk menemukan kesalahan dalam tubuh fungsi.
   o. Pengenalan jenis urutan pernyataan tambahan dalam suatu program tidak dapat meningkatkan kompleksitas siklomatik program.
   p. Istilah verifikasi perangkat lunak dan validasi perangkat lunak pada dasarnya adalah sinonim.

q. Panduan kode untuk modul biasanya dilakukan setelah pengujian unit selesai.
r. Panduan kode untuk sebuah modul biasanya dilakukan setelah modul berhasil dikompilasi.
s. Selama penelusuran kode, sebagian besar kesalahan sintaks diidentifikasi.
t. Target inspeksi kode untuk mengidentifikasi kesalahan algoritmik.
u. Kompleksitas siklomatik dari sepotong kode berkorelasi baik dengan kesulitan menguji kode secara memuaskan.
v. Penganalisis cakupan pengujian pada dasarnya adalah penganalisis statis.
w. Pengujian sistem dari implementasi sistem yang berorientasi objek akan jauh lebih mudah daripada implementasi prosedural dari sistem yang sama.
x. Cara yang memuaskan untuk menguji program berorientasi objek, adalah dengan menguji semua metode yang didukung oleh kelas yang berbeda secara individual dan kemudian dengan melakukan integrasi dan pengujian sistem yang memadai.
y. Kompilator dapat dianggap sebagai alat analisis program statis.
z. Sementara verifikasi berkaitan dengan fase penahanan kesalahan, tujuan validasi adalah bahwa produk akhir bebas dari kesalahan.

7. Nyatakan BENAR atau SALAH dari pernyataan berikut. Dukung jawaban Anda dengan alasan yang tepat:
   a. Pengujian unit dari berbagai modul program dilakukan selama fase pengujian.
   b. Jika pengujian yang lebih kuat telah dilakukan, maka pengujian yang lebih lemah tidak perlu dilakukan.
   c. Penganalisis kode statis dapat dengan mudah mendeteksi semua jenis jenis kesalahan "indeks array di luar batas".
   d. Pengujian kinerja direncanakan berdasarkan persyaratan fungsional produk yang diuji.
   e. Pendekatan pengujian Cleanroom membantu secara substansial mengurangi keseluruhan upaya pengujian.
   f. Jika kelas dasar diuji secara menyeluruh, maka metode yang diwarisi dalam kelas turunan tidak perlu diuji.
   g. Pembagian kelas kesetaraan adalah strategi pengujian kotak putih.
   h. Pendekatan big-bang lebih disukai untuk pengujian integrasi program besar.
   i. Rintisan uji lebih mudah ditulis dibandingkan dengan penggerak uji.
   j. Pedoman pengkodean adalah saran khusus untuk pemrogram, yang mungkin mereka ikuti atau tidak.
   k. Misalkan jumlah loop dan konstruksi kondisional yang digunakan dalam suatu program adalah n, maka kardinalitas himpunan jalur basisnya adalah 2*n−1.
   l. Jika pengujian kotak hitam suatu program telah berhasil dilakukan, maka pengujian kotak putih dapat dilewati dan sebaliknya.
   m. Cakupan pernyataan tidak dianggap sebagai pengujian yang memuaskan dari unit program. Jelaskan secara singkat alasan di balik ini. Berikan contoh bug, yang tidak akan terdeteksi melalui pengujian cakupan pernyataan.
   n. Prosedur pengujian sistem akan berbeda tergantung pada apakah paradigma berorientasi objek atau prosedural telah diikuti dalam pengembangan program.
   o. Pengujian mendeteksi kegagalan sedangkan inspeksi program mengidentifikasi kesalahan.

8. Apa perbedaan antara pengujian kotak hitam dan pengujian kotak putih? Berikan contoh bug yang terdeteksi oleh paket pengujian kotak hitam, tetapi tidak terdeteksi oleh paket pengujian kotak putih, dan sebaliknya.
9. Apa perbedaan antara dokumentasi internal dan eksternal? Apa saja cara berbeda untuk menyediakan dokumentasi internal? Dari semua ini, mana yang paling berguna?
10. Apa yang dimaksud dengan kompleksitas struktural suatu program? Tentukan metrik untuk mengukur kompleksitas struktural suatu program. Bagaimana kompleksitas struktural suatu program berbeda dari kompleksitas komputasinya? Bagaimana kompleksitas struktural berguna dalam pengembangan program?
11. Tulis fungsi C untuk mencari nilai integer dari urutan besar yang diurutkan dari nilai integer yang disimpan dalam larik berukuran 100, menggunakan metode pencarian biner.
    a. Bangun grafik aliran kontrol dari fungsi pencarian biner Anda, dan dengan demikian tentukan kompleksitas siklomatiknya.
    b. Bagaimana metrik siklomatik berguna dalam merancang rangkaian uji untuk cakupan jalur?
    c. Rancang rangkaian pengujian untuk menguji fungsi pencarian biner Anda.
12. Apa yang Anda pahami tentang kasus uji positif dan negatif? Berikan satu contoh masing-masing.
13. Diberikan perangkat lunak dan dokumen spesifikasi persyaratannya, jelaskan bagaimana Anda akan merancang rangkaian pengujian sistem untuk perangkat lunak tersebut.
14. Apa itu standar pengkodean? Identifikasi masalah yang mungkin terjadi jika para insinyur organisasi tidak mematuhi standar pengkodean apa pun?
15. Apa perbedaan antara standar pengkodean dan pedoman pengkodean? Mengapa perumusan dan penggunaan standar dan pedoman pengkodean yang sesuai dianggap penting bagi organisasi pengembangan perangkat lunak? Tuliskan lima standar pengkodean penting dan pedoman pengkodean yang akan Anda rekomendasikan.
16. Apa yang Anda pahami dengan standar pengkodean? Kapan selama proses pengembangan kepatuhan terhadap standar pengkodean diperiksa? Sebutkan dua standar pengkodean masing-masing untuk
    a. meningkatkan keterbacaan kode,
    b. penggunaan kembali kode,
    c. meningkatkan pemeliharaan kode.
17. Apa yang Anda pahami dengan testabilitas suatu program? Di antara program yang ditulis oleh dua programmer berbeda untuk masalah pemrograman yang pada dasarnya sama, bagaimana Anda bisa menentukan mana yang lebih dapat diuji?
18. Diskusikan berbagai jenis ulasan kode. Jelaskan kapan dan bagaimana pertemuan tinjauan kode dilakukan. Mengapa tinjauan kode dianggap sebagai cara yang lebih efisien untuk menghapus kesalahan dari kode dibandingkan dengan pengujian?
19. Bedakan antara verifikasi perangkat lunak dan validasi perangkat lunak. Bisakah satu digunakan sebagai pengganti yang lain? Justifikasi jawaban Anda. Di fase mana dari iterative waterfall SDLC kegiatan verifikasi dan validasi dilakukan?
20. Apa saja kegiatan yang dilakukan selama pengujian perangkat lunak? Secara skematis mewakili kegiatan ini. Manakah dari kegiatan ini yang membutuhkan upaya maksimal?
21. Manakah dari berikut ini yang merupakan teknik pengujian struktural terkuat — pengujian berbasis cakupan pernyataan, pengujian berbasis cakupan cabang, atau pengujian berbasis cakupan kondisi ganda? Justifikasi jawaban Anda.

22. Buktikan bahwa teknik pengujian berbasis cakupan cabang adalah teknik pengujian yang lebih kuat dibandingkan dengan teknik pengujian berbasis cakupan pernyataan.
23. Manakah yang merupakan pengujian yang lebih kuat—pengujian aliran data atau pengujian jalur? Berikan alasan di balik jawaban Anda.
24. Soroti secara singkat perbedaan antara pemeriksaan kode dan penelusuran kode. Bandingkan manfaat relatif dari pemeriksaan kode dan penelusuran kode.
25. Apa yang dimaksud dengan panduan kode? Apa saja jenis kesalahan penting yang diperiksa selama penelusuran kode? Berikan satu contoh dari masing-masing jenis kesalahan ini.
26. Jawab pertanyaan dibawah ini. Tunjukkan langkah-langkah perhitungan Anda, dan berikan alasan untuk setiap jawaban Anda.
    a. Misalkan sebuah program berisi N titik keputusan, yang masing-masing memiliki dua cabang. Berapa banyak kasus uji yang diperlukan untuk pengujian cabang?
    b. Jika terdapat M pilihan pada setiap titik keputusan, berapa banyak kasus uji yang diperlukan untuk pengujian cabang? Apakah mungkin untuk mencapai cakupan cabang menggunakan jumlah kasus uji yang lebih sedikit daripada yang Anda jawab tergantung pada kondisi cabang?
    c. Suatu program terdiri dari m urutan jenis pernyataan, n pernyataan keputusan, dan pernyataan p iteratif. Tentukan jumlah kasus uji yang diperlukan untuk mencapai cakupan keputusan dan cakupan jalur masing-masing.
    d. Untuk program yang berisi N cabang biner, berapa banyak kasus uji yang diperlukan untuk cakupan jalur?
    e. Untuk program yang berisi N jumlah cabang M-ary, berapa banyak kasus uji yang diperlukan untuk cakupan jalur?
27. Misalkan dua programmer diberi masalah pemrograman yang sama dan mereka mengembangkannya secara independen. Jelaskan bagaimana Anda dapat membandingkan program mereka sehubungan dengan:
    a. Upaya pengujian jalur,
    b. Kesulitan memahami
    c. Jumlah bug laten, dan
    d. Keandalan.
28. Biasanya produk perangkat lunak besar diuji pada tiga tingkat pengujian yang berbeda, yaitu pengujian unit, pengujian integrasi, dan pengujian sistem. Apa kerugian melakukan pengujian menyeluruh hanya setelah sistem dikembangkan sepenuhnya, misalnya, mendeteksi semua cacat produk selama pengujian sistem?
29. Apa yang Anda pahami dengan pengujian sistem? Apa saja jenis pengujian sistem yang biasanya dilakukan pada produk perangkat lunak besar?
30. Apakah pengujian sistem untuk program berorientasi objek berbeda dengan program prosedural? Jelaskan jawaban Anda.
31. Apakah pengujian integrasi program berorientasi objek berbeda dengan program prosedural? Jelaskan jawaban Anda.
32. Dengan menggunakan contoh yang sesuai, jelaskan bagaimana kasus uji dapat dirancang untuk program berorientasi objek dari diagram kelasnya.
33. Dengan menggunakan contoh yang sesuai, jelaskan bagaimana kasus uji dapat dirancang untuk program berorientasi objek dari diagram urutannya.

34. Bedakan antara pengujian alfa, beta, dan penerimaan. Bagaimana kasus uji dirancang untuk pengujian ini? Apakah kasus uji untuk ketiga jenis pengujian harus identik? Jelaskan jawaban Anda.
35. Kegunaan produk perangkat lunak diuji selama jenis pengujian apa: pengujian unit, integrasi, atau sistem? Bagaimana kegunaan diuji?
36. Misalkan perangkat lunak yang dikembangkan telah berhasil melewati ketiga level pengujian, yaitu pengujian unit, pengujian integrasi, dan pengujian sistem. Bisakah kita mengklaim bahwa perangkat lunak itu bebas cacat? Justifikasi jawaban Anda.
37. Bedakan antara test case, test suite, test scenario, dan test script.
38. Bedakan antara analisis statis dan dinamis dari suatu program. Jelaskan setidaknya satu metrik yang dilaporkan oleh alat analisis statis dan setidaknya satu metrik yang dilaporkan oleh alat analisis dinamis. Bagaimana metrik ini berguna?
39. Apa hasil penting yang biasanya dilaporkan oleh alat analisis statis dan alat analisis dinamis ketika diterapkan pada program yang sedang dikembangkan? Bagaimana hasil ini berguna?
40. Apa yang Anda pahami dengan analisis program otomatis? Berikan klasifikasi luas dari berbagai jenis alat analisis program yang digunakan selama pengembangan program. Apa jenis informasi yang berbeda yang dihasilkan oleh setiap jenis alat?
41. Rancang rangkaian pengujian kotak hitam untuk fungsi yang memeriksa apakah string karakter (panjangnya hingga dua puluh lima karakter) adalah palindrom.
42. Rancang rangkaian uji kotak hitam untuk fungsi yang mengambil nama buku sebagai input dan mencari file yang berisi nama-nama buku yang tersedia di Perpustakaan dan menampilkan detail buku jika buku tersedia di perpustakaan jika tidak, menampilkan pesan "buku tidak tersedia".
43. Mengapa penting untuk mendokumentasikan perangkat lunak dengan benar? Apa saja cara berbeda untuk mendokumentasikan produk perangkat lunak?
44. Apakah Anda mengerti dengan strategi kamar bersih? Apa kelebihannya?
45. Bagaimana Anda bisa menghitung kompleksitas siklomatik suatu program? Bagaimana kompleksitas siklomatik berguna dalam pengujian program?
46. Misalkan untuk memperkirakan jumlah kesalahan laten dalam suatu program, Anda menaburkannya dengan berbagai jenis kesalahan. Setelah menguji perangkat lunak menggunakan set pengujian lengkapnya, Anda hanya menemukan delapan puluh kesalahan yang diperkenalkan. Anda juga menemukan lima belas kesalahan lainnya. Perkirakan jumlah kesalahan laten dalam perangkat lunak. Apa keterbatasan metode penyemaian kesalahan?
47. Apa itu tes stres? Mengapa pengujian stres hanya berlaku untuk jenis sistem tertentu?
48. Apa yang Anda pahami dengan pengujian unit? Tulis kode untuk modul yang berisi fungsi untuk mengimplementasikan fungsionalitas tumpukan terbatas yang terdiri dari ratusan bilangan bulat. Elemen antrian dimuat dari dan disimpan ke dalam sistem database Oracle—Asumsikan bahwa tumpukan mendukung operasi: push, pop, dan is-empty. Rancang kasus uji unit untuk modul.
49. Apa yang Anda pahami dengan istilah pengujian integrasi? Jenis cacat apa yang ditemukan selama pengujian integrasi? Apa saja jenis metode pengujian integrasi yang dapat digunakan untuk melakukan pengujian integrasi produk perangkat lunak besar? Bandingkan kelebihan dan kekurangan dari berbagai strategi pengujian integrasi ini.
50. Diskusikan bagaimana Anda akan melakukan pengujian sistem perangkat lunak yang mengimplementasikan antrian terbatas elemen integral positif. Asumsikan bahwa

antrian hanya mendukung fungsi menyisipkan elemen, menghapus elemen, dan menemukan elemen.

51. Apa yang Anda pahami dengan efek samping dari panggilan fungsi? Berikan satu contoh efek samping. Mengapa efek samping yang tidak jelas tidak diinginkan?
52. Apakah yang Anda maksud: uji regresi Kapan pengujian regresi dilakukan? Mengapa pengujian regresi diperlukan? Bagaimana kasus uji regresi dirancang? Bagaimana pengujian regresi dilakukan?
53. Apakah Anda setuju dengan pernyataan berikut—"Pengujian sistem dapat dianggap sebagai pengujian kotak hitam murni." Justifikasi jawaban Anda.
54. Apa yang Anda pahami dengan pengujian integrasi big-bang? Bagaimana pengujian integrasi bigbang dilakukan? Apa keuntungan dan kerugian dari strategi pengujian integrasi big-bang? Jelaskan setidaknya satu situasi di mana pengujian integrasi big-bang diinginkan.
55. Apa hubungan antara kompleksitas siklomatik dan pemahaman program? Bisakah Anda membenarkan mengapa ada hubungan yang jelas seperti itu?
56. Pertimbangkan fungsi C berikut bernama bin-search:

```
/* num is the number the function searches in a presorted integer array
arr */
        int bin_search(int num){
        int min,max;
        min =0;
        max =100;
        while(min!=max){
                if(arr[(min+max)/2]>num)
                max=(min+max)/2;
                else if(arr[(min+max)/2]<num)
                min=(min+max)/2;
        else return((min+max)/2);
        } return(-1);
}
```

Rancang rangkaian pengujian untuk fungsi bin-search yang memenuhi strategi pengujian kotak putih berikut (Tunjukkan langkah-langkah perantara dalam menurunkan kasus uji):

   a. Cakupan pernyataan
   b. Cakupan cabang
   c. Cakupan kondisi
   d. Cakupan jalur

57. Perhatikan fungsi C berikut bernama sort.

```
/* sort takes an integer array and sorts it in ascending
order */
void sort(int a[], int n){
        int i,j;
        for(i=0;i<n-1;i++)
                for(j=i+1;j<n;j++)
                        if(a[i]>a[j])
                        {
                                temp=a[i];
                                a[i]=a[j];
                                a[j]=temp;
                        }
```

}

a. Tentukan kompleksitas siklomatik dari fungsi sortir.
b. Rancang rangkaian pengujian untuk jenis fungsi yang memenuhi strategi pengujian kotak putih berikut (Tunjukkan langkah-langkah penting dalam metode desain rangkaian pengujian Anda).
    i. Cakupan pernyataan
    ii. Cakupan cabang
    iii. Cakupan kondisi
    iv. Cakupan jalur

58. Gambarkan grafik aliran kontrol untuk fungsi berikut bernama findmaximum. Dari grafik aliran kontrol, tentukan kompleksitas siklomatiknya.

```
int find-maximum(int i,int j, int k){
        int max;
        if(i>j) then
                if(i>k) then max=i;
                        else max=k;
                else if(j>k) max=j
                else max=k;
        return(max);
}
```

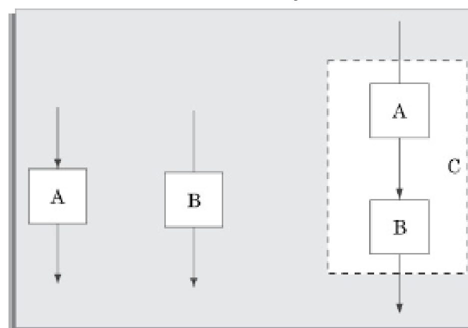
59. Misalkan sebuah program C memiliki 240 jenis pernyataan urutan, 50 jenis pernyataan pilihan dan 40 jenis pernyataan iterasi, tentukan jumlah minimum kasus uji yang diperlukan untuk pengujian jalur.
60. Hitung indeks Kabut dari pertanyaan ini. Apa yang dimaksud dengan indeks Kabut? Bagaimana indeks Fog berguna dalam menghasilkan dokumentasi perangkat lunak yang baik?
61. Identifikasi jenis cacat yang dapat Anda deteksi selama berikut ini:
    a. Inspeksi kode
    b. Panduan kode
62. Rancang rangkaian pengujian kotak hitam untuk fungsi bernama quadraticsolver. Fungsi kuadrat-solver menerima tiga bilangan floating point (a, b, c) yang mewakili persamaan kuadrat dalam bentuk ax2 + bx + c = 0. Fungsi ini menghitung dan menampilkan solusinya.
63. Rancang rangkaian pengujian kotak hitam untuk fungsi yang menerima empat pasang angka titik mengambang yang mewakili empat titik koordinat. Keempat titik koordinat ini mewakili pusat dua lingkaran dan titik pada keliling masing-masing dua lingkaran. Fungsi ini mencetak apakah dua lingkaran berpotongan, satu berada di dalam lingkaran lainnya, atau terputus.
64. Rancang rangkaian pengujian kotak hitam untuk fungsi yang disebut temukan-persimpangan. Fungsi menemukan-persimpangan mengambil empat bilangan real m1, c1, m2, c2 sebagai argumennya yang mewakili dua garis lurus y = m1x + c1 dan y = m2x + c2. Menentukan titik potong kedua garis. Bergantung pada nilai input ke fungsi, ini akan menampilkan salah satu dari pesan berikut:
    a. titik persimpangan tunggal
    b. garis yang tumpang tindih—titik perpotongan tak terhingga
    c. garis sejajar—tidak ada titik potong
    d. nilai masukan tidak valid
65. Rancang rangkaian uji kotak hitam untuk program berikut. Program menerima dua pasang koordinat (x1,y1),(x2,y2), (x3,y3), (x4,y4). Dua titik pertama (x1,y1) dan (x2,y2)

mewakili titik kiri bawah dan kanan atas persegi panjang pertama. Dua titik kedua (x3,y3) dan (x4,y4) mewakili titik kiri bawah dan kanan atas persegi panjang kedua. Diasumsikan bahwa panjang dan lebar persegi panjang sejajar dengan sumbu x atau sumbu y. Program menghitung titik potong kedua persegi panjang dan mencetak titik potongnya.

66. Rancang rangkaian uji kotak hitam untuk program yang menerima hingga sepuluh persamaan linier simultan dalam hingga sepuluh variabel independen dan menampilkan solusinya.
67. Rancang rangkaian pengujian kotak hitam untuk Perangkat Lunak Otomatisasi Perpustakaan berikut. Perangkat Lunak Otomasi Perpustakaan menerima string yang mewakili nama buku. Ini memeriksa katalog perpustakaan, dan menampilkan apakah buku tersebut terdaftar dalam katalog atau tidak. Jika buku terdaftar dalam katalog, ini akan menampilkan jumlah salinan yang saat ini tersedia di rak dan salinan yang dikeluarkan.
68. Rancang rangkaian pengujian kotak hitam untuk program yang menerima dua string dan memeriksa apakah string pertama adalah substring dari string kedua dan menampilkan berapa kali string pertama muncul di string kedua. Asumsikan bahwa masing-masing dari dua string memiliki ukuran kurang dari dua puluh karakter.
69. Rancang rangkaian uji kotak hitam untuk program yang menerima sepasang titik yang menentukan garis lurus dan titik lain serta angka mengambang yang menentukan pusat lingkaran dan jari-jarinya. Program ini dimaksudkan untuk menghitung titik perpotongan mereka dan mencetaknya.
70. Apa yang Anda pahami dengan bahasa spesifikasi yang dapat dieksekusi? Apa bedanya dengan bahasa pemrograman prosedural tradisional? Beri nama bahasa spesifikasi yang dapat dieksekusi.
71. Di antara fase pengembangan yang berbeda dari siklus hidup, pengujian biasanya membutuhkan tenaga kerja terbesar. Identifikasi alasan utama di balik kebutuhan tenaga kerja yang besar untuk fase pengujian.
72. Apa yang Anda pahami dengan pengujian kinerja? Apa jenis pengujian kinerja yang berbeda yang harus dilakukan untuk setiap masalah yang diuraikan dalam pertanyaan sepuluh-dua puluh dari Bab 6?
73. Identifikasi jenis informasi yang harus disajikan dalam laporan ringkasan pengujian.
74. Apa perbedaan antara pendekatan pengujian integrasi top-down dan bottom-up? Apa kelebihan dan kekurangan mereka? Jelaskan jawaban Anda dengan menggunakan contoh. Mengapa pendekatan pengujian integrasi campuran lebih disukai oleh banyak penguji?
75. Apa yang Anda pahami dengan “efektivitas tinjauan kode”? Bagaimana meninjau efektivitas untuk suatu organisasi diukur secara kuantitatif?
76. Apa yang Anda pahami dengan kompleksitas siklomatik suatu program? Bagaimana itu bisa diukur? Apa aplikasinya dalam pengembangan program?
    a. Apa yang Anda pahami dengan analisis program statis dan dinamis? Bagaimana hasil analisis program statis dan dinamis berguna?
    b. Apa perbedaan karakteristik program yang dilaporkan oleh
        i. alat analisis statis?
        ii. alat analisis dinamis?
    c. Tulis algoritme untuk penganalisis program dinamis yang akan menghitung dan melaporkan persentase jalur independen linier yang dicakup oleh rangkaian

pengujian. Jelaskan algoritma Anda. Apa kompleksitas komputasi dari algoritma Anda?

77. Apa saja pendekatan berbeda untuk pengujian integrasi? Pendekatan mana yang paling disukai untuk sistem perangkat lunak besar? Mengapa?
78. Apa saja jenis kesalahan yang ingin dideteksi oleh pengujian integrasi? Berikan dua contoh kesalahan tersebut.
79. Apa perbedaan antara pengujian integrasi bertahap dan inkremental? Bandingkan keuntungan dan kerugian dari dua pendekatan ini untuk pengujian integrasi.
80. Jelaskan perbedaan antara pengujian di besar dan pengujian di kecil. Apa tujuan masing-masing?
81. Jelaskan hal-hal penting di mana pengujian program prosedural dan berorientasi objek berbeda. Apakah berbagai fitur berorientasi objek mempermudah pengujian program berorientasi objek? Buktikan jawaban Anda dengan contoh yang sesuai.
82. Bisakah program berorientasi objek diuji dengan menguji setiap metode, kemudian mengintegrasikan metode, dan akhirnya melakukan pengujian sistem? Jika jawaban Anda adalah "ya", jelaskan bagaimana metode individual dapat diuji. Jika jawaban Anda "tidak", jelaskan mengapa tidak?
83. Apa implikasi dari pewarisan, polimorfisme, dan fitur enkapsulasi dari program berorientasi objek dalam pengujian program yang memuaskan?
84. Apa yang Anda pahami dengan pengujian kotak abu-abu? Mengapa pengujian kotak abu-abu dianggap penting untuk menguji program berorientasi objek?
85. Apa tingkat yang berbeda dari pengujian program berorientasi objek? Apa unit yang cocok untuk menguji program berorientasi objek?
86. Nyatakan aksioma antikomposisi Weyukar. Berikan pembenaran intuitif untuk hal yang sama.
87. Bagaimana pengujian integrasi program berorientasi objek dilakukan Jelaskan strategi pengujian integrasi yang berbeda untuk program berorientasi objek.
88. Apa itu pengujian alfa, beta, dan penerimaan? Apa perbedaan di antara berbagai jenis pengujian produk perangkat lunak ini? Jelaskan jawaban Anda tentang siapa yang melakukan tes, kapan tes dilakukan, dan tujuan tes.



Gambar 10.8 Kode segmen C diperoleh dengan menyandingkan kode segmen A dan B.

89. Apa yang Anda pahami dengan pengujian kinerja produk perangkat lunak? Kapan itu dilakukan? Apa tujuan dari pengujian kinerja? Apa saja jenis-jenis pengujian kinerja?
90. Kapan persyaratan non-fungsional diuji dalam siklus hidup produk perangkat lunak? Bagaimana persyaratan non-fungsional yang berbeda diuji? Jelaskan jawaban Anda sehubungan dengan berbagai kategori persyaratan non-fungsional.
91. Apa yang Anda pahami dengan analisis cakupan tes? Apa kegunaan dari analisis cakupan tes? Tentukan setidaknya dua metrik cakupan pengujian.

92. Apa yang Anda pahami dengan debugger simbolis? Bagaimana debugging dilakukan oleh debugger simbolis? Apa teknik populer lainnya untuk debugging?
93. Apa yang Anda pahami dengan pengujian aliran data? Bagaimana pengujian aliran data dilakukan? Apakah mungkin untuk merancang kasus uji aliran data secara manual? Jelaskan jawaban Anda.
94. Apa perbedaan antara pengujian kotak hitam dan kotak putih? Selama pengujian unit, dapatkah pengujian kotak hitam dilewati, jika seseorang berencana untuk melakukan pengujian kotak putih menyeluruh? Justifikasi jawaban Anda.
95. Bedakan antara analisis statis dan dinamis dari suatu program. Jelaskan setidaknya satu metrik yang dilaporkan oleh alat analisis statis dan setidaknya satu metrik yang dilaporkan oleh alat analisis dinamis. Bagaimana metrik ini berguna?
96. Misalkan kompleksitas siklomatik dari segmen kode A dan B (ditunjukkan pada Gambar 10.8) masing-masing adalah m dan n. Berapakah kompleksitas siklomatik dari segmen kode C yang diperoleh dengan menyandingkan segmen kode A dan B?

BAB 11
REABILITAS SSOFTWARE DAN MANAJEMEN KUALITAS

Keandalan produk perangkat lunak merupakan perhatian penting bagi sebagian besar pengguna. Pengguna tidak hanya menginginkan produk yang mereka beli sangat andal, tetapi untuk kategori produk tertentu mereka bahkan mungkin memerlukan jaminan kuantitatif atas keandalan produk sebelum membuat keputusan pembelian. Hal ini terutama berlaku untuk produk perangkat lunak tertanam dan kritis terhadap keselamatan. Namun, seperti yang kita bahas dalam Bab ini, sangat sulit untuk mengukur keandalan produk perangkat lunak secara akurat. Salah satu masalah utama yang dihadapi saat mengukur keandalan produk perangkat lunak secara kuantitatif adalah kenyataan bahwa keandalan bergantung pada pengamat. Artinya, kelompok pengguna yang berbeda mungkin sampai pada perkiraan keandalan yang berbeda untuk produk yang sama. Selain itu, beberapa masalah lain (seperti nilai keandalan yang sering berubah karena koreksi bug) membuat pengukuran keandalan produk perangkat lunak menjadi sulit. Meskipun tidak ada metrik yang sepenuhnya memuaskan untuk mengukur keandalan produk perangkat lunak, kita akan membahas beberapa metrik yang digunakan saat ini untuk mengukur keandalan produk perangkat lunak. Kita juga akan membahas masalah pemodelan pertumbuhan keandalan dan memeriksa bagaimana memprediksi kapan (dan jika sama sekali) tingkat keandalan tertentu akan tercapai. Kita juga akan memeriksa pendekatan pengujian statistik untuk estimasi reliabilitas.

Dalam bab ini, selain masalah keandalan perangkat lunak, Kita juga akan membahas berbagai masalah yang terkait dengan jaminan kualitas perangkat lunak (SQA). Jaminan kualitas perangkat lunak (SQA) telah muncul sebagai salah satu topik yang paling banyak dibicarakan dalam beberapa tahun terakhir di kalangan industri perangkat lunak. Tujuan utama SQA adalah membantu organisasi mengembangkan produk perangkat lunak berkualitas tinggi secara berulang. Organisasi pengembangan perangkat lunak dapat disebut dapat diulang jika proses pengembangan perangkat lunaknya tidak bergantung pada orang. Artinya, keberhasilan suatu proyek tidak tergantung pada siapa sebenarnya anggota tim proyek tersebut. Selain itu, kualitas perangkat lunak yang dikembangkan dan biaya pengembangan merupakan isu penting yang ditangani oleh SQA. Dalam bab ini, pertama-tama kita membahas beberapa masalah penting mengenai pengukuran dan prediksi keandalan perangkat lunak sebelum memulai diskusi tentang jaminan kualitas perangkat lunak.

11.1 REABILITAS SOFTWARE

Keandalan produk perangkat lunak pada dasarnya menunjukkan kepercayaan atau ketergantungannya. Atau, keandalan produk perangkat lunak juga dapat didefinisikan sebagai kemungkinan produk bekerja "dengan benar" selama periode waktu tertentu. Secara intuitif, jelas bahwa produk perangkat lunak yang memiliki banyak cacat tidak dapat diandalkan. Juga sangat masuk akal untuk mengasumsikan bahwa keandalan suatu sistem meningkat, karena jumlah cacat di dalamnya berkurang. Akan sangat bagus jika kita dapat secara matematis mengkarakterisasi hubungan antara keandalan dan jumlah bug yang ada dalam sistem menggunakan ekspresi bentuk tertutup sederhana. Sayangnya, sangat sulit untuk mengkarakterisasi keandalan yang diamati dari suatu sistem dalam hal jumlah cacat laten dalam sistem menggunakan ekspresi matematika sederhana.

Untuk mendapatkan wawasan tentang masalah ini, pertimbangkan hal berikut. Menghapus kesalahan dari bagian-bagian produk perangkat lunak yang sangat jarang dieksekusi, membuat sedikit perbedaan pada keandalan produk yang dirasakan. Secara

eksperimental menganalisis perilaku sejumlah besar program yang 90 persen dari waktu eksekusi program khas dihabiskan dalam mengeksekusi hanya 10 persen dari instruksi dalam program. Instruksi 10 persen yang paling sering digunakan sering disebut inti1 dari suatu program. Sisanya 90 persen dari pernyataan program disebut non-inti dan rata-rata dieksekusi hanya untuk 10 persen dari total waktu eksekusi. Oleh karena itu, mungkin tidak terlalu mengejutkan untuk dicatat bahwa menghilangkan 60 persen cacat produk dari bagian-bagian sistem yang paling jarang digunakan biasanya hanya menghasilkan 3 persen peningkatan keandalan produk. Jelas bahwa jumlah keseluruhan keandalan program meningkat karena koreksi kesalahan tunggal tergantung pada seberapa sering instruksi yang memiliki kesalahan dijalankan. Jika kesalahan dihapus dari instruksi yang sering dieksekusi (yaitu, milik inti program), maka ini akan muncul sebagai peningkatan besar pada angka keandalan. Di sisi lain, menghapus kesalahan dari bagian program yang jarang digunakan, mungkin tidak menyebabkan perubahan yang berarti pada keandalan produk.

Berdasarkan pembahasan di atas kita dapat mengatakan bahwa keandalan suatu produk tidak hanya bergantung pada jumlah kesalahan laten tetapi juga pada lokasi kesalahan yang tepat. Selain itu, keandalan juga tergantung pada bagaimana produk digunakan, atau pada profil eksekusinya. Jika pengguna hanya menjalankan fitur-fitur program yang "diimplementasikan dengan benar", tidak ada kesalahan yang akan terungkap dan keandalan produk yang dirasakan akan tinggi. Di sisi lain, jika hanya fungsi perangkat lunak yang mengandung kesalahan yang dipanggil, maka sejumlah besar kegagalan akan diamati dan keandalan sistem yang dirasakan akan sangat rendah. Kategori yang berbeda dari pengguna produk perangkat lunak biasanya menjalankan fungsi yang berbeda dari produk perangkat lunak. Misalnya, untuk Perangkat Lunak Otomasi Perpustakaan, anggota perpustakaan akan menggunakan fungsionalitas seperti buku terbitan, buku pencarian, dll., Di sisi lain, pustakawan biasanya akan menjalankan fitur-fitur seperti buat anggota, buat catatan buku, hapus catatan anggota, dll. Jadi cacat yang muncul untuk pustakawan, mungkin tidak muncul untuk anggota. Misalkan fungsi Perangkat Lunak Otomasi Perpustakaan yang digunakan anggota perpustakaan bebas dari kesalahan; dan fungsi yang digunakan Pustakawan memiliki banyak bug. Kemudian, kedua kategori pengguna ini akan memiliki pendapat yang sangat berbeda tentang keandalan perangkat lunak. Oleh karena itu, Berdasarkan pembahasan di atas, alasan utama yang membuat keandalan perangkat lunak lebih sulit diukur daripada keandalan perangkat keras:

- Peningkatan keandalan karena memperbaiki satu bug tergantung pada lokasi bug dalam kode.
- Keandalan yang dirasakan dari produk perangkat lunak bergantung pada pengamat.
- Keandalan produk terus berubah saat kesalahan terdeteksi dan diperbaiki.

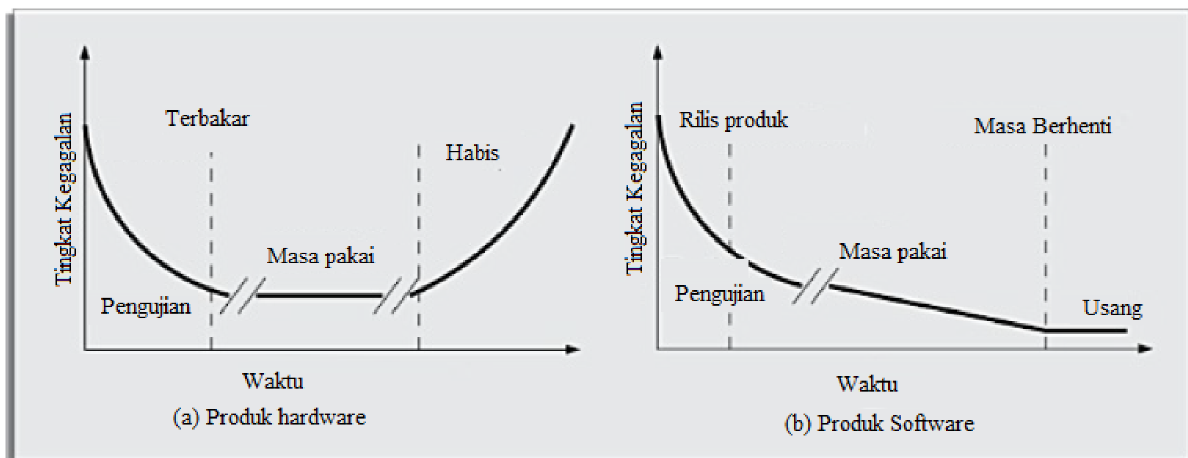
Keandalan Perangkat Keras versus Perangkat Lunak

Fitur karakteristik penting yang membedakan masalah keandalan perangkat keras dan perangkat lunak adalah perbedaan antara pola kegagalannya. Komponen perangkat keras gagal karena alasan yang sangat berbeda dibandingkan dengan komponen perangkat lunak. Komponen perangkat keras gagal sebagian besar karena keausan, sedangkan komponen perangkat lunak gagal karena bug.

Gerbang logika mungkin macet pada 1 atau 0, atau resistor mungkin mengalami korsleting. Untuk memperbaiki kesalahan perangkat keras, seseorang harus mengganti atau memperbaiki bagian yang gagal. Sebaliknya, produk perangkat lunak akan terus gagal sampai kesalahan dilacak dan desain atau kode diubah untuk memperbaiki bug. Untuk alasan ini, ketika bagian perangkat keras diperbaiki, keandalannya akan dipertahankan pada tingkat yang

ada sebelum kegagalan terjadi; sedangkan ketika kegagalan perangkat lunak diperbaiki, keandalan dapat meningkat atau menurun (keandalan dapat menurun jika perbaikan bug menimbulkan kesalahan baru). Untuk menempatkan fakta ini dalam perspektif yang berbeda, studi keandalan perangkat keras berkaitan dengan stabilitas (misalnya, waktu antar-kegagalan tetap konstan). Di sisi lain, tujuan dari studi keandalan perangkat lunak adalah pertumbuhan keandalan (yaitu, peningkatan waktu antar-kegagalan).

Perbandingan perubahan tingkat kegagalan selama masa hidup produk untuk produk perangkat keras yang khas serta produk perangkat lunak digambarkan pada Gambar 11.1. Amati bahwa plot perubahan keandalan dengan waktu untuk komponen perangkat keras (Gambar 11.1(a)) tampak seperti "bak mandi". Untuk komponen perangkat lunak, tingkat kegagalan awalnya tinggi, tetapi menurun karena komponen yang rusak diidentifikasi diperbaiki atau diganti. Sistem kemudian memasuki masa manfaatnya, di mana tingkat kegagalan hampir konstan. Setelah beberapa waktu (disebut waktu hidup produk) komponen utama aus, dan tingkat kegagalan meningkat. Kegagalan awal biasanya ditanggung melalui garansi pabrik. Akibat wajar dari pengamatan ini (walaupun menyimpang dari topik diskusi kita) adalah bahwa mungkin tidak bijaksana untuk membeli suatu produk (bahkan dengan diskon yang bagus untuk nilai nominalnya) menjelang akhir masa pakainya, Artinya, seseorang tidak perlu merasa senang untuk membeli mobil berusia sepuluh tahun dengan sepersepuluh dari harga mobil baru, karena akan berada di dekat tepi naik kurva bak mandi, dan seseorang harus menghabiskan waktu, tenaga, dan uang yang terlalu besar untuk memperbaiki dan berakhir sebagai pecundang. Berbeda dengan produk perangkat keras, produk perangkat lunak menunjukkan tingkat kegagalan tertinggi setelah pembelian dan pemasangan (lihat bagian awal plot pada Gambar 11.1 (b)). Saat sistem digunakan, semakin banyak kesalahan yang diidentifikasi dan dihapus sehingga mengurangi tingkat kegagalan. Penghapusan kesalahan ini berlanjut pada kecepatan yang lebih lambat selama masa manfaat produk. Karena perangkat lunak menjadi usang, tidak ada lagi koreksi kesalahan yang terjadi dan tingkat kegagalan tetap tidak berubah.



Gambar 11.1 Perubahan tingkat kegagalan suatu produk.

Metrik Keandalan Produk Perangkat Lunak

Persyaratan keandalan untuk berbagai kategori produk perangkat lunak mungkin berbeda. Untuk alasan ini, tingkat keandalan yang diperlukan untuk produk perangkat lunak harus ditentukan dalam dokumen spesifikasi persyaratan perangkat lunak (SRS). Untuk dapat melakukan ini, kita memerlukan beberapa metrik untuk secara kuantitatif mengekspresikan keandalan produk perangkat lunak. Ukuran keandalan yang baik harus independen dari pengamat, sehingga orang yang berbeda dapat menyepakati tingkat keandalan yang dimiliki

suatu sistem. Namun, dalam praktiknya, sangat sulit untuk merumuskan metrik yang menggunakan pengukuran reliabilitas yang tepat. Dengan tidak adanya ukuran tersebut, Kita membahas enam metrik yang berkorelasi dengan keandalan sebagai berikut:

Tingkat terjadinya kegagalan (ROCOF): ROCOF mengukur frekuensi terjadinya kegagalan. Ukuran ROCOF produk perangkat lunak dapat diperoleh dengan mengamati perilaku produk perangkat lunak dalam operasi selama interval waktu tertentu dan kemudian menghitung nilai ROCOF sebagai rasio jumlah kegagalan yang diamati dan durasi pengamatan. Namun, banyak produk perangkat lunak tidak berjalan terus menerus (tidak seperti mobil atau mixer), tetapi memberikan layanan tertentu ketika ada permintaan. Misalnya, perangkat lunak perpustakaan tidak digunakan sampai permintaan penerbitan buku dibuat. Oleh karena itu, untuk produk perangkat lunak biasa seperti perangkat lunak penggajian, penerapan ROCOF sangat terbatas.

Mean time to failure (MTTF): MTTF adalah waktu antara dua kegagalan berturut-turut, rata-rata untuk sejumlah besar kegagalan. Untuk mengukur MTTF, kita dapat merekam data kegagalan untuk n kegagalan. Biarkan kegagalan terjadi pada saat t1, t2, ..., tn. Kemudian, MTTF dapat dihitung sebagai

$$\sum_{i=1}^{n} \frac{t_i + 1 - t_i}{(n - 1_}$$

Penting untuk dicatat bahwa hanya waktu berjalan yang dipertimbangkan dalam pengukuran waktu. Artinya, waktu sistem mati untuk memperbaiki kesalahan, waktu boot, dll. tidak diperhitungkan dalam pengukuran waktu dan jam dihentikan pada waktu ini.

Mean Time to Repair (MTTR): Setelah terjadi kegagalan, beberapa waktu diperlukan untuk memperbaiki kesalahan. MTTR mengukur waktu rata-rata yang diperlukan untuk melacak kesalahan yang menyebabkan kegagalan dan memperbaikinya.

Mean Time Between Failure (MTBF): Metrik MTTF dan MTTR dapat digabungkan untuk mendapatkan metrik MTBF: MTBF=MTTF+MTTR. Jadi, MTBF 300 jam menunjukkan bahwa sekali kegagalan terjadi, kegagalan berikutnya diharapkan setelah 300 jam. Dalam hal ini, pengukuran waktu adalah waktu nyata dan bukan waktu eksekusi seperti pada MTTF.

Probabilitas kegagalan sesuai permintaan/*Probability of failure on demand* (POFOD): Tidak seperti metrik lain yang dibahas, metrik ini tidak secara eksplisit melibatkan pengukuran waktu. POFOD mengukur kemungkinan sistem gagal ketika permintaan layanan dibuat. Misalnya, POFOD 0,001 berarti bahwa 1 dari setiap 1000 permintaan layanan akan mengakibatkan kegagalan. Keandalan produk perangkat lunak harus ditentukan melalui pemanggilan layanan tertentu, daripada membuat perangkat lunak terus berjalan. Dengan demikian, metrik POFOD sangat sesuai untuk produk perangkat lunak yang tidak diharuskan berjalan terus menerus.

Ketersediaan: Ketersediaan sistem adalah ukuran seberapa besar kemungkinan sistem akan tersedia untuk digunakan selama periode waktu tertentu. Metrik ini tidak hanya mempertimbangkan jumlah kegagalan yang terjadi selama interval waktu, tetapi juga memperhitungkan waktu perbaikan (down time) suatu sistem ketika terjadi kegagalan. Metrik ini penting untuk sistem seperti sistem telekomunikasi, dan sistem operasi, dan pengontrol tertanam, dll. yang seharusnya tidak pernah down dan di mana waktu perbaikan dan restart signifikan dan hilangnya layanan selama waktu itu tidak dapat diabaikan.

Kekurangan metrik keandalan produk perangkat lunak

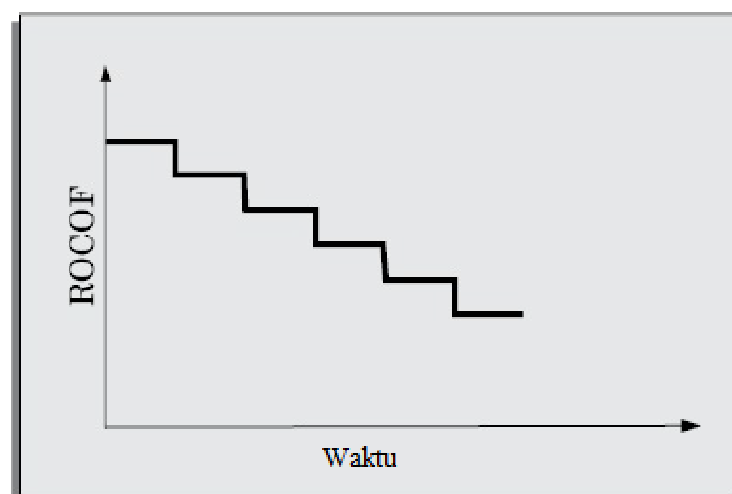
Semua metrik keandalan di atas memiliki beberapa kekurangan sejauh penggunaannya dalam pengukuran keandalan perangkat lunak. Salah satu alasannya adalah bahwa metrik ini dipusatkan di sekitar kemungkinan terjadinya kegagalan sistem tetapi tidak memperhitungkan konsekuensi kegagalan. Artinya, model keandalan ini tidak membedakan

tingkat keparahan relatif dari kegagalan yang berbeda. Kegagalan yang bersifat sementara dan yang konsekuensinya tidak serius dalam praktiknya tidak terlalu diperhatikan dalam penggunaan operasional produk perangkat lunak. Jenis kegagalan ini paling-paling bisa menjadi iritasi kecil. Di sisi lain, jenis kegagalan yang lebih parah dapat membuat sistem sama sekali tidak dapat digunakan. Untuk memperkirakan keandalan produk perangkat lunak secara lebih akurat, perlu untuk mengklasifikasikan berbagai jenis kegagalan. Harap dicatat bahwa kelas kegagalan yang berbeda mungkin tidak saling eksklusif. Klasifikasi ini didasarkan pada serangkaian kriteria yang sangat berbeda. Akibatnya, jenis kegagalan dapat pada saat yang sama milik lebih dari satu kelas. Skema klasifikasi kegagalan adalah sebagai berikut:

- **Transien**: Kegagalan sementara hanya terjadi untuk nilai input tertentu saat menjalankan fungsi sistem.
- **Permanen**: Kegagalan permanen terjadi untuk semua nilai input saat menjalankan fungsi sistem.
- **Dapat dipulihkan**: Ketika terjadi kegagalan yang dapat dipulihkan, sistem dapat pulih tanpa harus mematikan dan memulai ulang sistem (dengan atau tanpa campur tangan operator).
- **Tidak dapat dipulihkan**: Dalam kegagalan yang tidak dapat dipulihkan, sistem mungkin perlu dimulai ulang.
- **Kosmetik**: Kelas kegagalan ini hanya menyebabkan iritasi ringan, dan tidak menyebabkan hasil yang salah. Contoh kegagalan kosmetik adalah situasi di mana tombol mouse harus diklik dua kali, bukan sekali untuk menjalankan fungsi tertentu melalui antarmuka pengguna grafis.

Pemodelan Pertumbuhan Keandalan

Model pertumbuhan keandalan adalah model matematis tentang bagaimana keandalan perangkat lunak meningkat ketika kesalahan terdeteksi dan diperbaiki. Model pertumbuhan keandalan dapat digunakan untuk memprediksi kapan (atau jika sama sekali) tingkat keandalan tertentu kemungkinan akan dicapai. Dengan demikian, pemodelan pertumbuhan keandalan dapat digunakan untuk menentukan kapan harus menghentikan pengujian untuk mencapai tingkat keandalan yang diberikan. Meskipun beberapa model pertumbuhan keandalan yang berbeda telah diusulkan, dalam teks ini kita hanya akan membahas dua model pertumbuhan keandalan yang sangat sederhana.



Gambar 11.2 Model fungsi langkah pertumbuhan keandalan.

Model Jelinski dan Moranda

Model pertumbuhan keandalan yang paling sederhana adalah model fungsi langkah di mana diasumsikan bahwa keandalan meningkat dengan kenaikan konstan setiap kali kesalahan terdeteksi dan diperbaiki. Model seperti itu ditunjukkan pada Gambar 11.2. Namun, model keandalan sederhana ini yang secara implisit mengasumsikan bahwa semua kesalahan berkontribusi sama terhadap pertumbuhan keandalan, sangat tidak realistis karena kita sudah tahu bahwa koreksi kesalahan yang berbeda berkontribusi secara berbeda terhadap pertumbuhan keandalan.

Model Littlewood dan Verall

Model ini memungkinkan pertumbuhan kehandalan negatif untuk mencerminkan fakta bahwa ketika perbaikan dilakukan, mungkin menimbulkan kesalahan tambahan. Ini juga memodelkan fakta bahwa ketika kesalahan diperbaiki, peningkatan rata-rata keandalan produk per perbaikan menurun. Ini memperlakukan kontribusi kesalahan untuk peningkatan keandalan menjadi variabel acak independen yang memiliki distribusi Gamma. Distribusi ini memodelkan fakta bahwa koreksi kesalahan dengan kontribusi besar terhadap pertumbuhan keandalan dihilangkan terlebih dahulu. Ini menunjukkan hasil yang semakin berkurang saat pengujian berlanjut. Ada model pertumbuhan keandalan yang lebih kompleks, yang memberikan perkiraan yang lebih akurat untuk pertumbuhan keandalan. Namun, model-model ini berada di luar cakupan teks ini.

11.2 PENGUJIAN STATISTIK

Pengujian statistik adalah proses pengujian yang tujuannya adalah untuk menentukan keandalan produk daripada menemukan kesalahan. Kasus uji dirancang untuk pengujian statistik dengan tujuan yang sama sekali berbeda dari pengujian konvensional. Untuk melakukan pengujian statistik, pertama-tama kita perlu menentukan profil operasi produk.

Profil operasi: Kategori pengguna yang berbeda dapat menggunakan produk perangkat lunak untuk tujuan yang sangat berbeda. Misalnya, pustakawan mungkin menggunakan Perangkat Lunak Otomasi Perpustakaan untuk membuat catatan anggota, menghapus catatan anggota, menambahkan buku ke perpustakaan, dll., sedangkan anggota perpustakaan mungkin menggunakan perangkat lunak untuk menanyakan tentang ketersediaan buku, dan untuk menerbitkan dan mengembalikan buku. Secara formal, kita dapat mendefinisikan profil operasi perangkat lunak sebagai kemungkinan pengguna memilih fungsionalitas perangkat lunak yang berbeda. Jika kita menyatakan himpunan berbagai fungsi yang ditawarkan oleh perangkat lunak dengan *{fi},* profil operasional akan mengasosiasikan dengan setiap fungsi *{fi}* dengan probabilitas rata-rata pengguna akan memilih *{fi}* sebagai fungsi berikutnya untuk digunakan. Dengan demikian, kita dapat menganggap profil operasi sebagai menetapkan nilai probabilitas *pi* untuk setiap fungsionalitas *fi* perangkat lunak.

Bagaimana cara menentukan profil operasi untuk suatu produk?

Kita perlu membagi data input menjadi beberapa kelas input. Misalnya, untuk perangkat lunak editor grafis, kita dapat membagi input menjadi data yang terkait dengan operasi edit, cetak, dan file, kemudian menetapkan nilai probabilitas untuk setiap kelas input; untuk menandakan probabilitas nilai input dari kelas itu untuk dipilih. Profil operasi suatu produk perangkat lunak dapat ditentukan dengan mengamati dan menganalisis pola penggunaan perangkat lunak oleh sejumlah pengguna.

Langkah-langkah dalam Pengujian Statistik

Langkah pertama adalah menentukan profil operasi perangkat lunak. Langkah selanjutnya adalah menghasilkan satu set data uji yang sesuai dengan profil operasi yang ditentukan. Langkah ketiga adalah menerapkan kasus uji ke perangkat lunak dan mencatat waktu antara setiap kegagalan. Setelah sejumlah kegagalan yang signifikan secara statistik

telah diamati, keandalan dapat dihitung. Untuk hasil yang akurat, pengujian statistik memerlukan beberapa asumsi dasar yang harus dipenuhi. Ini membutuhkan jumlah kasus uji yang signifikan secara statistik untuk digunakan. Lebih lanjut mensyaratkan bahwa persentase kecil dari input pengujian yang mungkin menyebabkan kegagalan sistem untuk dimasukkan. Sekarang mari kita bahas implikasi dari asumsi-asumsi ini. Sangat mudah untuk menghasilkan kasus uji untuk jenis input umum, karena seseorang dapat dengan mudah menulis program generator kasus uji yang secara otomatis dapat menghasilkan kasus uji ini. Namun, juga diharuskan bahwa persentase input yang tidak mungkin secara statistik signifikan juga harus dimasukkan dalam rangkaian pengujian. Membuat input yang tidak mungkin ini menggunakan generator test case sangat sulit.

Pro dan kontra dari pengujian statistik

Pengujian statistik memungkinkan seseorang untuk berkonsentrasi pada pengujian bagian dari sistem yang paling mungkin digunakan. Oleh karena itu, ini menghasilkan sistem yang menurut pengguna lebih andal (daripada yang sebenarnya!). Selain itu, estimasi reliabilitas yang diperoleh dengan menggunakan pengujian statistik lebih akurat dibandingkan dengan metode lain yang dibahas. Namun, tidak mudah untuk melakukan pengujian statistik secara memuaskan karena dua alasan berikut. Tidak ada cara sederhana dan berulang untuk mendefinisikan profil operasi. Juga, jumlah kasus uji dengan sistem yang akan diuji harus signifikan secara statistik.

11.3 KUALITAS PERANGKAT LUNAK

Secara tradisional, kualitas suatu produk didefinisikan dalam hal kesesuaian tujuannya. Artinya, produk berkualitas baik melakukan persis apa yang diinginkan pengguna, karena untuk hampir setiap produk, kesesuaian tujuan ditafsirkan dalam hal kepuasan persyaratan yang ditetapkan dalam dokumen SRS. Meskipun "kesesuaian tujuan" adalah definisi kualitas yang memuaskan untuk banyak produk seperti mobil, kipas meja, mesin gerinda, dll.— "kesesuaian tujuan" bukanlah definisi kualitas yang sepenuhnya memuaskan untuk produk perangkat lunak. Untuk memberikan contoh mengapa demikian, pertimbangkan produk perangkat lunak yang secara fungsional benar. Artinya, ia melakukan dengan benar semua fungsi yang telah ditentukan dalam dokumen SRS-nya. Meskipun secara fungsional mungkin benarbenar, ini tidak dapat dianggap sebagai produk berkualitas jika memiliki antarmuka pengguna yang tidak dapat digunakan. Contoh lain adalah produk yang melakukan semua yang diinginkan pengguna tetapi memiliki kode yang hampir tidak dapat dipahami dan tidak dapat dipelihara. Oleh karena itu, konsep tradisional kualitas sebagai "kesesuaian tujuan" untuk produk perangkat lunak tidak sepenuhnya memuaskan.

Tidak seperti produk perangkat keras, perangkat lunak bertahan lama, dalam arti ia terus berkembang untuk mengakomodasi keadaan yang berubah. Pandangan modern tentang kualitas mengaitkan dengan produk perangkat lunak beberapa faktor kualitas (atau atribut) seperti berikut ini:

Portabilitas: Sebuah produk perangkat lunak dikatakan portabel, jika dapat dengan mudah dibuat untuk bekerja di lingkungan perangkat keras dan sistem operasi yang berbeda, dan dengan mudah berinteraksi dengan perangkat keras eksternal dan produk perangkat lunak.

Kegunaan: Produk perangkat lunak memiliki kegunaan yang baik, jika kategori pengguna yang berbeda (yaitu, pengguna ahli dan pemula) dapat dengan mudah menjalankan fungsi produk.

Reusability: Sebuah produk perangkat lunak memiliki reusability yang baik, jika modul yang berbeda dari produk dapat dengan mudah digunakan kembali untuk mengembangkan produk baru.

Kebenaran: Produk perangkat lunak benar, jika persyaratan yang berbeda seperti yang ditentukan dalam dokumen SRS telah diterapkan dengan benar.

Pemeliharaan: Produk perangkat lunak dapat dipelihara, jika kesalahan dapat dengan mudah diperbaiki saat dan ketika muncul, fungsi baru dapat dengan mudah ditambahkan ke produk, dan fungsi produk dapat dengan mudah dimodifikasi, dll.

Faktor kualitas McCall

McCall membedakan dua tingkat atribut kualitas [McCall]. Atribut tingkat yang lebih tinggi, yang dikenal sebagai faktor kualitas atau atribut eksternal hanya dapat diukur secara tidak langsung. Atribut kualitas tingkat kedua disebut kriteria kualitas. Kriteria kualitas dapat diukur secara langsung, baik secara objektif maupun subjektif. Dengan menggabungkan peringkat beberapa kriteria, kita dapat memperoleh peringkat untuk faktor kualitas, atau sejauh mana mereka puas. Misalnya, keandalan tidak dapat diukur secara langsung, tetapi dengan mengukur jumlah cacat yang ditemui selama periode waktu tertentu. Dengan demikian, keandalan adalah faktor kualitas tingkat yang lebih tinggi dan jumlah cacat adalah faktor kualitas tingkat rendah.

ISO 9126

ISO 9126 mendefinisikan serangkaian karakteristik kualitas hierarkis. Setiap subkarakteristik dalam hal ini terkait dengan tepat satu karakteristik kualitas. Ini berbeda dengan atribut kualitas McCall yang sangat saling terkait. Perbedaan lainnya adalah bahwa karakteristik ISO secara ketat mengacu pada produk perangkat lunak, sedangkan atribut McCall juga menangkap masalah kualitas proses. Pengguna dan juga manajer cenderung tertarik pada atribut kualitas tingkat tinggi (faktor kualitas).

11.4 SISTEM MANAJEMEN KUALITAS PERANGKAT LUNAK

Sistem manajemen mutu (sering disebut sebagai sistem mutu) adalah metodologi utama yang digunakan oleh organisasi untuk memastikan bahwa produk yang mereka kembangkan memiliki kualitas yang diinginkan. Beberapa masalah penting yang terkait dengan sistem mutu adalah sebagai berikut:

Struktur manajerial dan tanggung jawab individu

Sistem mutu adalah tanggung jawab organisasi secara keseluruhan. Namun, setiap organisasi memiliki departemen mutu yang terpisah untuk melakukan beberapa aktivitas sistem mutu. Sistem mutu suatu organisasi harus mendapat dukungan penuh dari manajemen puncak. Tanpa dukungan untuk sistem mutu pada tingkat tinggi di sebuah perusahaan, beberapa anggota staf akan menganggap serius sistem mutu.

Kegiatan sistem mutu

Kegiatan sistem mutu meliputi:

- Audit proyek untuk memeriksa apakah proses sedang diikuti.
- Kumpulkan metrik proses dan produk dan analisis untuk memeriksa apakah sasaran kualitas terpenuhi.
- Review sistem mutu agar lebih efektif.
- Pengembangan standar, prosedur, dan pedoman.
- Menghasilkan laporan untuk manajemen puncak yang meringkas efektivitas sistem mutu dalam organisasi.

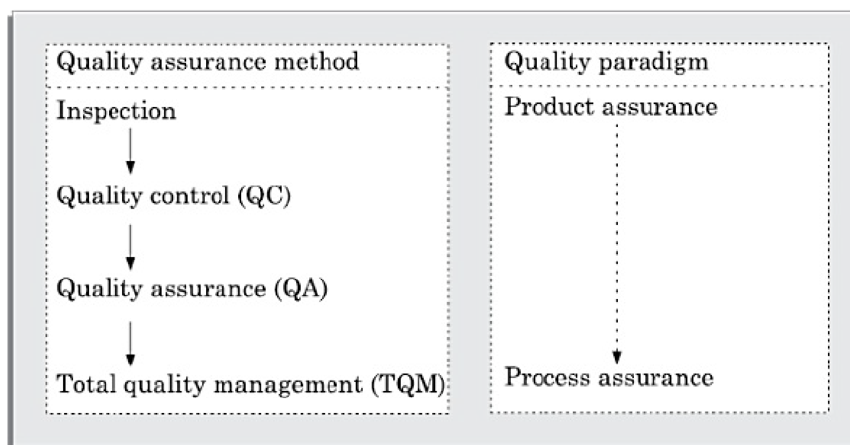
Sistem mutu yang baik harus didokumentasikan dengan baik. Tanpa sistem kualitas yang terdokumentasi dengan baik, penerapan kontrol kualitas dan prosedur menjadi ad hoc,

menghasilkan variasi besar dalam kualitas produk yang dikirimkan. Juga, sistem mutu yang tidak berdokumen mengirimkan pesan yang jelas kepada staf tentang sikap organisasi terhadap jaminan mutu. Standar internasional seperti ISO 9000 memberikan panduan tentang bagaimana mengatur sistem mutu.

Evolusi Sistem Kualitas

Sistem kualitas telah berkembang pesat selama enam dekade terakhir. Sebelum Perang Dunia II, metode yang biasa digunakan untuk menghasilkan produk berkualitas adalah dengan memeriksa produk jadi untuk menghilangkan produk yang cacat. Misalnya, sebuah perusahaan yang memproduksi mur dan baut akan memeriksa barang jadinya dan akan menolak mur dan baut yang berada di luar kisaran toleransi tertentu yang ditentukan.

Sejak saat itu, sistem mutu organisasi telah mengalami empat tahap evolusi seperti yang ditunjukkan pada Gambar 11.3. Metode inspeksi produk awal memberi jalan kepada prinsip-prinsip kontrol kualitas (QC). Quality control (QC) tidak hanya berfokus pada mendeteksi produk yang cacat dan menghilangkannya, tetapi juga menentukan penyebab di balik cacat tersebut, sehingga tingkat penolakan produk dapat dikurangi.



Gambar 11.3 Evolusi sistem mutu dan pergeseran yang sesuai dalam paradigma kualitas.

Dengan demikian, pengendalian kualitas bertujuan untuk memperbaiki penyebab kesalahan dan tidak hanya menolak produk yang cacat. Terobosan berikutnya dalam sistem mutu, adalah pengembangan prinsip jaminan mutu (QA). Premis dasar jaminan kualitas modern adalah bahwa jika proses organisasi baik dan diikuti dengan ketat, maka produknya pasti berkualitas baik.

Paradigma jaminan kualitas modern mencakup pedoman untuk mengenali, mendefinisikan, menganalisis, dan meningkatkan proses produksi. Total quality management (TQM) menganjurkan bahwa proses yang diikuti oleh suatu organisasi harus terus ditingkatkan melalui pengukuran proses. TQM melangkah lebih jauh dari jaminan kualitas dan bertujuan untuk perbaikan proses yang berkelanjutan. TQM melampaui mendokumentasikan proses untuk mengoptimalkannya melalui desain ulang. Istilah yang terkait dengan TQM adalah rekayasa ulang proses bisnis (BPR), yang bertujuan untuk merekayasa ulang cara bisnis dijalankan dalam suatu organisasi. Dari pembahasan di atas, kita dapat mengatakan bahwa selama sekitar enam dekade terakhir, paradigma kualitas telah bergeser dari jaminan produk ke jaminan proses (lihat Gambar 11.3).

Metrik Produk versus Metrik Proses

Semua sistem kualitas modern menekankan pada pengumpulan metrik produk dan proses tertentu selama pengembangan produk. Mari kita pahami dulu perbedaan mendasar antara metrik produk dan proses. Metrik produk membantu mengukur karakteristik produk

yang sedang dikembangkan, sedangkan metrik proses membantu mengukur kinerja suatu proses.

Contoh metrik produk adalah LOC dan titik fungsi untuk mengukur ukuran, PM (orang-bulan) untuk mengukur upaya yang diperlukan untuk mengembangkannya, bulan untuk mengukur waktu yang dibutuhkan untuk mengembangkan produk, kompleksitas waktu dari algoritma, dll. Contoh proses metrik adalah efektivitas tinjauan, jumlah rata-rata cacat yang ditemukan per jam inspeksi, waktu koreksi cacat rata-rata, produktivitas, jumlah rata-rata kegagalan yang terdeteksi selama pengujian per LOC, jumlah cacat laten per baris kode dalam produk yang dikembangkan.

ISO 9000

Organisasi standar internasional (ISO) adalah konsorsium dari 63 negara yang didirikan untuk merumuskan dan mendorong standardisasi. ISO menerbitkan 9000 seri standar pada tahun 1987.

Apa itu Sertifikasi ISO 9000?

Sertifikasi ISO 9000 berfungsi sebagai acuan untuk kontrak antara pihak independen. Secara khusus, perusahaan yang memberikan kontrak pengembangan dapat membentuk opininya tentang kemungkinan kinerja vendor berdasarkan apakah vendor tersebut telah memperoleh sertifikasi ISO 9000 atau tidak. Dalam konteks ini, standar ISO 9000 menetapkan pedoman untuk memelihara sistem mutu. Kita telah melihat bahwa sistem mutu suatu organisasi berlaku untuk semua aktivitasnya yang terkait dengan produk atau layanannya. Standar ISO membahas aspek operasional (yaitu, proses) dan aspek organisasi seperti tanggung jawab, pelaporan, dll. Singkatnya, ISO 9000 menetapkan serangkaian rekomendasi untuk pengembangan produk yang dapat diulang dan berkualitas tinggi. Penting untuk disadari bahwa standar ISO 9000 adalah seperangkat pedoman untuk proses produksi dan tidak secara langsung berkaitan dengan produk itu sendiri.

ISO 9000 adalah rangkaian dari tiga standar—ISO 9001, ISO 9002, dan ISO 9003. Rangkaian standar ISO 9000 didasarkan pada premis bahwa jika proses yang tepat diikuti untuk produksi, maka produk berkualitas baik pasti akan mengikuti secara otomatis. Jenis perusahaan perangkat lunak yang menerapkan standar ISO yang berbeda adalah sebagai berikut:

ISO 9001: Standar ini berlaku untuk organisasi yang terlibat dalam desain, pengembangan, produksi, dan servis barang. Ini adalah standar yang berlaku untuk sebagian besar organisasi pengembangan perangkat lunak.

ISO 9002: Standar ini berlaku untuk organisasi yang tidak merancang produk tetapi hanya terlibat dalam produksi. Contoh kategori industri ini termasuk industri manufaktur baja dan mobil yang membeli produk dan desain pabrik dari sumber eksternal dan hanya terlibat dalam pembuatan produk tersebut. Oleh karena itu, ISO 9002 tidak berlaku untuk organisasi pengembangan perangkat lunak.

ISO 9003: Standar ini berlaku untuk organisasi yang hanya terlibat dalam pemasangan dan pengujian produk.

ISO 9000 untuk Industri Perangkat Lunak

ISO 9000 adalah standar umum yang dapat diterapkan pada banyak industri, mulai dari industri manufaktur baja hingga perusahaan penyedia layanan. Oleh karena itu, banyak dari klausa dokumen ISO 9000 ditulis menggunakan istilah umum dan sangat sulit untuk menafsirkannya dalam konteks organisasi pengembangan perangkat lunak. Alasan penting di balik situasi seperti itu adalah kenyataan bahwa pengembangan perangkat lunak dalam banyak hal sangat berbeda dari pengembangan jenis produk lainnya. Dua perbedaan utama

antara pengembangan perangkat lunak dan pengembangan jenis produk lainnya adalah sebagai berikut:

- Perangkat lunak tidak berwujud dan karena itu sulit dikendalikan. Ini berarti bahwa perangkat lunak tidak akan terlihat oleh pengguna sampai pengembangan selesai dan perangkat lunak aktif dan berjalan. Sulit untuk mengontrol dan mengelola apa pun yang tidak dapat Anda lihat dan rasakan. Sebaliknya, dalam jenis manufaktur produk lainnya seperti manufaktur mobil, Anda dapat melihat produk dikembangkan melalui berbagai tahapan seperti pemasangan mesin, pemasangan pintu, dll. Oleh karena itu, menjadi mudah untuk menentukan secara akurat berapa banyak pekerjaan yang telah diselesaikan dan untuk memperkirakan berapa lama lagi waktu yang dibutuhkan.
- Selama pengembangan perangkat lunak, satu-satunya bahan baku yang dikonsumsi adalah data. Sebaliknya, sejumlah besar bahan mentah dikonsumsi selama pengembangan produk lain. Sebagai contoh, pertimbangkan sebuah perusahaan pembuat baja. Perusahaan akan mengkonsumsi bahan baku dalam jumlah besar seperti bijih besi, batu bara, kapur, mangan, dll. Tidak mengherankan jika banyak klausul standar ISO 9000 berkaitan dengan pengendalian bahan baku. Klausa ini jelas tidak relevan untuk organisasi pengembangan perangkat lunak.

Karena perbedaan radikal antara perangkat lunak dan jenis pengembangan produk lainnya, sulit untuk menafsirkan berbagai klausa dari standar ISO asli dalam konteks industri perangkat lunak. Oleh karena itu, ISO merilis dokumen terpisah yang disebut ISO 9000 part-3 pada tahun 1991 untuk membantu menginterpretasikan standar ISO untuk industri perangkat lunak. Saat ini, panduan resmi tidak memadai mengenai interpretasi berbagai klausul standar ISO 9000 dalam konteks industri perangkat lunak dan kita harus terus melakukan referensi silang dokumen ISO 9000-3.

Mengapa Mendapatkan Sertifikasi ISO 9000?

Ada perebutan gila di antara organisasi pengembangan perangkat lunak untuk mendapatkan sertifikasi ISO karena manfaat yang ditawarkannya. Mari kita periksa beberapa manfaat yang diperoleh organisasi yang memperoleh sertifikasi ISO:

- Keyakinan pelanggan dalam suatu organisasi meningkat ketika organisasi memenuhi syarat untuk sertifikasi ISO 9001. Ini terutama berlaku di pasar internasional. Faktanya, banyak organisasi yang memberikan kontrak pengembangan perangkat lunak internasional bersikeras bahwa organisasi pengembangan memiliki sertifikasi ISO 9000. Untuk alasan ini, sangat penting bagi organisasi perangkat lunak yang terlibat dalam ekspor perangkat lunak untuk mendapatkan sertifikasi ISO 9000.
- ISO 9000 membutuhkan proses produksi perangkat lunak yang terdokumentasi dengan baik. Proses produksi perangkat lunak yang terdokumentasi dengan baik berkontribusi pada kualitas yang dapat diulang dan lebih tinggi dari perangkat lunak yang dikembangkan.
- ISO 9000 membuat proses pengembangan terfokus, efisien, dan hemat biaya.
- Sertifikasi ISO 9000 menunjukkan titik lemah organisasi dan merekomendasikan tindakan perbaikan.
- ISO 9000 menetapkan kerangka dasar untuk pengembangan proses dan TQM yang optimal.

Bagaimana Mendapatkan Sertifikasi ISO 9000?

Organisasi yang ingin mendapatkan sertifikasi ISO 9000 mengajukan pendaftaran ke pendaftar ISO 9000. Proses registrasi ISO 9000 terdiri dari tahapan sebagai berikut:

Tahap aplikasi: Setelah sebuah organisasi memutuskan untuk mengikuti sertifikasi ISO 9000, itu berlaku untuk registrar untuk pendaftaran.

Pra-penilaian: Selama tahap ini pencatat membuat penilaian kasar terhadap organisasi.

Tinjauan dokumen dan audit kecukupan: Selama tahap ini, pencatat meninjau dokumen yang diserahkan oleh organisasi dan membuat saran untuk kemungkinan perbaikan.

Audit kepatuhan: Selama tahap ini, pencatat memeriksa apakah saran yang dibuat olehnya selama peninjauan telah dipatuhi oleh organisasi atau tidak.

Pendaftaran: Pendaftar memberikan sertifikat ISO 9000 setelah berhasil menyelesaikan semua fase sebelumnya.

Pengawasan lanjutan: Pencatat terus memantau organisasi secara berkala.

ISO mengamanatkan bahwa organisasi bersertifikat dapat menggunakan sertifikat untuk iklan perusahaan tetapi tidak dapat menggunakan sertifikat untuk mengiklankan produknya. Ini mungkin karena fakta bahwa sertifikat ISO 9000 dikeluarkan untuk proses organisasi dan bukan untuk produk spesifik organisasi apa pun. Organisasi yang menggunakan sertifikat ISO untuk iklan produk menghadapi risiko pencabutan sertifikat. Di India, sertifikasi ISO 9000 ditawarkan oleh BIS (Biro Standar India), STQC (Standarisasi, pengujian, dan kontrol kualitas), dan IRQS (Sistem Kualitas Daftar India). IRQS telah diakreditasi oleh dewan badan sertifikasi Belanda (RVC).

Ringkasan Persyaratan ISO 9001

Ringkasan persyaratan utama ISO 9001 yang terkait dengan pengembangan perangkat lunak adalah sebagai berikut: Nomor bagian dalam tanda kurung sesuai dengan standar itu sendiri:

Tanggung jawab manajemen (4.1)

- Manajemen harus memiliki kebijakan mutu yang efektif.
- Tanggung jawab dan wewenang semua orang yang pekerjaannya mempengaruhi kualitas harus ditetapkan dan didokumentasikan.
- Perwakilan manajemen, independen dari proses pengembangan, harus bertanggung jawab atas sistem mutu. Persyaratan ini mungkin telah ditetapkan agar orang yang bertanggung jawab atas sistem mutu dapat bekerja dengan cara yang tidak memihak.
- Efektivitas sistem mutu harus ditinjau secara berkala melalui audit.

Sistem mutu (4.2)

Sistem mutu harus dipelihara dan didokumentasikan.

Tinjauan kontrak (4.3)

Sebelum menandatangani kontrak, organisasi harus meninjau kontrak untuk memastikan bahwa kontrak tersebut dipahami, dan bahwa organisasi memiliki kemampuan yang diperlukan untuk melaksanakan kewajibannya.

Kontrol desain (4.4)

- Proses desain harus dikontrol dengan baik, ini termasuk mengontrol pengkodean juga. Persyaratan ini berarti bahwa sistem kontrol konfigurasi yang baik harus ada.
- Masukan desain harus diverifikasi memadai.
- Desain harus diverifikasi.
- Hasil desain harus sesuai dengan kualitas yang dipersyaratkan.
- Perubahan desain harus dikendalikan.

Kontrol dokumen (4.5)

- Harus ada prosedur yang tepat untuk persetujuan, penerbitan dan penghapusan dokumen.

- Perubahan dokumen harus dikontrol. Dengan demikian, penggunaan beberapa alat manajemen konfigurasi diperlukan.

Pembelian (4.6)

Materi yang dibeli, termasuk perangkat lunak yang dibeli harus diperiksa kesesuaiannya dengan persyaratan.

Produk yang disediakan pembeli (4.7)

Materi yang dipasok oleh pembeli, misalnya, perangkat lunak yang disediakan klien harus dikelola dan diperiksa dengan benar.

Identifikasi produk (4.8)

Produk harus dapat diidentifikasi pada semua tahap proses. Dalam istilah perangkat lunak ini berarti manajemen konfigurasi.

Kontrol proses (4.9)

- Pembangunan harus dikelola dengan baik.
- Persyaratan kualitas harus diidentifikasi dalam rencana kualitas.

Inspeksi dan pengujian (4.10)

Dalam istilah perangkat lunak ini memerlukan pengujian yang efektif yaitu, pengujian unit, pengujian integrasi dan pengujian sistem. Catatan pengujian harus dipelihara.

Peralatan inspeksi, pengukuran dan pengujian (4.11)

Jika peralatan integrasi, pengukuran, dan pengujian digunakan, peralatan tersebut harus dipelihara dan dikalibrasi dengan baik.

Status inspeksi dan pengujian (4.12)

Status item harus diidentifikasi. Dalam istilah perangkat lunak ini menyiratkan manajemen konfigurasi dan kontrol rilis.

Pengendalian produk yang tidak sesuai (4.13)

Dalam istilah perangkat lunak, ini berarti menjauhkan perangkat lunak yang belum diuji atau rusak dari produk yang dirilis, atau tempat lain yang dapat menyebabkan kerusakan.

Tindakan korektif (4.14)

Persyaratan ini adalah tentang mengoreksi kesalahan ketika ditemukan, dan juga menyelidiki mengapa kesalahan terjadi dan meningkatkan proses untuk mencegah terjadinya. Jika terjadi kesalahan meskipun sistem kualitas, sistem perlu perbaikan.

Penanganan (4.15)

Klausul ini berkaitan dengan penyimpanan, pengemasan, dan pengiriman produk perangkat lunak.

Catatan kualitas (4.16)

Merekam langkah-langkah yang diambil untuk mengontrol kualitas proses sangat penting untuk dapat memastikan bahwa mereka benar-benar terjadi.

Audit kualitas (4.17)

Audit sistem mutu harus dilakukan untuk memastikan bahwa itu efektif.

Pelatihan (4.18)

Kebutuhan pelatihan harus diidentifikasi dan dipenuhi. Berbagai persyaratan ISO 9001 sebagian besar masuk akal. Panduan resmi pada interpretasi ISO 9001 tidak memadai pada saat ini, dan mengambil nasihat ahli biasanya bermanfaat.

Fitur Penting dari Persyaratan ISO 9001

Fitur yang menonjol dari semua persyaratan untuk sertifikasi ISO 9001 secara singkat adalah sebagai berikut:

Kontrol dokumen: Semua dokumen yang berkaitan dengan pengembangan produk perangkat lunak harus dikelola, disahkan, dan dikendalikan dengan benar. Ini membutuhkan sistem manajemen konfigurasi untuk diterapkan.

Perencanaan: Rencana yang tepat harus disiapkan dan kemudian kemajuan terhadap rencana ini harus dipantau.

Tinjauan: Dokumen penting di semua fase harus diperiksa dan ditinjau secara independen untuk efektivitas dan kebenaran.

Pengujian: Produk harus diuji terhadap spesifikasi.

Aspek organisasi: Beberapa aspek organisasi harus ditangani misalnya, pelaporan manajemen tim mutu.

ISO 9000-2000

ISO merevisi standar kualitas pada tahun 2000 untuk menyempurnakan standar. Perubahan besar termasuk mekanisme untuk perbaikan proses yang berkesinambungan. Ada juga peningkatan penekanan pada peran manajemen puncak, termasuk menetapkan tujuan yang terukur untuk berbagai peran dan tingkat organisasi. Standar baru mengakui bahwa mungkin ada banyak proses dalam suatu organisasi.

Kekurangan Sertifikasi ISO 9000

Meskipun ISO 9000 secara luas digunakan untuk menyiapkan sistem mutu yang efektif dalam suatu organisasi, ia mengalami beberapa kekurangan. Beberapa kekurangan dari proses sertifikasi ISO 9000 adalah sebagai berikut:

- ISO 9000 membutuhkan proses produksi perangkat lunak yang harus dipatuhi, tetapi tidak menjamin prosesnya berkualitas tinggi. Itu juga tidak memberikan pedoman apa pun untuk mendefinisikan proses yang sesuai.
- Proses sertifikasi ISO 9000 tidak mudah dan tidak ada lembaga akreditasi internasional. Oleh karena itu, kemungkinan besar variasi dalam norma pemberian sertifikat dapat terjadi di antara lembaga akreditasi yang berbeda dan juga di antara pendaftar.
- Organisasi yang mendapatkan sertifikasi ISO 9000 sering cenderung meremehkan keahlian domain dan kecerdikan pengembang. Organisasi-organisasi ini mulai percaya bahwa karena ada proses yang baik, hasil pengembangan benar-benar mandiri. Artinya, setiap developer sama efektifnya dengan developer lain dalam melakukan aktivitas pengembangan perangkat lunak tertentu. Dalam industri manufaktur ada hubungan yang jelas antara kualitas proses dan kualitas produk. Setelah suatu proses dikalibrasi, dapat dijalankan lagi dan lagi menghasilkan barang berkualitas. Banyak bidang pengembangan perangkat lunak yang sangat khusus sehingga keahlian dan pengalaman khusus di bidang ini (keahlian domain) diperlukan. Selain itu, tidak seperti dalam pembuatan produk umum, kecerdikan dan efektivitas praktik pribadi memainkan peran penting dalam menentukan hasil yang dihasilkan oleh pengembang. Dengan kata lain, pengembangan perangkat lunak adalah proses kreatif dan keterampilan serta pengalaman individu adalah penting.
- ISO 9000 tidak secara otomatis mengarah pada peningkatan proses berkelanjutan. Dengan kata lain, tidak secara otomatis mengarah pada TQM.

11.5 MODEL KEMATIAN KEMAMPUAN SEI

Model kematangan kemampuan SEI (SEI CMM) diusulkan oleh Institut Rekayasa Perangkat Lunak dari Universitas Carnegie Mellon, AS. CMM berpola setelah karya perintis Philip Crosby yang menerbitkan grid kematangannya dari lima tahap evolusi dalam mengadopsi praktik kualitas dalam bukunya “Quality is Free” [Crosby79]. Departemen Pertahanan Amerika Serikat (US DoD) adalah pembeli terbesar produk perangkat lunak. Itu sering menghadapi kesulitan dalam kinerja vendor, dan harus berkali-kali hidup dengan

produk berkualitas rendah, pengiriman terlambat, dan peningkatan biaya. Dalam konteks ini, SEI CMM pada awalnya dikembangkan untuk membantu Departemen Pertahanan (DoD) AS dalam akuisisi perangkat lunak. Alasannya adalah untuk memasukkan kemungkinan kinerja kontraktor sebagai faktor dalam pemberian kontrak. Sebagian besar kontraktor DoD utama memulai inisiatif peningkatan proses berbasis CMM saat mereka bersaing untuk mendapatkan kontrak DoD. Diamati bahwa model SEI CMM membantu organisasi untuk meningkatkan kualitas perangkat lunak yang mereka kembangkan dan oleh karena itu adopsi model SEI CMM memiliki manfaat bisnis yang signifikan.

Secara bertahap banyak organisasi komersial mulai mengadopsi CMM sebagai kerangka kerja untuk inisiatif perbaikan internal mereka sendiri. Dengan kata sederhana, CMM adalah model referensi untuk menilai kematangan proses perangkat lunak ke tingkat yang berbeda. Ini dapat digunakan untuk memprediksi hasil yang paling mungkin diharapkan dari proyek berikutnya yang dilakukan organisasi. Harus diingat bahwa SEI CMM dapat digunakan dalam dua cara - evaluasi kemampuan dan penilaian proses perangkat lunak. Evaluasi kemampuan dan penilaian proses perangkat lunak berbeda dalam motivasi, tujuan, dan penggunaan hasil akhir. Evaluasi kemampuan menyediakan cara untuk menilai kemampuan proses perangkat lunak suatu organisasi. Evaluasi kemampuan dikelola oleh otoritas pemberi kontrak, dan oleh karena itu hasilnya akan menunjukkan kemungkinan kinerja kontraktor jika kontraktor diberikan pekerjaan. Di sisi lain, penilaian proses perangkat lunak digunakan oleh organisasi dengan tujuan untuk meningkatkan kemampuan prosesnya sendiri. Dengan demikian, jenis penilaian yang terakhir adalah untuk penggunaan internal murni oleh perusahaan.

Tingkat SEI CMM yang berbeda telah dirancang sedemikian rupa sehingga mudah bagi organisasi untuk membangun sistem kualitasnya secara perlahan mulai dari awal. SEI CMM mengklasifikasikan industri pengembangan perangkat lunak ke dalam lima tingkat kematangan berikut:

Tingkat 1: Awal

Organisasi pengembangan perangkat lunak pada tingkat ini dicirikan oleh aktivitas ad hoc. Sangat sedikit atau tidak ada proses yang didefinisikan dan diikuti. Karena proses produksi perangkat lunak tidak didefinisikan, insinyur yang berbeda mengikuti proses mereka sendiri dan sebagai akibatnya upaya pengembangan menjadi kacau. Oleh karena itu, ini juga disebut tingkat kacau. Keberhasilan proyek tergantung pada upaya individu dan heroik. Ketika seorang developer meninggalkan organisasi, penerusnya akan mengalami kesulitan besar dalam memahami proses yang diikuti dan pekerjaan yang diselesaikan. Juga, tidak ada praktik manajemen proyek formal yang diikuti. Akibatnya, tekanan waktu meningkat menjelang akhir waktu pengiriman, akibatnya jalan pintas dicoba untuk menghasilkan produk berkualitas rendah.

Level 2: Dapat diulang

Pada tingkat ini, praktik manajemen proyek dasar seperti pelacakan biaya dan jadwal ditetapkan. Alat manajemen konfigurasi digunakan pada item yang diidentifikasi untuk kontrol konfigurasi. Teknik estimasi ukuran dan biaya seperti analisis titik fungsi, COCOMO, dll., digunakan. Disiplin proses yang diperlukan diterapkan untuk mengulangi kesuksesan sebelumnya pada proyek dengan aplikasi serupa. Meskipun ada pemahaman kasar di antara para developer tentang proses yang diikuti, prosesnya tidak didokumentasikan. Praktik manajemen konfigurasi digunakan untuk semua hasil proyek. Harap diingat bahwa kesempatan untuk mengulang suatu proses hanya ada ketika sebuah perusahaan memproduksi sekelompok produk. Karena produknya sangat mirip, kisah sukses pengembangan satu produk dapat diulang untuk produk lainnya. Dalam organisasi

pengembangan perangkat lunak yang tidak dapat diulang, proyek pengembangan produk perangkat lunak menjadi sukses terutama karena inisiatif, usaha, kecemerlangan, atau antusiasme yang ditampilkan oleh individu-individu tertentu. Di sisi lain, dalam organisasi pengembangan perangkat lunak yang tidak dapat diulang, kemungkinan keberhasilan penyelesaian proyek perangkat lunak sebagian besar tergantung pada siapa anggota tim. Untuk alasan ini, keberhasilan pengembangan satu produk oleh organisasi semacam itu tidak secara otomatis berarti bahwa pengembangan produk berikutnya akan berhasil.

Tingkat 3: Ditetapkan

Pada tingkat ini, proses untuk aktivitas manajemen dan pengembangan ditetapkan dan didokumentasikan. Ada pemahaman umum di seluruh organisasi tentang kegiatan, peran, dan tanggung jawab. Proses meskipun didefinisikan, proses dan kualitas produk tidak diukur. Pada tingkat ini, organisasi membangun kemampuan karyawannya melalui program pelatihan berkala. Juga, teknik review ditekankan dan didokumentasikan untuk mencapai fase penahanan kesalahan. ISO 9000 bertujuan untuk mencapai level ini.

Level 4: Terkelola

Pada level ini, fokusnya adalah pada metrik perangkat lunak. Metrik proses dan produk dikumpulkan. Sasaran mutu kuantitatif ditetapkan untuk produk dan pada saat penyelesaian pengembangan diperiksa apakah sasaran mutu kuantitatif untuk produk terpenuhi. Berbagai alat seperti bagan Pareto, diagram tulang ikan, dll. digunakan untuk mengukur kualitas produk dan proses. Metrik proses digunakan untuk memeriksa apakah proyek dilakukan dengan memuaskan. Dengan demikian, hasil pengukuran proses digunakan untuk mengevaluasi kinerja proyek daripada meningkatkan proses.

Level 5: Mengoptimalkan

Pada tahap ini, metrik proses dan produk dikumpulkan. Data pengukuran proses dan produk dianalisis untuk perbaikan proses yang berkelanjutan. Misalnya, jika dari analisis hasil pengukuran proses, ditemukan bahwa tinjauan kode tidak terlalu efektif dan sejumlah besar kesalahan hanya terdeteksi selama pengujian unit, maka proses akan disesuaikan untuk membuat tinjauan lebih lanjut. efektif. Juga, pelajaran yang dipetik dari proyek-proyek tertentu dimasukkan ke dalam proses. Perbaikan proses yang berkelanjutan dicapai baik dengan menganalisis umpan balik kuantitatif secara hati-hati dari pengukuran proses dan juga dari penerapan ide dan teknologi inovatif. Pada CMM level 5, sebuah organisasi akan mengidentifikasi praktik dan inovasi rekayasa perangkat lunak terbaik (yang mungkin berupa alat, metode, atau proses) dan akan mentransfernya ke seluruh organisasi. Organisasi Level 5 biasanya memiliki departemen yang tanggung jawab tunggalnya adalah mengasimilasi alat dan teknologi terbaru dan menyebarkannya ke seluruh organisasi. Karena proses berubah terus-menerus, menjadi penting untuk mengelola proses yang berubah secara efektif. Oleh karena itu, organisasi level 5 menggunakan teknik manajemen konfigurasi untuk mengelola perubahan proses.

Kecuali untuk level 1, setiap level maturitas dicirikan oleh beberapa area proses utama (KPA) yang menunjukkan area yang harus difokuskan organisasi untuk meningkatkan proses perangkat lunaknya ke level ini dari level sebelumnya. Masing-masing area fokus mengidentifikasi sejumlah praktik atau kegiatan utama yang perlu dilaksanakan. Dengan kata lain, KPA menangkap area fokus dari suatu level. Fokus dari setiap level dan area proses utama yang sesuai ditunjukkan di Tabel 11.1:

Tabel 11.1 Area fokus level CMM dan Area Proses Utama

Level CMM	***Fokus***	***KPA (Key Process Areas)***
Inisial	Orang yang kompeten	

Pengulangan	Manajemen proyek	Perencanaan proyek perangkat lunak Manajemen konfigurasi perangkat lunak
Pendefinisian	Definisi proses	Definisi proses Program pelatihan Ulasan sejawat
Pengelolaan	Produk dan kualitas proses	Metrik proses kuantitatif Manajemen kualitas perangkat lunak
Optiisasi	Improvisasi proses berkelanjutan	Pencegahan cacat Manajemen perubahan proses Manajemen perubahan teknologi

SEI CMM menyediakan daftar area utama yang menjadi fokus untuk membawa organisasi dari satu tingkat kedewasaan ke tingkat berikutnya. Dengan demikian, ini menyediakan cara untuk peningkatan kualitas secara bertahap melalui beberapa tahap. Setiap tahap telah dirancang dengan hati-hati sehingga satu tahap meningkatkan kemampuan yang sudah dibangun. Misalnya, mencoba menerapkan proses yang ditentukan (tingkat 3) sebelum proses yang dapat diulang (tingkat 2) akan menjadi kontraproduktif karena menjadi sulit untuk mengikuti proses yang ditentukan karena tekanan jadwal dan anggaran.

Bukti substansial kini telah terakumulasi yang menunjukkan bahwa mengadopsi SEI CMM memiliki beberapa keuntungan bisnis. Namun, organisasi yang mencoba CMM sering menghadapi masalah yang berasal dari karakteristik CMM itu sendiri.

Kekurangan CMM: CMM memang memiliki beberapa kekurangan. Yang penting di antaranya adalah sebagai berikut:

- Keluhan yang paling sering oleh organisasi saat mencoba inisiatif peningkatan proses berbasis CMM adalah bahwa mereka memahami apa yang perlu ditingkatkan, tetapi mereka membutuhkan lebih banyak panduan tentang cara meningkatkannya.
- Kekurangan lainnya (yang umum pada ISO 9000) adalah dokumen yang lebih tebal, informasi yang lebih detail, dan rapat yang lebih lama dianggap lebih baik. Ini berbeda dengan prinsip-prinsip ekonomi perangkat lunak—mengurangi kompleksitas dan menjaga dokumentasi seminimal mungkin tanpa mengorbankan detail yang relevan.
- Mendapatkan ukuran yang akurat dari tingkat kematangan organisasi saat ini juga merupakan masalah. CMM mengambil pendekatan berbasis aktivitas untuk mengukur kedewasaan; jika Anda melakukan serangkaian aktivitas yang ditentukan maka Anda berada pada level tertentu. Tidak ada yang mencirikan atau mengukur apakah Anda melakukan aktivitas ini dengan cukup baik untuk memberikan hasil yang diinginkan.

Perbandingan Antara Sertifikasi ISO 9000 dan SEI/CMM

Mari kita bandingkan beberapa karakteristik utama sertifikasi ISO 9000 dan model SEI CMM untuk penilaian kualitas:

- ISO 9000 diberikan oleh badan standar internasional. Oleh karena itu, sertifikasi ISO 9000 dapat dikutip oleh suatu organisasi dalam dokumen resmi, komunikasi dengan pihak eksternal, dan dalam penawaran tender. Namun, penilaian SEI CMM murni untuk penggunaan internal.
- SEI CMM dikembangkan secara khusus untuk industri perangkat lunak dan oleh karena itu membahas banyak masalah yang khusus untuk industri perangkat lunak saja.

- SEI CMM melampaui jaminan kualitas dan mempersiapkan organisasi untuk akhirnya mencapai TQM. Bahkan, ISO 9001 bertujuan pada level 3 model SEI CMM.
- Model SEI CMM menyediakan daftar area proses utama (KPA) di mana organisasi pada setiap tingkat kematangan perlu berkonsentrasi untuk membawanya dari satu tingkat kematangan ke tingkat berikutnya. Dengan demikian, ini menyediakan cara untuk mencapai peningkatan kualitas secara bertahap. Sebaliknya, organisasi yang mengadopsi ISO 9000 memenuhi syarat untuk itu atau tidak memenuhi syarat.

Apakah SEI CMM Berlaku untuk Organisasi Kecil?

Pendekatan yang sangat sistematis dan terukur untuk pengembangan perangkat lunak sesuai dengan organisasi besar yang berurusan dengan perangkat lunak yang dinegosiasikan, perangkat lunak yang kritis terhadap keselamatan, dll. Namun, bagaimana dengan organisasi kecil? Organisasi-organisasi ini biasanya menangani aplikasi seperti Internet kecil, aplikasi e-commerce, dan seringkali tanpa rangkaian produk yang mapan, basis pendapatan, dan pengalaman pada proyek sebelumnya, dll. Untuk organisasi semacam itu, penilaian berbasis CMM mungkin berlebihan. Organisasi-organisasi ini perlu beroperasi lebih efisien pada tingkat kedewasaan yang lebih rendah. Misalnya, mereka perlu mempraktikkan manajemen proyek yang efektif, ulasan, manajemen konfigurasi, dll.

Integrasi Model Kematangan Kemampuan (CMMI)

Integrasi model maturitas kapabilitas (CMMI) adalah penerus dari model maturitas kapabilitas (CMM). CMM dikembangkan dari tahun 1987 hingga 1997. Pada tahun 2002, CMMI Versi 1.1 dirilis. Versi 1.2 diikuti pada tahun 2006. CMMI bertujuan untuk meningkatkan kegunaan model kematangan dengan mengintegrasikan banyak model yang berbeda ke dalam satu kerangka kerja.

Setelah CMMI pertama kali dirilis pada tahun 1990, CMMI diadopsi dan digunakan di banyak domain. Misalnya, CMM dikembangkan untuk disiplin ilmu seperti rekayasa sistem (SE-CMM), manajemen orang (PCMM), akuisisi perangkat lunak (SA-CMM), dan lain-lain. Meskipun banyak organisasi menemukan model ini berguna, mereka juga berjuang dengan masalah yang disebabkan oleh tumpang tindih, inkonsistensi, dan integrasi model. Dalam konteks ini, CMMI digeneralisasi untuk dapat diterapkan ke banyak domain. Misalnya, kata "perangkat lunak" tidak muncul dalam definisi CMMI. Penyatuan berbagai jenis domain menjadi satu model membuat CMMI sangat abstrak. CMMI, seperti pendahulunya, menggambarkan lima tingkat kedewasaan yang berbeda.

11.6 BEBERAPA STANDAR KUALITAS PENTING LAINNYA

Peningkatan Proses Perangkat Lunak dan Penentuan Kemampuan (SPICE)

SPICE adalah singkatan dari Software Process Improvement and Capability Determination. Ini adalah standar ISO (IEC 15504). Ini membedakan berbagai jenis proses—proses rekayasa, proses manajemen, pemasok-pelanggan, dukungan. Untuk setiap proses, ini mendefinisikan enam tingkat kematangan kemampuan. Ini mengintegrasikan standar yang ada untuk menyediakan model referensi proses tunggal dan model penilaian proses yang membahas kategori luas dari proses perusahaan.

Personal Software Process (PSP)

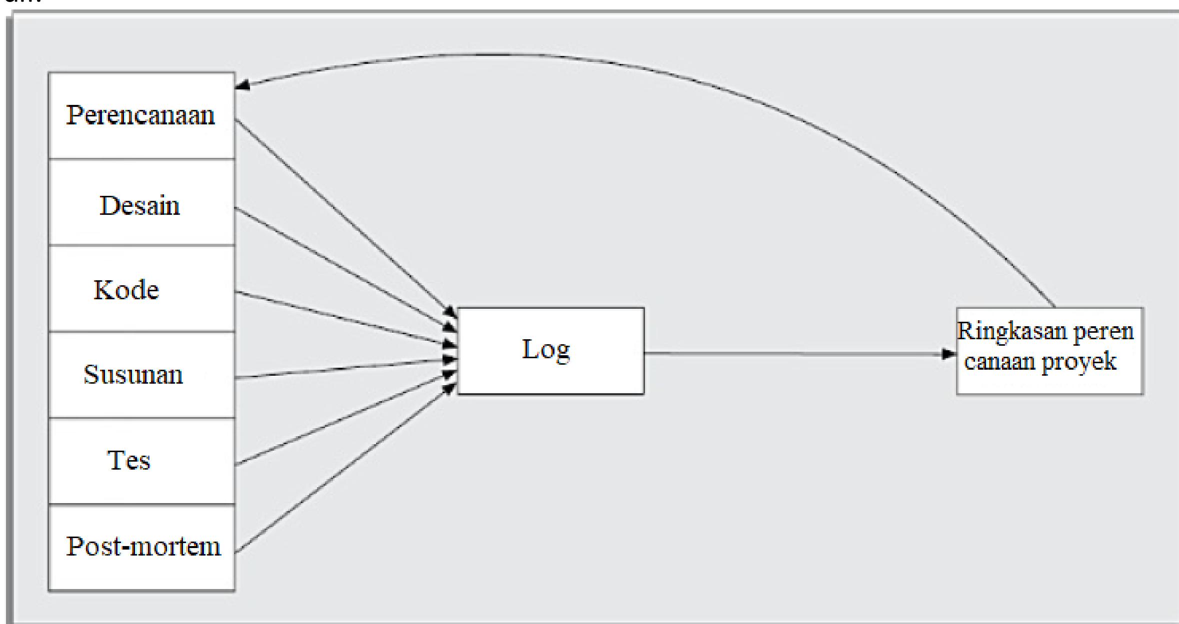
PSP didasarkan pada karya David Humphrey [Hum97]. PSP adalah versi yang diperkecil dari proses perangkat lunak industri yang dibahas di bagian terakhir. PSP cocok untuk penggunaan individu. Penting untuk dicatat bahwa SEI CMM tidak memberi tahu developer perangkat lunak cara menganalisis, merancang, mengkode, menguji, atau

mendokumentasikan produk perangkat lunak, tetapi mengasumsikan bahwa para insinyur menggunakan praktik pribadi yang efektif. PSP menyadari bahwa proses untuk penggunaan individu berbeda dari yang diperlukan untuk sebuah tim. Kualitas dan produktivitas seorang insinyur sangat bergantung pada prosesnya. PSP adalah kerangka kerja yang membantu para insinyur untuk mengukur dan meningkatkan cara mereka bekerja. Ini membantu dalam mengembangkan keterampilan dan metode pribadi dengan memperkirakan, merencanakan, dan melacak kinerja terhadap rencana, dan menyediakan proses yang ditentukan yang dapat disetel oleh individu.

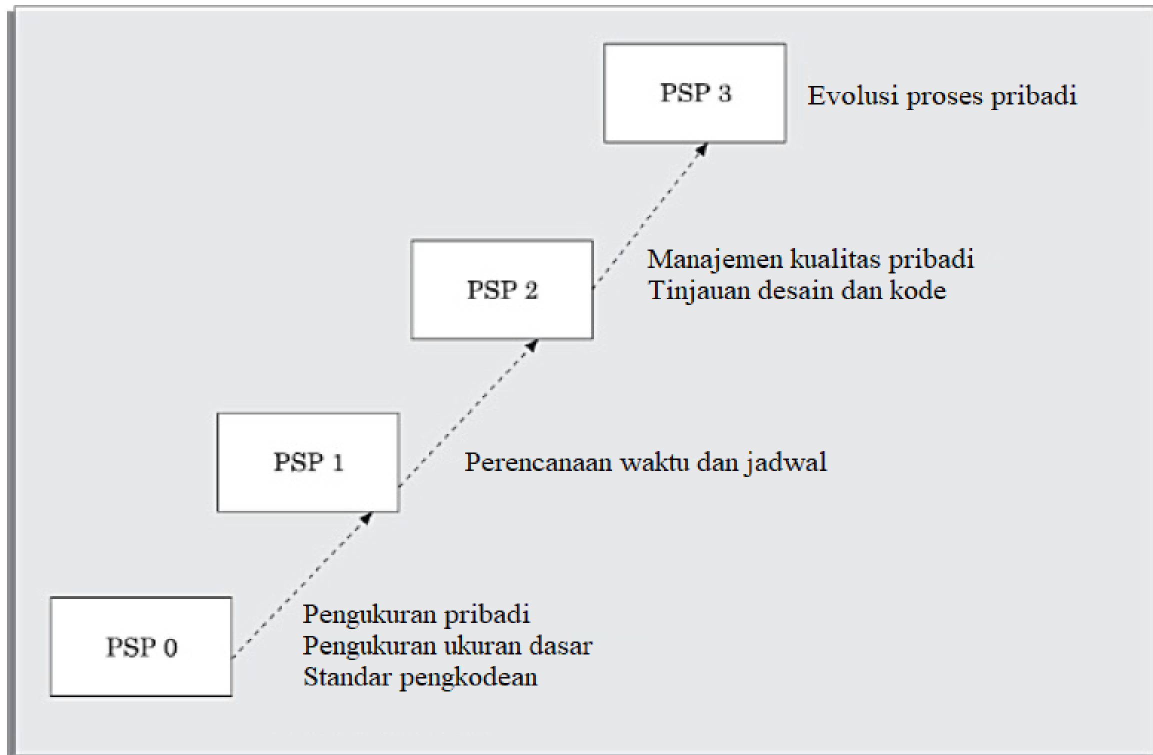
Pengukuran waktu: PSP menganjurkan bahwa developer harus memeras cara mereka menghabiskan waktu. Sebab, kegiatan yang membosankan terkesan lebih lama dari yang sebenarnya dan kegiatan yang menarik terkesan singkat. Oleh karena itu, waktu aktual yang dihabiskan untuk suatu tugas harus diukur dengan bantuan stop-watch untuk mendapatkan gambaran objektif tentang waktu yang dihabiskan. Misalnya, dia mungkin menghentikan jam saat menghadiri panggilan telepon, istirahat minum kopi, dll. Seorang insinyur harus mengukur waktu yang dia habiskan untuk berbagai kegiatan pengembangan seperti merancang, menulis kode, pengujian, dll.

Perencanaan PSP: Individu harus merencanakan proyek mereka. Kecuali jika seseorang merencanakan aktivitasnya dengan benar, upaya tinggi yang tidak proporsional dapat dihabiskan untuk aktivitas sepele dan aktivitas penting dapat dikompromikan, yang mengarah pada hasil berkualitas buruk. developer harus memperkirakan LOC maksimum, minimum, dan rata-rata yang diperlukan untuk produk. Mereka harus menggunakan produktivitas mereka dalam menit/LOC untuk menghitung waktu pengembangan maksimum, minimum, dan rata-rata. Mereka harus mencatat data rencana dalam ringkasan rencana proyek.

PSP secara skematis ditunjukkan pada Gambar 11.4. Saat melakukan fase yang berbeda, seorang individu harus mencatat data log menggunakan pengukuran waktu. Selama post-mortem, mereka dapat membandingkan data log dengan rencana proyek mereka untuk mencapai perencanaan yang lebih baik di proyek masa depan, untuk meningkatkan prosesnya, dll.



Gambar 11.4 Representasi skematis dari PSP.



Gambar 11.5 Tingkatan PSP.

Tingkat PSP diringkas dalam Gambar 11.5. PSP2 memperkenalkan manajemen cacat melalui penggunaan daftar periksa untuk tinjauan kode dan desain. Daftar periksa dikembangkan dengan menganalisis data cacat yang dikumpulkan dari proyek sebelumnya.

11.7 ENAM SIGMA

Perusahaan General Electric (GE) pertama kali memulai Six Sigma pada tahun 1995 setelah Motorola dan Allied Signal merintis jejak Six Sigma. Sejak mereka, ribuan perusahaan di seluruh dunia telah menemukan manfaat Six Sigma yang luas. Tujuan dari Six Sigma adalah untuk meningkatkan proses untuk melakukan sesuatu dengan lebih baik, lebih cepat, dan dengan biaya yang lebih rendah. Ini dapat digunakan untuk meningkatkan setiap aspek bisnis, mulai dari produksi, hingga sumber daya manusia, hingga entri pesanan, hingga dukungan teknis. Six Sigma dapat digunakan untuk setiap aktivitas yang berkaitan dengan biaya, ketepatan waktu, dan kualitas hasil. Oleh karena itu, ini berlaku untuk hampir semua industri.

Six Sigma di banyak organisasi berarti berjuang untuk mendekati kesempurnaan. Six Sigma adalah pendekatan disiplin berbasis data untuk menghilangkan cacat dalam proses apa pun – mulai dari manufaktur hingga transaksional dan dari produk ke layanan. Representasi statistik dari Six Sigma menggambarkan secara kuantitatif bagaimana suatu proses bekerja. Untuk mencapai Six Sigma, suatu proses tidak boleh menghasilkan lebih dari 3,4 cacat per sejuta peluang. Cacat Six Sigma didefinisikan sebagai perilaku sistem yang tidak sesuai dengan spesifikasi pelanggan. Jumlah total peluang Six Sigma kemudian jumlah total peluang untuk cacat. Proses sigma dapat dengan mudah dihitung menggunakan kalkulator Six Sigma.

Tujuan mendasar dari metodologi Six Sigma adalah penerapan strategi berbasis pengukuran yang berfokus pada peningkatan proses dan pengurangan variasi melalui penerapan proyek peningkatan Six Sigma. Hal ini dicapai melalui penggunaan dua sub-metodologi Six Sigma—DMAIC dan DMADV. Proses Six Sigma DMAIC (mendefinisikan, mengukur, menganalisis, meningkatkan, mengontrol) adalah sistem perbaikan untuk proses

yang ada di bawah spesifikasi dan mencari peningkatan tambahan. Proses Six Sigma DMADV (mendefinisikan, mengukur, menganalisis, merancang, memverifikasi) adalah sistem perbaikan yang digunakan untuk mengembangkan proses atau produk baru pada tingkat kualitas Six Sigma. Ini juga dapat digunakan jika proses saat ini membutuhkan lebih dari sekadar peningkatan bertahap. Kedua proses Six Sigma dijalankan oleh Six Sigma Green Belts dan Six Sigma Black Belts, dan diawasi oleh Six Sigma Master Black Belts. Banyak kerangka kerja yang ada untuk menerapkan metodologi Six Sigma. Konsultan Six Sigma di seluruh dunia juga telah mengembangkan metodologi eksklusif untuk menerapkan kualitas Six Sigma, berdasarkan filosofi manajemen perubahan dan aplikasi alat yang serupa.

11.8 RINGKASAN

- Jika jumlah cacat yang tersisa dalam produk perangkat lunak berkorelasi dengan keandalannya, maka tetap tidak ada hubungan sederhana antara keduanya. Alasan penting di balik ini adalah bahwa kesalahan yang ada di bagian inti dan non-inti dari produk perangkat lunak mempengaruhi keandalan produk secara berbeda.
- Kita membahas beberapa metrik untuk mengukur keandalan perangkat lunak tertentu. Kekurangan metrik ini dan menyimpulkan bahwa tak satu pun dari metrik ini dapat digunakan untuk memberikan ukuran keandalan produk perangkat lunak yang sepenuhnya memuaskan ditunjukkan pada bagian ini.
- Pemodelan pertumbuhan keandalan dan penggunaannya untuk menentukan berapa lama untuk menguji suatu produk.
- Metode Penilaian Proses Perangkat Lunak mulai digunakan secara lebih umum dalam pengelolaan pengembangan, akuisisi, dan pemanfaatan perangkat lunak, di hadapan bukti substansial keberhasilan metode tersebut dalam mendorong peningkatan baik dalam kualitas maupun produktivitas.
- ISO 9000 dan SEI CMM sebagai dua set pedoman untuk menyiapkan sistem mutu. Seri ISO 9000 adalah standar yang berlaku untuk spektrum industri yang luas, sedangkan model SEI CMM adalah seperangkat pedoman untuk menyiapkan sistem mutu yang secara khusus menangani kebutuhan organisasi pengembangan perangkat lunak. Oleh karena itu, model SEI CMM membahas berbagai masalah yang berkaitan dengan industri perangkat lunak secara lebih terfokus. Misalnya, model SEI CMM menyarankan struktur 5 tingkat. Di sisi lain, ISO 9000 telah dirumuskan oleh badan standar dan oleh karena itu sertifikat dapat digunakan sebagai kontrak antara pihak eksternal yang independen, sedangkan SEI CMM membahas langkah demi langkah peningkatan praktik kualitas organisasi.

11.9 LATIHAN

1. Pilih opsi yang benar:
    a. Manakah dari berikut ini yang merupakan penggunaan praktis dari pemodelan pertumbuhan keandalan?
        i. Menentukan umur operasional perangkat lunak aplikasi
        ii. Tentukan kapan harus menghentikan pengujian
        iii. Memasukkan informasi keandalan saat merancang
        iv. Memasukkan informasi pertumbuhan keandalan dalam kode
    b. Bagaimana ketersediaan perangkat lunak dengan angka keandalan berikut? Mean Time Between Failure (MTBF) = 25 hari, Mean Time To Repair (MTTR) = 6 jam:

   i. 1 persen
   ii. 24 persen
   iii. 99 persen
   iv. 99,09 persen
c. Organisasi perangkat lunak telah dinilai pada SEI CMM Level 4. Manakah dari berikut ini yang merupakan prasyarat untuk mencapai Level 5:
   i. Deteksi Cacat
   ii. Pencegahan Cacat
   iii. Isolasi Cacat
   iv. Propagasi Cacat
d. Manakah dari berikut ini yang menjadi fokus paradigma kualitas modern:
   i. Jaminan proses
   ii. Jaminan produk
   iii. Pengujian menyeluruh
   iv. Pengujian dan penolakan menyeluruh terhadap produk yang buruk
e. Manakah dari berikut ini yang ditunjukkan oleh pengembangan perangkat lunak berulang SEI CMM:
   i. Keberhasilan dalam pengembangan perangkat lunak dapat diulang
   ii. Keberhasilan dalam pengembangan perangkat lunak dapat diulang dalam proyek pengembangan perangkat lunak terkait.
   iii. Keberhasilan dalam pengembangan perangkat lunak dapat diulang di semua proyek pengembangan perangkat lunak yang mungkin dilakukan organisasi.
   iv. Ketika tim pengembangan yang sama dipilih untuk mengembangkan perangkat lunak lain, mereka dapat mengulangi kesuksesan mereka.
f. Manakah dari berikut ini yang merupakan tujuan utama pengujian statistik:
   i. Gunakan teknik statistik untuk merancang kasus uji
   ii. Menerapkan teknik statistik pada hasil pengujian untuk menentukan apakah perangkat lunak telah diuji secara memadai
   iii. Perkirakan keandalan perangkat lunak
   iv. Menerapkan teknik statistik pada hasil pengujian untuk menentukan berapa lama pengujian perlu dilakukan.

2. Definisikan istilah keandalan perangkat lunak dan kualitas perangkat lunak. Bagaimana ini bisa diukur?
3. Identifikasi faktor-faktor yang membuat pengukuran keandalan perangkat lunak menjadi masalah yang jauh lebih sulit daripada pengukuran keandalan perangkat keras.
4. Melalui plot sederhana, jelaskan bagaimana keandalan produk perangkat lunak berubah selama masa pakainya. Gambarkan perubahan keandalan untuk produk perangkat keras selama masa pakainya dan jelaskan mengapa kedua plot terlihat sangat berbeda.
5. Apa yang Anda pahami dengan model pertumbuhan keandalan? Bagaimana pemodelan pertumbuhan keandalan berguna?
6. Jelaskan dengan menggunakan satu kalimat sederhana masing-masing apa yang Anda pahami dengan ukuran reliabilitas berikut:
   a. POFOD 0,001
   b. ROCOF 0,002
   c. MTBF 200 unit

   d. Ketersediaan 0,998
7. Apa itu pengujian statistik? Dalam hal apa itu berguna selama pengembangan perangkat lunak? Jelaskan dalam langkah-langkah yang berbeda dari pengujian statistik.
8. Tentukan tiga metrik untuk mengukur keandalan perangkat lunak. Apakah Anda menganggap metrik ini sepenuhnya memuaskan untuk memberikan ukuran keandalan sistem? Justifikasi jawaban Anda.
9. Bagaimana Anda bisa menentukan jumlah cacat laten dalam produk perangkat lunak selama fase pengujian?
10. Nyatakan TR UE atau FALSE dari berikut ini. Dukung jawaban Anda dengan alasan yang tepat:
    a. Keandalan produk perangkat lunak meningkat hampir secara linier, setiap kali cacat terdeteksi dan diperbaiki.
    b. Saat pengujian berlanjut, tingkat pertumbuhan keandalan melambat yang menunjukkan pengembalian pertumbuhan keandalan yang semakin berkurang dengan upaya pengujian.
    c. Paradigma jaminan kualitas modern berpusat pada pelaksanaan pengujian produk secara menyeluruh.
    d. Penggunaan penting dari penerimaan sertifikasi ISO 9001 oleh organisasi perangkat lunak adalah dapat meningkatkan upaya penjualannya dengan mengiklankan produknya sesuai dengan sertifikasi ISO 9001.
    e. Perangkat lunak yang sangat andal dapat disebut sebagai perangkat lunak yang berkualitas baik.
    f. Jika organisasi yang dinilai pada SEI CMM level 1 telah berhasil mengembangkan satu produk perangkat lunak, maka diharapkan untuk mengulangi kesuksesannya pada produk serupa.
11. Apa yang dimaksud dengan parameter kualitas "kesesuaian tujuan" dalam konteks produk perangkat lunak? Mengapa ini bukan kriteria yang memuaskan untuk menentukan kualitas produk perangkat lunak?
12. Dapatkah keandalan produk perangkat lunak ditentukan dengan memperkirakan jumlah cacat laten dalam perangkat lunak? Jika jawaban Anda adalah “ya”, jelaskan bagaimana keandalan dapat ditentukan dari perkiraan jumlah cacat laten dalam produk perangkat lunak. Jika jawaban Anda “tidak”, jelaskan mengapa keandalan produk perangkat lunak tidak dapat ditentukan dari perkiraan jumlah cacat laten.
13. Mengapa penting bagi organisasi pengembangan perangkat lunak untuk mendapatkan sertifikasi ISO 9001?
14. Diskusikan manfaat relatif dari sertifikasi ISO 9001 dan penilaian kualitas berbasis SEI CMM.
15. Sebutkan lima persyaratan penting yang harus dipatuhi oleh organisasi pengembangan perangkat lunak sebelum dapat diberikan sertifikat ISO 9001.
16. Apa saja kekurangan dari proses sertifikasi ISO?
17. Dengan bantuan contoh yang sesuai, diskusikan jenis organisasi perangkat lunak yang menerapkan standar ISO 9001, 9002, dan 9003.
18. Selama proses pengujian perangkat lunak, mengapa pertumbuhan keandalan awalnya tinggi, tetapi kemudian melambat?
19. Jika sebuah organisasi tidak mendokumentasikan sistem kualitasnya, masalah apa yang akan dihadapinya?
20. Menurut Anda apa produk software yang berkualitas?

21. Diskusikan tahapan di mana paradigma sistem mutu dan metode penjaminan mutu telah berkembang selama bertahun-tahun.
22. Standar mana yang berlaku untuk industri perangkat lunak, ISO 9001, ISO 9002, atau ISO 9003?
23. Dalam organisasi pengembangan perangkat lunak, identifikasi orang yang bertanggung jawab untuk melaksanakan kegiatan jaminan kualitas. Jelaskan tugas utama yang mereka lakukan untuk memenuhi tanggung jawab ini.
24. Misalkan sebuah organisasi menyebutkan dalam iklan pekerjaannya bahwa ia telah dinilai pada level 3 SEI CMM, apa yang dapat Anda simpulkan tentang praktik kualitas saat ini di organisasi tersebut? Apa yang harus dilakukan organisasi ini untuk mencapai SEI CMM level 4?
25. Misalkan sebagai presiden perusahaan, Anda memiliki pilihan untuk memilih model kualitas berbasis ISO 9000 atau model berbasis SEI CMM, mana yang akan Anda pilih? Berikan alasan di balik pilihan Anda.
26. Apa yang Anda pahami tentang manajemen kualitas total (TQM)? Apa keuntungan dari TQM? Apakah standar ISO 9000 bertujuan untuk TQM?
27. Apa kegiatan utama dari sistem mutu modern?
28. Dalam organisasi pengembangan perangkat lunak, siapa yang bertanggung jawab untuk memastikan bahwa produknya berkualitas tinggi? Jelaskan tugas utama yang mereka lakukan untuk memenuhi tanggung jawab ini.
29. Apa yang Anda pahami dengan pengembangan perangkat lunak berulang? Organisasi yang dinilai pada tingkat kematangan SEI CMM mana yang mencapai pengembangan perangkat lunak berulang?
30. Apa yang Anda pahami tentang key process area (KPA), dalam konteks SEI CMM? Apakah ada masalah jika organisasi mencoba menerapkan KPA CMM SEI tingkat yang lebih tinggi sebelum mencapai KPA tingkat yang lebih rendah? Buktikan jawaban Anda dengan menggunakan contoh-contoh yang sesuai.
31. Apa inisiatif kualitas Six Sigma? Untuk kategori industri mana itu berlaku? Jelaskan teknik Six Sigma yang diadopsi oleh organisasi perangkat lunak sehubungan dengan tujuan, prosedur, dan hasilnya
32. Apa perbedaan antara metrik proses dan metrik produk? Berikan masing-masing empat contoh.
33. Sebuah sistem perangkat lunak terdiri dari 50 modul. Setiap modul dijamin memiliki keandalan R tidak kurang dari 0,999. Apa yang akan menjadi kasus terbaik dan keandalan seluruh sistem? Apa yang harus menjadi keandalan modul jika kita mengharuskan sistem menunjukkan keandalan yang sama dengan 0,99999?
34. Jelaskan pentingnya manajemen konfigurasi perangkat lunak dalam paradigma kualitas modern seperti SEI CMM dan ISO 9001. Organisasi yang tidak menggunakan alat manajemen konfigurasi apa pun dapat memenuhi syarat untuk tingkat SEI CMM yang mana?
35. Sebutkan empat metrik yang dapat ditentukan dari analisis kode sumber program dan akan berkorelasi baik dengan keandalan perangkat lunak yang dikirimkan.
36. Diskusikan fitur yang menonjol dari struktur pelaporan organisasi kelompok SQA seperti yang direkomendasikan oleh SEI CMM dan ISO 9001. Apa alasan di balik memiliki struktur pelaporan seperti itu?
37. Sarankan dua organisasi pengembangan yang model kapabilitas SEI-nya tidak mungkin sesuai. Berikan alasan mengapa hal ini terjadi.

38. Apa yang Anda pahami tentang Key Process Area (KPA), dalam konteks SEI CMM? Apakah organisasi akan menghadapi masalah, jika mencoba menerapkan KPA CMM SEI tingkat yang lebih tinggi sebelum mencapai KPA tingkat yang lebih rendah? Buktikan jawaban Anda dengan menggunakan contoh-contoh yang sesuai.
39. Bisakah sebuah program benar dan masih belum menunjukkan kualitas yang baik? Menjelaskan.
40. Apa yang Anda pahami tentang pencegahan cacat? Jelaskan bagaimana pencegahan cacat dapat dicapai.

BAB 12
TEKNIK PERANGKAT LUNAK BERBANTUAN KOMPUTER

Dalam bab ini, kita akan membahas tentang rekayasa perangkat lunak berbantuan komputer (CASE) dan bagaimana penggunaan alat CASE membantu meningkatkan upaya pengembangan perangkat lunak dan upaya pemeliharaan. Akhir-akhir ini, CASE telah muncul sebagai topik yang banyak dibicarakan di industri perangkat lunak. Perangkat lunak menjadi komponen paling mahal dalam instalasi komputer mana pun. Meskipun harga perangkat keras terus turun seperti tidak pernah dan jatuh di bawah harapan yang paling optimis sekalipun, harga perangkat lunak menjadi lebih mahal karena meningkatnya biaya tenaga kerja. Skenario ini membuat sebagian besar manajer khawatir. Dalam adegan ini, alat CASE menjanjikan upaya dan pengurangan biaya dalam pengembangan dan pemeliharaan perangkat lunak. Oleh karena itu, penyebaran dan pengembangan alat CASE telah menjadi subjek utama bagi sebagian besar manajer proyek perangkat lunak. Untuk insinyur perangkat lunak, alat CASE berjanji untuk menghilangkan pekerjaan rutin yang membosankan, dan membantu mengembangkan produk berkualitas lebih baik secara lebih efisien. Dengan pengenalan singkat dan motivasi untuk mempelajari alat CASE ini, pertama-tama kita akan mendefinisikan ruang lingkup CASE dan memeriksa berbagai konsep yang terkait dengan CASE. Selanjutnya, kita akan membahas fitur dari berbagai jenis alat CASE.

12.1 KASUS DAN RUANG LINGKUPNYA

Pertama-tama kita perlu mendefinisikan apa itu alat CASE dan apa itu lingkungan CASE. Alat CASE adalah istilah umum yang digunakan untuk menunjukkan segala bentuk dukungan otomatis untuk rekayasa perangkat lunak. Dalam arti yang lebih terbatas, alat CASE dapat berarti alat apa pun yang digunakan untuk mengotomatisasi beberapa aktivitas yang terkait dengan pengembangan perangkat lunak. Banyak alat CASE sekarang tersedia. Beberapa alat ini membantu dalam tugas terkait fase seperti spesifikasi, analisis terstruktur, desain, pengkodean, pengujian, dll. dan lainnya untuk aktivitas non-fase seperti manajemen proyek dan manajemen konfigurasi. Tujuan utama dalam menggunakan alat CASE apa pun adalah:

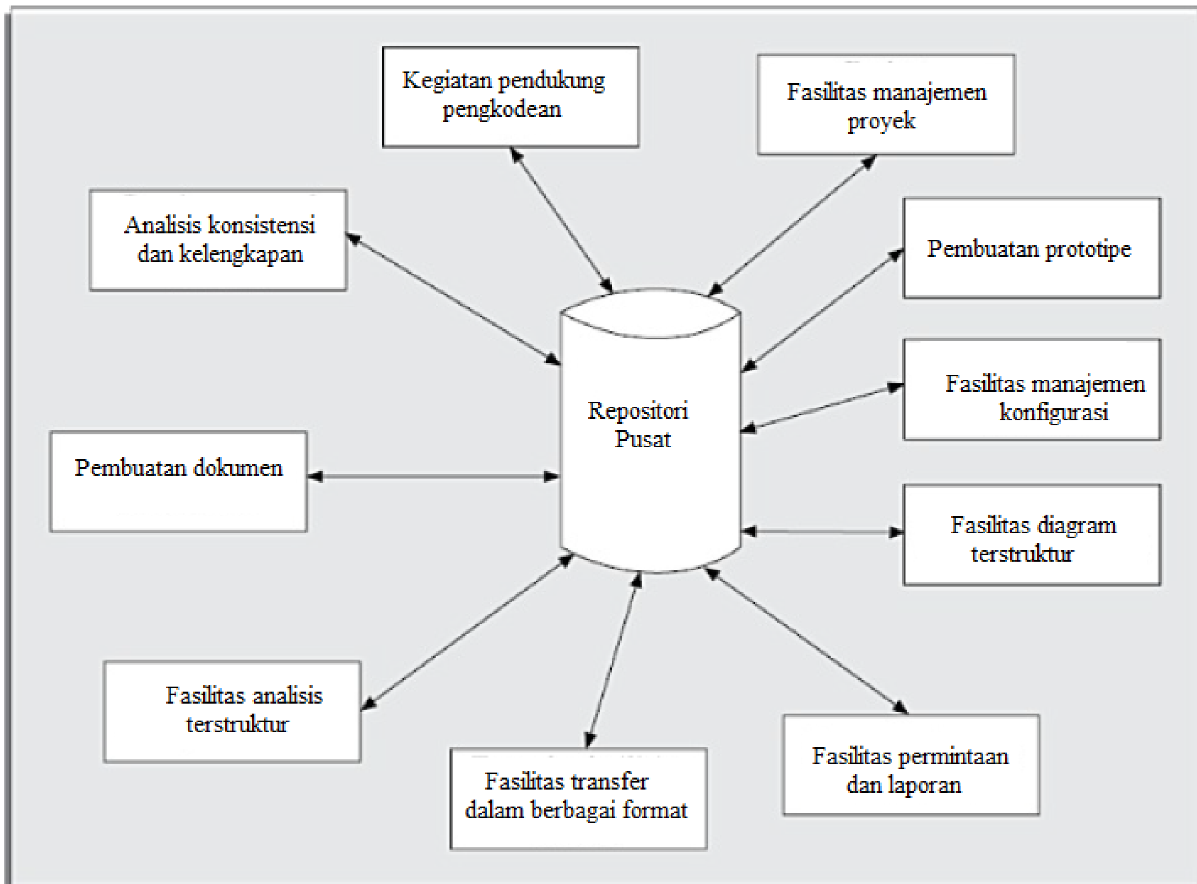
- Untuk meningkatkan produktivitas.
- Untuk membantu menghasilkan perangkat lunak berkualitas lebih baik dengan biaya lebih rendah.

12.2 LINGKUNGAN KASUS

Meskipun alat CASE individu berguna, kekuatan sebenarnya dari seperangkat alat dapat direalisasikan hanya ketika seperangkat alat ini diintegrasikan ke dalam kerangka kerja atau lingkungan umum. Jika alat CASE yang berbeda tidak terintegrasi, maka data yang dihasilkan oleh satu alat harus dimasukkan ke alat lainnya. Ini mungkin juga melibatkan konversi format karena alat yang dikembangkan oleh vendor yang berbeda cenderung menggunakan format yang berbeda. Ini menghasilkan upaya tambahan untuk mengekspor data dari satu alat dan mengimpor ke alat lain. Selain itu, banyak alat tidak mengizinkan ekspor data dan memelihara data dalam format kepemilikan.

Alat CASE dicirikan oleh tahapan atau tahapan siklus hidup pengembangan perangkat lunak yang menjadi fokusnya. Karena alat yang berbeda yang mencakup tahapan yang berbeda berbagi informasi umum, diperlukan bahwa alat tersebut berintegrasi melalui beberapa repositori pusat untuk memiliki pandangan yang konsisten tentang informasi yang terkait dengan perangkat lunak. Repositori pusat ini biasanya berupa kamus data yang berisi

definisi semua item data gabungan dan dasar. Melalui repositori pusat, semua alat CASE dalam lingkungan CASE berbagi informasi umum di antara mereka sendiri. Dengan demikian lingkungan CASE memfasilitasi otomatisasi metodologi langkah demi langkah untuk pengembangan perangkat lunak. Berbeda dengan lingkungan CASE, lingkungan pemrograman adalah kumpulan alat yang terintegrasi untuk mendukung hanya fase pengkodean pengembangan perangkat lunak. Alat yang umumnya terintegrasi dalam lingkungan pemrograman adalah editor teks, kompiler, dan debugger. Alat-alat yang berbeda terintegrasi sejauh setelah kompiler mendeteksi kesalahan, editor secara otomatis membuka pernyataan yang salah dan pernyataan kesalahan disorot. Contoh lingkungan pemrograman yang populer adalah lingkungan Turbo C, Visual Basic, Visual C++, dll. Representasi skematis dari lingkungan CASE ditunjukkan pada Gambar 12.1.



Gambar 12.1 Lingkungan KASUS

Lingkungan pemrograman standar seperti Turbo C, Visual C++, dll. dilengkapi dengan editor program, compiler, debugger, linker, dll., Semua alat ini terintegrasi. Jika Anda mengklik kesalahan yang dilaporkan oleh kompiler, tidak hanya membawa Anda ke editor, tetapi juga membawa kursor ke baris atau pernyataan tertentu yang menyebabkan kesalahan.

Manfaat CASE

Beberapa manfaat diperoleh dari penggunaan lingkungan CASE atau bahkan alat CASE yang terisolasi. Mari kita periksa beberapa manfaat ini:

- Manfaat utama yang timbul dari penggunaan lingkungan CASE adalah penghematan biaya melalui semua fase pengembangan. Studi yang berbeda dilakukan untuk mengukur dampak CASE, menempatkan pengurangan upaya antara 30 persen dan 40 persen.

- Penggunaan alat CASE mengarah pada peningkatan kualitas yang cukup besar. Hal ini terutama disebabkan oleh fakta bahwa seseorang dapat dengan mudah beralih melalui berbagai fase pengembangan perangkat lunak, dan kemungkinan kesalahan manusia sangat berkurang.
- Alat CASE membantu menghasilkan dokumen berkualitas tinggi dan konsisten. Karena data penting yang berkaitan dengan produk perangkat lunak disimpan di repositori pusat, redundansi dalam data yang disimpan berkurang, dan oleh karena itu, kemungkinan dokumentasi yang tidak konsisten sangat berkurang.
- Alat CASE menghilangkan sebagian besar pekerjaan yang membosankan dalam pekerjaan insinyur perangkat lunak. Misalnya, mereka tidak perlu memeriksa keseimbangan DFD dengan cermat, tetapi dapat melakukannya dengan mudah hanya dengan menekan sebuah tombol.
- Alat CASE telah menghasilkan penghematan biaya yang revolusioner dalam upaya pemeliharaan perangkat lunak. Ini muncul tidak hanya karena nilai luar biasa dari lingkungan CASE dalam keterlacakan dan pemeriksaan konsistensi, tetapi juga karena penangkapan informasi yang sistematis selama berbagai fase pengembangan perangkat lunak sebagai akibat dari mengikuti lingkungan CASE.
- Pengenalan lingkungan CASE berdampak pada gaya kerja perusahaan, dan membuatnya berorientasi pada pendekatan terstruktur dan teratur.

12.3 DUKUNGAN KASUS DALAM SIKLUS HIDUP PERANGKAT LUNAK

Mari kita periksa berbagai jenis dukungan yang disediakan CASE selama fase yang berbeda dari siklus hidup perangkat lunak. Alat CASE harus mendukung metodologi pengembangan, membantu menegakkan hal yang sama, dan menyediakan sejumlah pemeriksaan konsistensi antara fase yang berbeda. Beberapa kemungkinan dukungan yang biasanya disediakan oleh alat CASE dalam siklus hidup pengembangan perangkat lunak dibahas di bawah ini.

Dukungan Prototipe

Kita telah melihat bahwa pembuatan prototipe berguna untuk memahami persyaratan produk perangkat lunak yang kompleks, untuk mendemonstrasikan sebuah konsep, untuk memasarkan ide-ide baru, dan sebagainya. Persyaratan alat CASE prototyping adalah sebagai berikut:

- Tentukan interaksi pengguna.
- Tentukan aliran kontrol sistem.
- Menyimpan dan mengambil data yang dibutuhkan oleh sistem.
- Menggabungkan beberapa logika pemrosesan.

Ada beberapa alat prototyping yang berdiri sendiri. Tetapi alat yang terintegrasi dengan kamus data dapat menggunakan entri dalam kamus data, membantu mengisi kamus data dan memastikan konsistensi antara data desain dan prototipe. Alat prototyping yang baik harus mendukung fitur-fitur berikut:

- Karena salah satu kegunaan utama alat CASE prototyping adalah pengembangan antarmuka pengguna grafis (GUI), alat CASE prototyping harus mendukung pengguna untuk membuat GUI menggunakan editor grafis. Pengguna harus diizinkan untuk menentukan semua formulir entri data, menu, dan kontrol.
- Itu harus terintegrasi dengan kamus data lingkungan CASE.
- Jika memungkinkan, itu harus dapat berintegrasi dengan modul yang ditentukan pengguna eksternal yang ditulis dalam C atau beberapa bahasa pemrograman tingkat tinggi yang populer.

- Pengguna harus dapat menentukan urutan status di mana prototipe yang dibuat dapat berjalan. Pengguna juga harus diizinkan untuk mengontrol jalannya prototipe.
- Prototipe sistem run time harus mendukung mock up run dari sistem aktual dan pengelolaan data input dan output.

Analisis dan Desain Terstruktur

Beberapa teknik diagram digunakan untuk analisis terstruktur dan desain terstruktur. Alat CASE harus mendukung satu atau lebih dari analisis terstruktur dan teknik desain. Alat CASE harus mendukung analisis gambar dan diagram desain dengan mudah. Alat CASE harus mendukung menggambar diagram yang cukup kompleks dan sebaiknya melalui hierarki level. Ini harus menyediakan navigasi yang mudah melalui tingkat yang berbeda dan melalui desain dan analisis. Alat tersebut harus mendukung pemeriksaan kelengkapan dan konsistensi di seluruh desain dan analisis dan melalui semua tingkat hierarki analisis. Jika memungkinkan, sistem harus melarang operasi yang tidak konsisten, tetapi mungkin sangat sulit untuk mengimplementasikan fitur seperti itu. Setiap kali ada beban komputasi yang berat saat pemeriksaan konsistensi, pemeriksaan konsistensi harus dapat dinonaktifkan untuk sementara.

Pembuatan Kode

Sejauh menyangkut pembuatan kode, harapan umum dari alat CASE cukup rendah. Persyaratan yang masuk akal adalah ketertelusuran dari file sumber ke data desain. Dukungan lebih pragmatis yang diharapkan dari alat CASE selama fase pembuatan kode adalah sebagai berikut:

- Alat CASE harus mendukung pembuatan kerangka modul atau templat dalam satu atau lebih bahasa populer. Seharusnya dimungkinkan untuk memasukkan pesan hak cipta, deskripsi singkat modul, nama penulis dan tanggal pembuatan dalam beberapa format yang dapat dipilih.
- Alat harus menghasilkan catatan, struktur, definisi kelas secara otomatis dari isi kamus data dalam satu atau lebih bahasa pemrograman populer.
- Ini harus menghasilkan tabel database untuk sistem manajemen database relasional.
- Alat harus menghasilkan kode untuk antarmuka pengguna dari definisi prototipe untuk jendela X dan aplikasi berbasis jendela MS.

Generator Kasus Uji

Alat CASE untuk pembuatan kasus uji harus memiliki fitur berikut:

- Itu harus mendukung pengujian desain dan persyaratan
- Ini harus menghasilkan laporan set pengujian dalam format ASCII yang dapat langsung diimpor ke dalam dokumen rencana pengujian.

12.4 KARAKTERISTIK ALAT KASUS LAINNYA

Karakteristik yang tercantum di bagian ini bukanlah inti dari fungsionalitas alat CASE tetapi secara signifikan meningkatkan efektivitas dan kegunaan alat CASE.

Persyaratan Perangkat Keras dan Lingkungan

Dalam kebanyakan kasus, itu adalah perangkat keras yang ada yang akan menempatkan kendala pada pemilihan alat CASE. Jadi, alih-alih mendefinisikan persyaratan perangkat keras untuk alat CASE, tugas yang ada menjadi sesuai dengan konfigurasi alat CASE yang optimal dalam kemampuan perangkat keras yang ada. Oleh karena itu, kita harus menekankan pada pemilihan konfigurasi alat CASE yang paling optimal untuk konfigurasi perangkat keras yang diberikan.

Jaringan heterogen adalah salah satu contoh lingkungan terdistribusi dan ini sebagai ilustrasi karena lebih populer karena fitur mesinnya yang independen. Implementasi CASE tool pada jaringan heterogen menggunakan paradigma client-server. Beberapa klien yang menjalankan modul yang berbeda mengakses kamus data melalui server ini. Server kamus data mungkin mendukung satu atau lebih proyek. Meskipun dimungkinkan untuk menjalankan banyak server untuk proyek yang berbeda tetapi implementasi kamus data yang didistribusikan tidak umum. Kumpulan alat terintegrasi melalui kamus data yang mendukung banyak proyek, banyak pengguna yang bekerja secara bersamaan dan memungkinkan untuk berbagi informasi antara pengguna dan proyek. Kamus data memberikan tampilan yang konsisten dari semua entitas proyek, misalnya, definisi catatan data dan diagram hubungan entitasnya menjadi konsisten. Server harus menggambarkan tampilan logis per proyek dari kamus data. Ini berarti harus memungkinkan pencadangan/pemulihan, penyalinan, pembersihan bagian dari kamus data, dll. Alat harus bekerja dengan memuaskan untuk jumlah maksimum pengguna yang bekerja secara bersamaan. Alat harus mendukung lingkungan multiwindow bagi pengguna. Ini penting untuk memungkinkan pengguna melihat lebih dari satu diagram sekaligus. Ini juga memfasilitasi navigasi dan beralih dari satu bagian ke bagian lainnya.

Dukungan Dokumentasi

Dokumen yang dapat dikirim harus diatur secara grafis dan harus dapat menggabungkan teks dan diagram dari repositori pusat. Ini membantu dalam menghasilkan dokumentasi yang up-to-date. Alat CASE harus terintegrasi dengan satu atau lebih paket penerbitan desktop yang tersedia secara komersial. Seharusnya dimungkinkan untuk mengekspor teks, grafik, tabel, laporan kamus data ke paket DTP dalam bentuk standar seperti PostScript.

Manajemen proyek

Ini harus mendukung pengumpulan, penyimpanan, dan analisis informasi tentang kemajuan proyek perangkat lunak seperti perkiraan durasi tugas, awal tugas terjadwal dan aktual, tanggal penyelesaian, tanggal dan hasil tinjauan, dll.

Antarmuka Eksternal

Alat harus memungkinkan pertukaran informasi untuk penggunaan kembali desain. Informasi yang akan diekspor oleh alat sebaiknya dalam format ASCII dan mendukung arsitektur terbuka. Demikian pula, kamus data harus menyediakan antarmuka pemrograman untuk mengakses informasi. Hal ini diperlukan untuk integrasi utilitas kustom, membangun teknik baru, atau mengisi kamus data.

Dukungan Rekayasa Terbalik

Alat ini harus mendukung pembuatan bagan struktur dan kamus data dari kode sumber yang ada. Itu harus mengisi kamus data dari kode sumber. Jika alat tersebut digunakan untuk merekayasa ulang sistem informasi, alat tersebut harus berisi alat konversi dari struktur file sekuensial yang diindeks, basis data hierarkis dan jaringan ke sistem basis data relasional.

Antarmuka Kamus Data

Antarmuka kamus data harus menyediakan akses tampilan dan pembaruan ke entitas dan relasi yang tersimpan di dalamnya. Itu harus memiliki fasilitas cetak untuk mendapatkan hard copy dari layar yang dilihat. Itu harus memberikan laporan analisis seperti referensi silang, analisis dampak, dll. Idealnya, itu harus mendukung bahasa kueri untuk melihat isinya.

Tutorial dan Bantuan

Penerapan alat CASE dan dengan demikian keberhasilannya bergantung pada kemampuan pengguna untuk menggunakan semua fitur yang didukung secara efektif. Oleh karena itu, bagi pengguna yang belum tahu, tutorial sangat penting. Tutorial tidak boleh

terbatas pada pengajaran bagian antarmuka pengguna saja, tetapi harus mencakup poin-poin berikut secara komprehensif:

- Tutorial harus mencakup semua teknik dan fasilitas melalui bagian yang diklasifikasikan secara logis.
- Tutorial harus didukung oleh dokumentasi yang tepat.

12.5 MENUJU ALAT KASUS GENERASI KEDUA

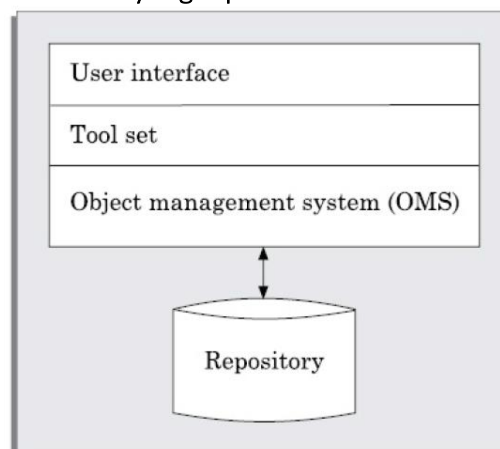
Fitur penting dari alat CASE generasi kedua adalah dukungan langsung dari setiap metodologi yang diadaptasi. Ini akan memerlukan fungsi administrator CASE untuk setiap organisasi, yang dapat menyesuaikan alat CASE dengan metodologi tertentu. Selain itu, fitur di alat CASE generasi kedua adalah sebaga berikut:

Dukungan diagram cerdas: Fakta bahwa teknik diagram berguna untuk analisis dan desain sistem sudah mapan. Alat CASE masa depan akan memberikan bantuan untuk tata letak diagram secara estetis dan otomatis.

Integrasi dengan lingkungan implementasi: Alat CASE harus menyediakan integrasi antara desain dan implementasi.

Standar kamus data: Pengguna harus diizinkan untuk mengintegrasikan banyak alat pengembangan ke dalam satu lingkungan. Sangat tidak mungkin bahwa satu vendor akan dapat memberikan solusi total. Selain itu, alat yang disukai akan membutuhkan penyetelan untuk sistem tertentu. Dengan demikian pengguna akan bertindak sebagai integrator sistem. Ini hanya mungkin jika beberapa standar pada kamus data muncul.

Dukungan kustomisasi: Pengguna harus diizinkan untuk menentukan jenis objek dan koneksi baru. Fasilitas ini dapat digunakan untuk membangun beberapa metodologi khusus. Idealnya harus memungkinkan untuk menentukan aturan metodologi ke mesin aturan untuk melakukan pemeriksaan konsistensi yang diperlukan.



Gambar 12.2 Arsitektur lingkungan CASE modern

12.6 ARSITEKTUR LINGKUNGAN KASUS

Arsitektur lingkungan CASE modern yang khas ditunjukkan secara diagram pada Gambar 12.2. Komponen penting dari lingkungan CASE modern adalah antarmuka pengguna, set alat, sistem manajemen objek (OMS), dan repositori. Kita telah melihat karakteristik dari tool set. Mari kita periksa komponen lain dari lingkungan CASE.

Antarmuka pengguna

Antarmuka pengguna menyediakan kerangka kerja yang konsisten untuk mengakses alat yang berbeda sehingga memudahkan pengguna untuk berinteraksi dengan alat yang berbeda dan mengurangi biaya belajar bagaimana alat yang berbeda digunakan.

Sistem manajemen objek dan repositori

Alat kasus yang berbeda mewakili produk perangkat lunak sebagai satu set entitas seperti spesifikasi, desain, data teks, rencana proyek, dll. Sistem manajemen objek memetakan entitas logis ini ke dalam sistem manajemen penyimpanan (repositori) yang mendasarinya. Sistem manajemen basis data relasional komersial diarahkan untuk mendukung sejumlah besar informasi yang terstruktur sebagai catatan sederhana yang relatif singkat. Ada beberapa jenis entitas tetapi sejumlah besar contoh. Sebaliknya, alat CASE membuat sejumlah besar entitas dan tipe relasi dengan mungkin beberapa contoh masing-masing. Jadi sistem manajemen objek menangani pemetaan entitas ini dengan tepat ke dalam sistem manajemen penyimpanan yang mendasarinya.

12.7 RINGKASAN

- Penggunaan tool CASE menjadi sangat diperlukan untuk proyek software besar di mana tim insinyur perangkat lunak bekerja sama. Tren saat ini adalah menuju tool CASE berbasis workstation terdistribusi.

12.8 LATIHAN

1. Pilih opsi yang benar:
   a. Manakah dari berikut ini yang secara efektif mengintegrasikan alat-alat yang berbeda dalam lingkungan KASUS?
      i. Dokumen spesifikasi persyaratan perangkat lunak (SRS)
      ii. Pusat penyimpanan data
      iii. Kompilasi tambahan
      iv. Intervensi pengguna
   b. Manakah dari alat CASE berikut yang biasanya bukan bagian dari lingkungan pemrograman?
      i. Kompilator
      ii. debugger
      iii. Alat pemodelan
      iv. Redaktur
   c. Manakah dari alat CASE berikut yang biasanya tidak berguna selama kegiatan pemeliharaan korektif?
      i. Alat seleksi uji regresi
      ii. Alat rekayasa terbalik
      iii. Debugger simbolis
      iv. Alat penangkap kebutuhan
2. Apa yang Anda pahami dengan istilah alat KASUS dan lingkungan KASUS? Mengapa alat integrasi meningkatkan kekuatan alat? Jelaskan dengan menggunakan beberapa contoh.
3. Apa itu lingkungan pemrograman?
4. Apa keuntungan utama menggunakan alat CASE?
5. Apa saja fitur penting yang harus didukung oleh alat CASE generasi mendatang?
6. Identifikasi dukungan CASE yang dapat digunakan selama upaya pemeliharaan besar terkait perangkat lunak lama yang besar.
7. Diskusikan peran kamus data dalam lingkungan CASE.
8. Gambarkan arsitektur lingkungan CASE secara skematis dan jelaskan bagaimana berbagai alat terintegrasi.

BAB 13
PEMELIHARAAN PERANGKAT LUNAK

Banyak mahasiswa dan insinyur yang berpraktik memiliki prasangka terhadap pekerjaan pemeliharaan perangkat lunak. Penyebutan kata perawatan memunculkan citra seorang tukang obeng, montir yang memegang dengan tangan kotor memegang sekantong penuh suku cadang. Tujuan bab ini adalah untuk menjernihkan istilah yang salah ini, memberikan beberapa pemahaman intuitif tentang proyek pemeliharaan perangkat lunak, dan untuk membiasakan Anda dengan teknik terbaru dalam pemeliharaan perangkat lunak.

Pemeliharaan perangkat lunak menunjukkan setiap perubahan yang dilakukan pada produk perangkat lunak setelah dikirimkan ke pelanggan. Pemeliharaan tidak dapat dihindari untuk hampir semua jenis produk. Namun, sebagian besar produk memerlukan perawatan karena keausan yang disebabkan oleh penggunaan. Di sisi lain, produk perangkat lunak tidak memerlukan pemeliharaan dalam jumlah ini, tetapi memerlukan pemeliharaan untuk memperbaiki kesalahan, meningkatkan fitur, port ke platform baru, dll.

13.1 KARAKTERISTIK PEMELIHARAAN PERANGKAT LUNAK

Pada bagian ini, pertama-tama kita mengklasifikasikan upaya pemeliharaan yang berbeda ke dalam beberapa kelas. Selanjutnya, kita membahas beberapa karakteristik umum dari proyek pemeliharaan, juga beberapa masalah khusus yang terkait dengan proyek pemeliharaan. Pemeliharaan perangkat lunak menjadi kegiatan penting dari sejumlah besar organisasi. Ini tidak mengherankan, mengingat tingkat keusangan perangkat keras, keabadian produk perangkat lunak itu sendiri, dan permintaan komunitas pengguna untuk melihat produk perangkat lunak yang ada berjalan pada platform yang lebih baru, berjalan di lingkungan yang lebih baru, dan/atau dengan fitur yang disempurnakan. Ketika platform perangkat keras berubah, dan produk perangkat lunak melakukan beberapa fungsi tingkat rendah, pemeliharaan diperlukan. Juga, setiap kali lingkungan dukungan produk perangkat lunak berubah, produk perangkat lunak memerlukan pengerjaan ulang untuk mengatasi antarmuka yang lebih baru. Misalnya, produk perangkat lunak mungkin perlu dipertahankan ketika sistem operasi berubah. Dengan demikian, setiap produk perangkat lunak terus berkembang setelah pengembangannya melalui upaya pemeliharaan.

Jenis Pemeliharaan Perangkat Lunak

Ada tiga jenis pemeliharaan perangkat lunak, yang dijelaskan sebagai berikut:

Korektif: Pemeliharaan korektif dari produk perangkat lunak diperlukan baik untuk memperbaiki bug yang diamati saat sistem sedang digunakan.

Adaptif: Produk perangkat lunak mungkin memerlukan pemeliharaan ketika pelanggan membutuhkan produk untuk berjalan pada platform baru, pada sistem operasi baru, atau ketika mereka membutuhkan produk untuk berinteraksi dengan perangkat keras atau perangkat lunak baru.

Sempurna: Produk perangkat lunak memerlukan pemeliharaan untuk mendukung fitur-fitur baru yang diinginkan pengguna untuk didukung, untuk mengubah fungsionalitas sistem yang berbeda sesuai dengan permintaan pelanggan, atau untuk meningkatkan kinerja sistem.

Karakteristik Evolusi Perangkat Lunak

Lehman dan Belady telah mempelajari karakteristik evolusi produk perangkat lunak sever [1980]. Mereka telah mengungkapkan pengamatan mereka dalam bentuk undang-undang. Hukum penting mereka disajikan dalam sub-bagian berikut. Tetapi peringatan di sini

adalah bahwa ini adalah generalisasi dan mungkin tidak berlaku untuk kasus tertentu dan juga sebagian besar pengamatan ini menyangkut proyek perangkat lunak besar dan mungkin tidak sesuai untuk pemeliharaan dan evolusi produk yang sangat kecil.

Hukum pertama Lehman: Sebuah produk perangkat lunak harus berubah terus menerus atau menjadi semakin kurang berguna. Setiap produk perangkat lunak terus berkembang setelah pengembangannya melalui upaya pemeliharaan. Produk yang lebih besar tetap beroperasi untuk waktu yang lebih lama karena biaya penggantian yang lebih tinggi dan oleh karena itu cenderung memerlukan upaya pemeliharaan yang lebih tinggi. Undang-undang ini dengan jelas menunjukkan bahwa setiap produk terlepas dari seberapa baik dirancang harus menjalani perawatan. Padahal, ketika suatu produk tidak memerlukan perawatan lagi, itu adalah tanda bahwa produk tersebut akan segera dihentikan/dibuang. Ini berbeda dengan intuisi umum bahwa hanya produk yang dirancang dengan buruk yang membutuhkan perawatan. Padahal, produk yang baik dipertahankan dan produk yang buruk dibuang.

Hukum kedua Lehman: Struktur suatu program cenderung menurun karena semakin banyak pemeliharaan yang dilakukan padanya. Alasan untuk struktur yang terdegradasi adalah ketika Anda menambahkan fungsi selama pemeliharaan, Anda membangun di atas program yang sudah ada, seringkali dengan cara yang tidak dimaksudkan untuk didukung oleh program yang ada. Jika Anda tidak mendesain ulang sistem, penambahannya akan lebih kompleks dari yang seharusnya. Karena solusi perbaikan cepat, selain degradasi struktur, dokumentasi menjadi tidak konsisten dan menjadi kurang bermanfaat karena semakin banyak pemeliharaan yang dilakukan.

Hukum ketiga Lehman: Selama masa hidup program, laju perkembangannya kira-kira konstan. Tingkat perkembangan dapat diukur dalam bentuk baris kode yang ditulis atau dimodifikasi. Oleh karena itu hukum ini menyatakan bahwa kecepatan penulisan atau modifikasi kode kira-kira sama selama pengembangan dan pemeliharaan.

Masalah Khusus Terkait dengan Pemeliharaan Perangkat Lunak

Pekerjaan pemeliharaan perangkat lunak saat ini biasanya jauh lebih mahal daripada yang seharusnya dan membutuhkan lebih banyak waktu daripada yang dibutuhkan. Alasan untuk situasi ini adalah sebagai berikut: Pekerjaan pemeliharaan perangkat lunak dalam organisasi sebagian besar dilakukan dengan menggunakan teknik ad hoc. Alasan utamanya adalah bahwa pemeliharaan perangkat lunak adalah salah satu bidang rekayasa perangkat lunak yang paling diabaikan. Meskipun pemeliharaan perangkat lunak dengan cepat menjadi bidang pekerjaan yang penting bagi banyak perusahaan sebagai produk perangkat lunak dari masa lalu, masih pemeliharaan perangkat lunak sebagian besar dilakukan sebagai operasi pemadam kebakaran, bukan melalui kegiatan sistematis dan terencana.

Pemeliharaan perangkat lunak memiliki citra yang sangat buruk di industri. Oleh karena itu, sebuah organisasi seringkali tidak dapat mempekerjakan insinyur yang cerdas untuk melakukan pekerjaan pemeliharaan. Meskipun pemeliharaan menderita citra yang buruk, pekerjaan yang terlibat seringkali lebih menantang daripada pekerjaan pengembangan. Selama pemeliharaan, perlu untuk benar-benar memahami pekerjaan orang lain, dan kemudian melakukan modifikasi dan ekstensi yang diperlukan.

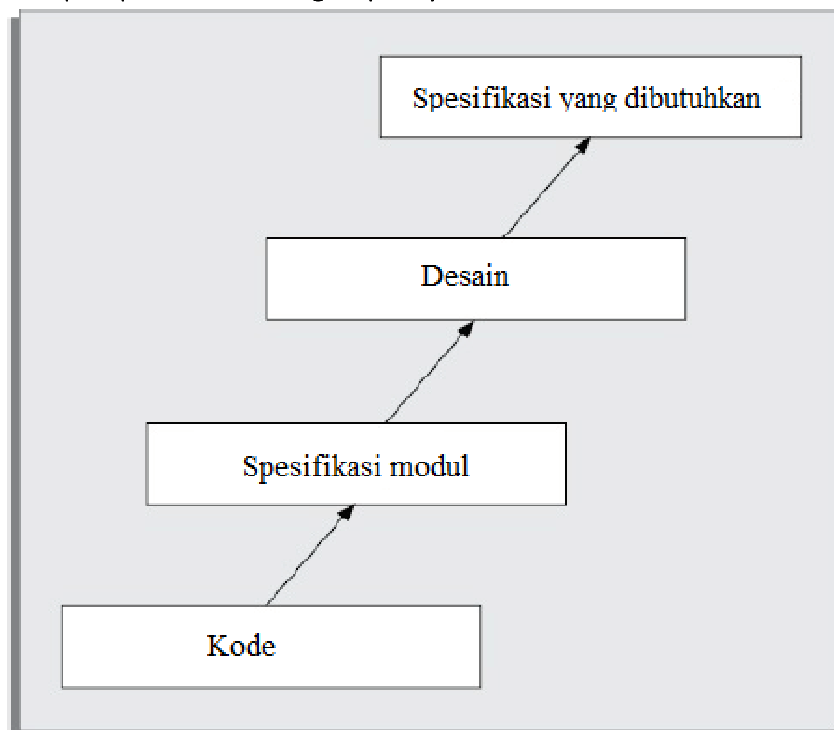
Masalah lain yang terkait dengan pekerjaan pemeliharaan adalah bahwa sebagian besar produk perangkat lunak yang memerlukan pemeliharaan adalah produk lama. Meskipun kata warisan menyiratkan perangkat lunak "tua", tetapi tidak ada kesepakatan tentang apa sebenarnya sistem warisan. Adalah bijaksana untuk mendefinisikan sistem warisan sebagai sistem perangkat lunak apa pun yang sulit untuk dipelihara. Masalah khas yang terkait dengan sistem warisan adalah dokumentasi yang buruk, tidak terstruktur (kode spageti dengan

struktur kontrol yang buruk), dan kurangnya personel yang memiliki pengetahuan tentang produk. Banyak sistem warisan dikembangkan sejak lama. Namun, ada kemungkinan bahwa sistem yang baru dikembangkan dengan desain dan dokumentasi yang buruk dapat dianggap sebagai sistem warisan.

13.2 TEKNIK PEMBALIK PERANGKAT LUNAK

Rekayasa balik perangkat lunak adalah proses memulihkan desain dan spesifikasi persyaratan suatu produk dari analisis kodenya. Tujuan dari reverse engineering adalah untuk memfasilitasi pekerjaan pemeliharaan dengan meningkatkan pemahaman sistem dan untuk menghasilkan dokumen yang diperlukan untuk sistem warisan. Rekayasa terbalik menjadi penting, karena produk perangkat lunak lama tidak memiliki dokumentasi yang tepat, dan sangat tidak terstruktur. Bahkan produk yang dirancang dengan baik menjadi perangkat lunak warisan karena strukturnya menurun melalui serangkaian upaya pemeliharaan.

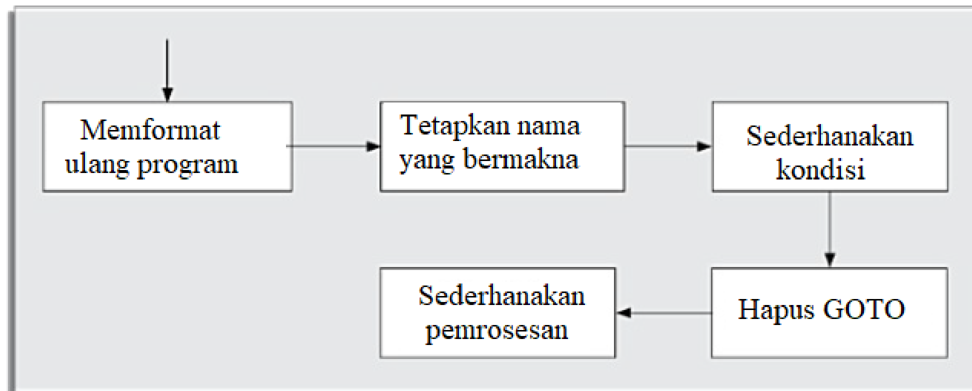
Tahap pertama dari reverse engineering biasanya berfokus pada melakukan perubahan kosmetik pada kode untuk meningkatkan keterbacaan, struktur, dan pemahamannya, tanpa mengubah fungsinya. Cara untuk melakukan perubahan kosmetik ini ditunjukkan secara skematis pada Gambar 13.1. Sebuah program dapat diformat ulang menggunakan salah satu dari beberapa program printer cantik yang tersedia yang mengatur tata letak program dengan rapi. Banyak produk perangkat lunak lama sulit dipahami dengan struktur kontrol yang kompleks dan nama variabel yang tidak dipikirkan. Menetapkan nama variabel yang bermakna adalah penting karena kita telah melihat di Bab 9 bahwa nama variabel yang bermakna adalah dokumentasi kode yang paling membantu. Semua variabel, struktur data, dan fungsi harus diberi nama yang bermakna sedapat mungkin. Kondisi bersarang kompleks dalam program dapat diganti dengan pernyataan kondisional yang lebih sederhana atau kapan pun sesuai dengan pernyataan kasus.



Gambar 13.1 Model proses untuk rekayasa balik

Setelah perubahan kosmetik dilakukan pada perangkat lunak lama, proses ekstraksi kode, desain, dan spesifikasi persyaratan dapat dimulai. Kegiatan ini secara skematis

ditunjukkan pada Gambar 13.2. Untuk mengekstrak desain, pemahaman penuh tentang kode diperlukan. Beberapa alat otomatis dapat digunakan untuk menurunkan aliran data dan diagram aliran kontrol dari kode. Bagan struktur (urutan pemanggilan modul dan pertukaran data antar modul) juga harus diekstraksi. Dokumen SRS dapat ditulis setelah kode lengkap dipahami secara menyeluruh dan desain diekstraksi.



Gambar 13.2 Perubahan kosmetik yang dilakukan sebelum rekayasa balik

13.3 MODEL PROSES PEMELIHARAAN PERANGKAT LUNAK

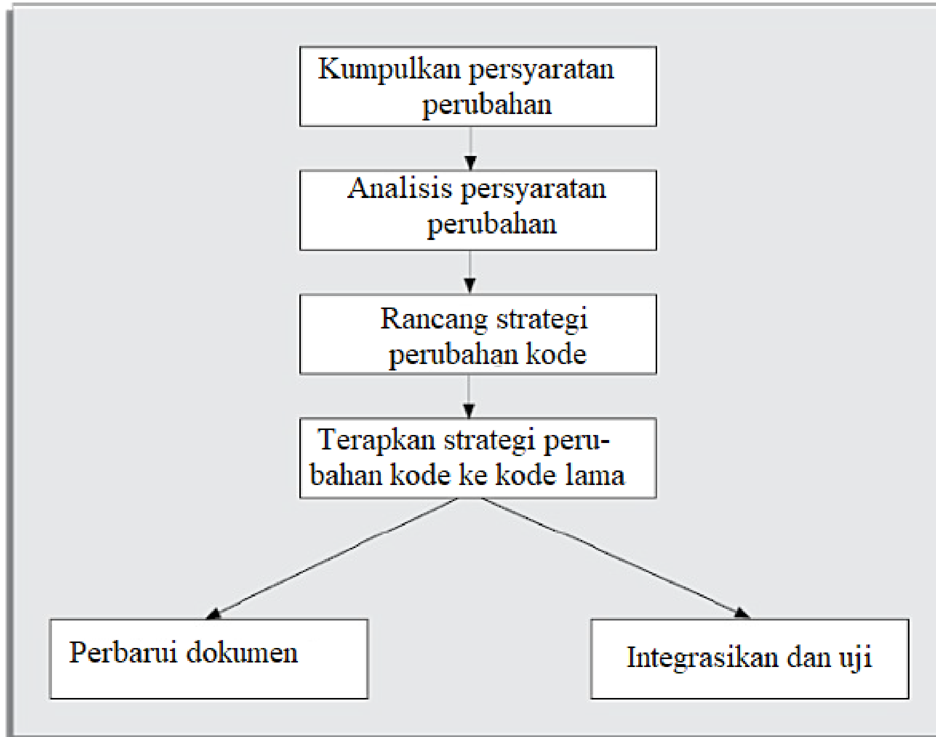
Sebelum membahas model proses untuk pemeliharaan perangkat lunak, kita perlu menganalisis berbagai aktivitas yang terlibat dalam proyek pemeliharaan perangkat lunak yang khas. Kegiatan yang terlibat dalam proyek pemeliharaan perangkat lunak tidak unik dan tergantung pada beberapa faktor seperti: (i) tingkat modifikasi produk yang diperlukan, (ii) sumber daya yang tersedia untuk tim pemeliharaan, (iii) kondisi yang ada produk (misalnya, seberapa terstrukturnya, seberapa baik didokumentasikan, dll.), (iii) risiko proyek yang diharapkan, dll. Ketika perubahan yang diperlukan pada produk perangkat lunak kecil dan langsung, kode dapat langsung dimodifikasi dan perubahan tepat tercermin dalam semua dokumen.

Namun, kegiatan yang lebih rumit diperlukan ketika perubahan yang diperlukan tidak begitu sepele. Biasanya, untuk proyek pemeliharaan kompleks untuk sistem warisan, proses perangkat lunak dapat diwakili oleh siklus rekayasa balik diikuti oleh siklus rekayasa maju dengan penekanan pada penggunaan kembali sebanyak mungkin dari kode yang ada dan dokumen lainnya. Karena ruang lingkup (aktivitas yang diperlukan) untuk proyek pemeliharaan yang berbeda sangat bervariasi, tidak ada model proses pemeliharaan tunggal yang dapat dikembangkan agar sesuai dengan setiap jenis proyek pemeliharaan. Namun, dua kategori besar model proses dapat diusulkan.

Model pertama

Model pertama lebih disukai untuk proyek yang melibatkan pengerjaan ulang kecil di mana kode diubah secara langsung dan perubahan tersebut tercermin dalam dokumen yang relevan nanti. Proses pemeliharaan ini secara grafis disajikan pada Gambar 13.3. Dalam pendekatan ini, proyek dimulai dengan mengumpulkan persyaratan untuk perubahan. Persyaratan selanjutnya dianalisis untuk merumuskan strategi yang akan diadopsi untuk perubahan kode. Pada tahap ini, asosiasi setidaknya beberapa anggota tim pengembangan asli sangat membantu dalam mengurangi waktu siklus, terutama untuk proyek-proyek yang melibatkan kode yang tidak terstruktur dan tidak terdokumentasi secara memadai. Ketersediaan sistem lama yang berfungsi untuk teknisi pemeliharaan di lokasi pemeliharaan sangat memudahkan tugas tim pemeliharaan karena mereka mendapatkan wawasan yang baik tentang cara kerja sistem lama dan juga dapat membandingkan cara kerja sistem yang

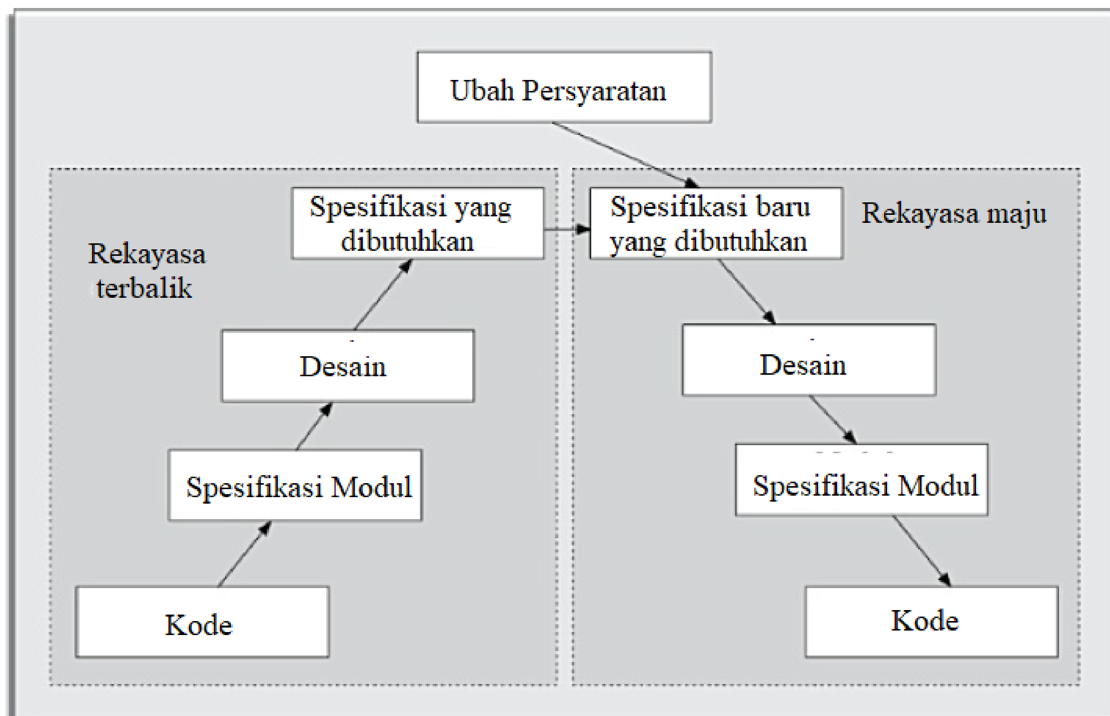
dimodifikasi dengan sistem lama. Juga, debugging sistem yang direkayasa ulang menjadi lebih mudah karena jejak program dari kedua sistem dapat dibandingkan untuk melokalisasi bug.



Gambar 13.3 Model proses perawatan 1.

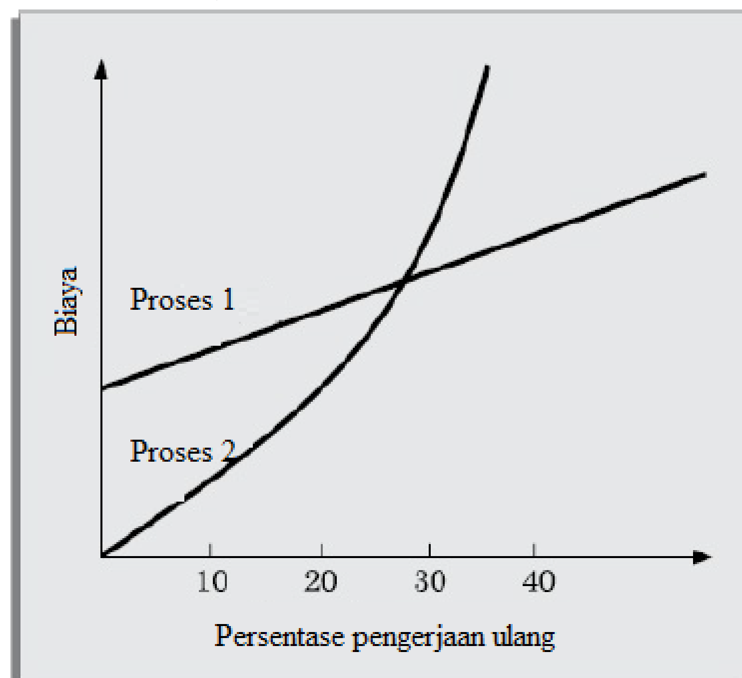
Model kedua

Model kedua lebih disukai untuk proyek-proyek di mana jumlah pengerjaan ulang yang dibutuhkan signifikan. Pendekatan ini dapat diwakili oleh siklus rekayasa balik diikuti oleh siklus rekayasa maju. Pendekatan seperti itu juga dikenal sebagai rekayasa ulang perangkat lunak. Model proses ini digambarkan pada Gambar 13.4.



Gambar 13.4 Model proses perawatan 2.

Siklus rekayasa balik diperlukan untuk produk lama. Selama rekayasa balik, kode lama dianalisis (disarikan) untuk mengekstrak spesifikasi modul. Spesifikasi modul kemudian dianalisis untuk menghasilkan desain. Desain dianalisis (disarikan) untuk menghasilkan spesifikasi kebutuhan asli. Permintaan perubahan kemudian diterapkan ke spesifikasi persyaratan ini untuk sampai pada spesifikasi persyaratan baru. Pada titik ini rekayasa maju dilakukan untuk menghasilkan kode baru. Pada desain, spesifikasi modul, dan pengkodean, penggunaan kembali substansial dibuat dari produk rekayasa balik. Keuntungan penting dari pendekatan ini adalah menghasilkan desain yang lebih terstruktur dibandingkan dengan apa yang dimiliki produk asli, menghasilkan dokumentasi yang baik, dan sangat sering menghasilkan peningkatan efisiensi. Peningkatan efisiensi dibawa oleh desain yang lebih efisien. Namun, pendekatan ini lebih mahal daripada pendekatan pertama. Sebuah studi empiris menunjukkan bahwa proses 1 lebih disukai ketika jumlah pengerjaan ulang tidak lebih dari 15 persen (lihat Gambar 13.5).



Gambar 13.5 Estimasi empiris biaya pemeliharaan versus persentase pengerjaan ulang

Selain jumlah pengerjaan ulang, beberapa faktor lain mungkin mempengaruhi keputusan mengenai penggunaan model proses 1 di atas model proses 2 sebagai berikut:

- Rekayasa ulang mungkin lebih disukai untuk produk yang menunjukkan tingkat kegagalan yang tinggi.
- Rekayasa ulang mungkin juga lebih disukai untuk produk lama yang memiliki desain dan struktur kode yang buruk.

13.4 ESTIMASI BIAYA PERAWATAN

Upaya pemeliharaan membutuhkan sekitar 60 persen dari total biaya siklus hidup untuk produk perangkat lunak yang khas. Namun, biaya pemeliharaan sangat bervariasi dari satu domain aplikasi ke domain lainnya. Untuk sistem tertanam, biaya pemeliharaan dapat mencapai 2 hingga 4 kali lipat biaya pengembangan. Boehm [1981] mengusulkan formula untuk memperkirakan biaya pemeliharaan sebagai bagian dari model estimasi biaya COCOMO-nya. Estimasi biaya pemeliharaan Boehm dibuat dalam besaran yang disebut

dengan annual change traffic (ACT). Boehm mendefinisikan ACT sebagai bagian dari instruksi sumber produk perangkat lunak yang mengalami perubahan selama tahun tertentu baik melalui penambahan atau penghapusan.

$$ACT = \frac{KLOC_{addedd} + KLOC_{deleted}}{KLOC_{total}}$$

di mana, KLOCadded adalah total kilo baris kode sumber yang ditambahkan selama pemeliharaan. KLOCdeleted adalah total KLOC yang dihapus selama maintenance. Dengan demikian, kode yang diubah, harus diperhitungkan baik kode yang ditambahkan maupun kode yang dihapus. Lalu lintas perubahan tahunan (ACT) dikalikan dengan total biaya pengembangan untuk sampai pada biaya pemeliharaan:

Biaya pemeliharaan = ACT × Biaya pengembangan

Kebanyakan model estimasi biaya pemeliharaan, bagaimanapun, hanya memberikan hasil perkiraan karena mereka tidak memperhitungkan beberapa faktor seperti tingkat pengalaman para insinyur, dan keakraban para insinyur dengan produk, persyaratan perangkat keras, kompleksitas perangkat lunak, dll.

13.5 RINGKASAN

- Pemeliharaan adalah fase paling mahal dari siklus hidup perangkat lunak dan oleh karena itu biasanya hemat biaya untuk berinvestasi dalam waktu dan upaya saat mengembangkan produk dan untuk menekankan pada pemeliharaan produk untuk mengurangi biaya pemeliharaan.

13.6 LATIHAN

1. Pilih opsi yang benar
   (a) Manakah dari berikut ini yang bukan merupakan penyebab pemeliharaan perangkat lunak untuk produk biasa?
      (i) Tidak mungkin untuk menjamin bahwa perangkat lunak bebas cacat bahkan setelah pengujian menyeluruh.
      (ii) Platform penerapan dapat berubah seiring waktu.
      (iii) Kebutuhan pengguna dapat berubah seiring waktu.
      (iv) Perangkat lunak mengalami keausan setelah penggunaan yang lama.
   (b) Produk perangkat lunak lama mengacu pada perangkat lunak yang:
      (i) Dikembangkan setidaknya 50 tahun yang lalu.
      (ii) Produk perangkat lunak usang.
      (iii) Produk perangkat lunak yang memiliki struktur dan kode desain yang buruk.
      (iv) Produk perangkat lunak yang tidak dapat diuji dengan benar sebelum pengiriman produk.
   (c) Manakah dari pernyataan berikut yang benar?
      (i) Produk lama secara otomatis menyiratkan produk yang sangat tua.
      (ii) Upaya total yang dihabiskan untuk mempertahankan produk rata-rata biasanya melebihi upaya dalam mengembangkannya.
      (iii) Reverse engineering meliputi re engineering.
      (iv) Reengineering meliputi reverse engineering.
   (d) Manakah dari jenis pemeliharaan berikut yang menghabiskan upaya maksimum untuk perangkat lunak biasa?
      (i) Adaptif
      (ii) Korektif

(iii) Pencegahan
(iv) Sempurna

2. Apa saja jenis perawatan berbeda yang mungkin dibutuhkan produk perangkat lunak? Mengapa perawatan ini diperlukan?
3. Jelaskan mengapa setiap sistem perangkat lunak harus menjalani pemeliharaan atau semakin menjadi kurang berguna.
4. Diskusikan model proses untuk pemeliharaan perangkat lunak dan tunjukkan bagaimana Anda akan memilih model pemeliharaan yang sesuai untuk proyek pemeliharaan yang ada.
5. Nyatakan apakah pernyataan berikut ini **BENAR** atau **SALAH**. Berikan alasan untuk jawaban Anda.
6. Apa yang Anda maksud dengan istilah rekayasa balik perangkat lunak? Mengapa diperlukan? Jelaskan berbagaI aktivitas yang dilakukan selama reverse engineering.
   (a) Produk perangkat lunak lama adalah produk yang telah dikembangkan sejak lama.
   (b) Pemeliharaan korektif adalah jenis pemeliharaan yang paling sering dilakukan pada produk perangkat lunak yang khas.
7. Apa yang Anda maksud dengan istilah rekayasa ulang perangkat lunak? Mengapa diperlukan? Jelaskan berbagai kegiatan yang dilakukan selama reverse engineering.
8. Jika produk perangkat lunak seharga Rp. 10.000.000 untuk pengembangan, hitung biaya pemeliharaan tahunan mengingat bahwa setiap tahun kira-kira 5 persen dari kode perlu dimodifikasi. Mengidentifikasi faktor-faktor yang membuat estimasi biaya pemeliharaan tidak akurat?
9. Apa yang dimaksud dengan produk perangkat lunak warisan? Jelaskan masalah yang akan dihadapi seseorang saat mempertahankan produk warisan.

BAB 14
PENGGUNAAN PERANGKAT LUNAK

Produk perangkat lunak mahal, ini membuat manajer proyek perangkat lunak selalu khawatir tentang tingginya biaya pengembangan perangkat lunak dan putus asa mencari cara untuk memotong biaya pengembangan. Cara yang mungkin untuk mengurangi biaya pengembangan adalah dengan menggunakan kembali bagian dari perangkat lunak yang dikembangkan sebelumnya. Selain mengurangi biaya dan waktu pengembangan, penggunaan kembali juga mengarah pada peningkatan kualitas produk yang dikembangkan karena komponen yang dapat digunakan kembali dipastikan memiliki kualitas tinggi. Pendekatan penggunaan kembali yang akhir-akhir ini menjadi terkenal adalah pengembangan berbasis komponen. Pengembangan perangkat lunak berbasis komponen berbeda dari pengembangan perangkat lunak tradisional dalam arti bahwa perangkat lunak dikembangkan dengan merakit perangkat lunak dari komponen yang sudah ada. Pengembangan perangkat lunak dengan penggunaan kembali sangat mirip ke insinyur perangkat keras modern yang membangun sirkuit elektronik dengan menggunakan tipe standar IC dan komponen lainnya. Dalam Bab ini, kita akan meninjau keadaan seni dalam penggunaan kembali perangkat lunak.

14.1 APA YANG DAPAT DIGUNAKAN KEMBALI?

Sebelum membahas rincian teknik penggunaan kembali, penting untuk mempertimbangkan jenis artefak yang terkait dengan pengembangan perangkat lunak yang dapat digunakan kembali. Hampir semua artefak yang terkait dengan pengembangan perangkat lunak, termasuk rencana proyek dan rencana pengujian dapat digunakan kembali. Namun, barang-barang menonjol yang dapat digunakan kembali secara efektif adalah:

- Spesifikasi kebutuhan
- Rancangan
- Kode
- Kasus uji
- Pengetahuan

Pengetahuan adalah artefak pengembangan paling abstrak yang dapat digunakan kembali. Dari semua artefak penggunaan kembali, penggunaan kembali pengetahuan terjadi secara otomatis tanpa upaya sadar ke arah ini. Namun, dua kesulitan utama dengan penggunaan kembali pengetahuan yang tidak direncanakan adalah bahwa developer yang berpengalaman dalam satu jenis produk dapat dimasukkan dalam tim yang mengembangkan jenis perangkat lunak yang berbeda. Juga, sulit untuk mengingat detail dari pengetahuan pengembangan yang berpotensi dapat digunakan kembali. Penggunaan kembali pengetahuan yang direncanakan dapat meningkatkan efektivitas penggunaan kembali. Untuk ini, pengetahuan yang dapat digunakan kembali harus diekstraksi dan didokumentasikan secara sistematis. Namun, biasanya sangat sulit untuk mengekstrak dan mendokumentasikan pengetahuan yang dapat digunakan kembali.

14.2 MENGAPA HAMPIR TIDAK ADA REUSE SEJAUH INI?

Skenario umum di banyak industri pengembangan perangkat lunak dijelaskan lebih lanjut. Insinyur yang bekerja di organisasi pengembangan perangkat lunak sering merasa bahwa sistem saat ini yang mereka kembangkan mirip dengan beberapa sistem terakhir yang dibangun. Namun, tidak ada perhatian yang diberikan tentang bagaimana tidak menduplikasi apa yang dapat digunakan kembali dari sistem yang dikembangkan sebelumnya. Semuanya sedang dibangun dari awal. Sistem saat ini tertinggal dari jadwal dan tidak ada yang punya waktu untuk mengetahui bagaimana kesamaan antara sistem saat ini dan sistem yang dikembangkan di masa lalu dapat dimanfaatkan.

Bahkan organisasi-organisasi yang memulai program penggunaan kembali, terlepas dari kesulitan di atas, menghadapi masalah lain. Pembuatan komponen yang dapat digunakan kembali dalam aplikasi yang berbeda adalah masalah yang sulit. Sangat sulit untuk mengantisipasi komponen yang tepat yang dapat digunakan kembali di berbagai aplikasi. Namun, bahkan ketika komponen yang dapat digunakan kembali dibuat dengan hati-hati dan tersedia untuk digunakan kembali, programmer lebih memilih untuk membuatnya sendiri, karena komponen yang tersedia sulit untuk dipahami dan disesuaikan dengan aplikasi baru.

Dalam konteks ini, pengamatan berikut ini penting: Rutinitas perpustakaan matematika digunakan kembali dengan sangat sukses oleh hampir setiap programmer. Tak seorang pun dalam pikiran mereka akan berpikir untuk menulis rutin untuk menghitung sinus atau kosinus. Mari kita selidiki mengapa penggunaan kembali fungsi matematika yang umum digunakan begitu mudah. Beberapa aspek menarik muncul. Cosinus berarti sama untuk semua. Setiap orang memiliki gagasan yang jelas tentang argumen seperti apa yang harus diambil cosinus, jenis pemrosesan yang akan dilakukan dan hasil yang dikembalikan. Kedua, perpustakaan matematika memiliki antarmuka kecil. Misalnya, cosinus hanya membutuhkan satu parameter. Juga, format data parameter distandarisasi. Ini adalah beberapa masalah mendasar yang akan tetap berlaku untuk semua diskusi selanjutnya tentang penggunaan kembali.

14.3 MASALAH DASAR DALAM SETIAP PROGRAM PENGGUNAAN KEMBALI

Berikut ini adalah beberapa masalah dasar yang harus jelas dipahami untuk memulai program penggunaan kembali:

- Pembuatan komponen.
- Pengindeksan dan penyimpanan komponen.
- Pencarian komponen.
- Pemahaman komponen.
- Adaptasi komponen.
- Pemeliharaan repositori.

Pembuatan komponen: Untuk pembuatan komponen, komponen yang dapat digunakan kembali harus diidentifikasi terlebih dahulu. Pemilihan jenis komponen yang tepat memiliki potensi untuk digunakan kembali adalah penting.

Pengindeksan dan penyimpanan komponen

Pengindeksan membutuhkan klasifikasi komponen yang dapat digunakan kembali sehingga dapat dengan mudah dicari ketika kita mencari komponen untuk digunakan kembali.

Komponen perlu disimpan dalam *relational database management system* (RDBMS) atau *object-oriented database system* (ODBMS) untuk akses yang efisien ketika jumlah komponen menjadi besar.

Pencarian komponen

Programmer perlu mencari komponen yang tepat yang sesuai dengan kebutuhan mereka dalam database komponen. Untuk dapat mencari komponen secara efisien, programmer memerlukan metode yang tepat untuk menggambarkan komponen yang mereka cari.

Pemahaman komponen

Programmer membutuhkan pemahaman yang tepat dan cukup lengkap tentang apa yang dilakukan komponen untuk dapat memutuskan apakah mereka dapat menggunakan kembali komponen tersebut. Untuk memudahkan pemahaman, komponen harus didokumentasikan dengan baik dan harus melakukan sesuatu yang sederhana.

Adaptasi komponen

Seringkali, komponen mungkin memerlukan adaptasi sebelum dapat digunakan kembali, karena komponen yang dipilih mungkin tidak sesuai dengan masalah yang dihadapi. Namun, mengutak-atik kode juga bukan solusi yang memuaskan karena sangat mungkin menjadi sumber bug.

Pemeliharaan repositori

Repositori komponen setelah dibuat membutuhkan pemeliharaan berkelanjutan. Komponen baru, saat dan ketika dibuat harus dimasukkan ke dalam repositori. Komponen yang rusak harus dilacak. Selanjutnya, ketika aplikasi baru muncul, aplikasi lama menjadi usang. Dalam hal ini, komponen usang mungkin harus dihapus dari repositori.

14.4 PENDEKATAN PENGGUNAAN KEMBALI

Pendekatan menjanjikan yang diadopsi oleh banyak organisasi adalah dengan memperkenalkan pendekatan blok bangunan ke dalam proses pengembangan perangkat lunak. Untuk ini, komponen yang dapat digunakan kembali perlu diidentifikasi setelah setiap proyek pengembangan selesai. Penggunaan kembali komponen yang diidentifikasi harus ditingkatkan dan ini harus dikatalogkan ke perpustakaan komponen. Harus dipahami dengan jelas bahwa masalah penting untuk setiap upaya penggunaan kembali adalah identifikasi komponen yang dapat digunakan kembali. Analisis domain adalah pendekatan yang menjanjikan untuk mengidentifikasi komponen yang dapat digunakan kembali. Pendekatan analisis domain untuk membuat komponen yang dapat digunakan kembali adalah sebagai berikut:

Analisis Domain

Tujuan dari analisis domain adalah untuk mengidentifikasi komponen yang dapat digunakan kembali untuk domain masalah.

Gunakan kembali domain

Domain reuse adalah kumpulan area aplikasi yang terkait secara teknis. Sebuah badan informasi dianggap sebagai domain masalah untuk digunakan kembali, jika ada hubungan yang mendalam dan komprehensif di antara item informasi yang dicirikan oleh pola kesamaan

di antara komponen pengembangan produk perangkat lunak. Sebuah domain reuse adalah pemahaman bersama dari beberapa komunitas, ditandai dengan konsep, teknik, dan terminologi yang menunjukkan beberapa koherensi. Contoh domain adalah domain software akuntansi, domain software perbankan, domain software bisnis, domain software otomatisasi manufaktur, domain software telekomunikasi, dll.

Hanya untuk menjadi akrab dengan kosa kata domain membutuhkan interaksi berbulan-bulan dengan para ahli. Seringkali, seseorang harus terbiasa dengan jaringan domain terkait untuk berhasil melakukan analisis domain. Analisis domain mengidentifikasi objek, operasi, dan hubungan di antara mereka. Misalnya, pertimbangkan sistem reservasi maskapai, objek yang dapat digunakan kembali dapat berupa kursi, penerbangan, bandara, kru, pesanan makanan, dll. Operasi yang dapat digunakan kembali dapat menjadwalkan penerbangan, memesan kursi, menugaskan kru untuk penerbangan, dll. Kita dapat melihat bahwa analisis domain menggeneralisasi domain aplikasi. Model domain melampaui aplikasi tertentu. Karakteristik umum atau kesamaan antara sistem digeneralisasikan.

Selama analisis domain, komunitas khusus developer perangkat lunak berkumpul untuk membahas solusi komunitas secara luas. Analisis domain aplikasi diperlukan untuk mengidentifikasi komponen yang dapat digunakan kembali. Konstruksi sebenarnya dari komponen yang dapat digunakan kembali untuk domain disebut *rekayasa domain.*

Evolusi domain penggunaan kembali

Hasil akhir dari analisis domain adalah pengembangan bahasa berorientasi masalah. Bahasa berorientasi masalah juga dikenal sebagai generator aplikasi. Generator aplikasi ini, setelah dikembangkan membentuk standar pengembangan aplikasi. Domain perlahan berkembang. Saat domain berkembang, kita dapat membedakan berbagai tahapan yang dilaluinya:

Tahap 1 : Tidak ada set notasi yang jelas dan konsisten. Jelas, tidak ada komponen yang dapat digunakan kembali yang tersedia. Semua perangkat lunak ditulis dari awal.

Tahap 2: Hanya pengalaman dari proyek serupa yang digunakan dalam upaya pengembangan. Artinya yang ada hanya penggunaan pengetahuan kembali.

Tahap 3: Pada tahap ini, domain sudah siap untuk digunakan kembali. Himpunan konsep distabilkan dan notasi distandarisasi. Solusi standar untuk masalah standar tersedia. Ada pengetahuan dan penggunaan kembali komponen.

Tahap 4: Domain telah sepenuhnya dieksplorasi. Pengembangan perangkat lunak untuk domain sebagian besar dapat diotomatisasi. Program tidak lagi ditulis dalam pengertian tradisional. Program ditulis menggunakan bahasa domain tertentu, yang juga dikenal sebagai *generator aplikasi.*

Klasifikasi Komponen

Komponen perlu diklasifikasikan dengan benar untuk mengembangkan skema pengindeksan dan penyimpanan yang efektif. Penggunaan kembali perangkat keras telah sangat berhasil. Jika kita melihat klasifikasi komponen perangkat keras untuk petunjuk, maka kita dapat mengamati bahwa komponen perangkat keras diklasifikasikan menggunakan hierarki bertingkat. Pada level terendah, komponen dijelaskan dalam beberapa bentuk — deskripsi bahasa alami, skema logika, informasi waktu, dll. Semakin tinggi level di mana komponen dijelaskan, semakin banyak ambiguitas. Ini telah memotivasi skema klasifikasi Prieto-Diaz.

Skema klasifikasi Prieto-Diaz

Setiap komponen paling baik dijelaskan dengan menggunakan sejumlah karakteristik atau aspek yang berbeda. Misalnya, objek dapat diklasifikasikan menggunakan yang berikut:

- Tindakan yang mereka wujudkan.
- Objek yang mereka manipulasi.
- Struktur data yang digunakan.
- Sistem tempat mereka menjadi bagian, dll.

Skema klasifikasi segi Prieto-Diaz membutuhkan pemilihan *n-tupel* yang paling sesuai dengan komponen. Klasifikasi faceted memiliki keunggulan dibandingkan klasifikasi enumeratif. Skema enumeratif yang ketat menggunakan hierarki yang telah ditentukan sebelumnya. Oleh karena itu, ini memaksa Anda untuk mencari item yang paling sesuai dengan komponen yang akan diklasifikasikan. Hal ini membuat sangat sulit untuk mencari komponen yang dibutuhkan. Meskipun referensi silang ke item lain dapat dimasukkan, jaringan yang dihasilkan menjadi rumit.

Pencarian

Repositori domain mungkin berisi ribuan item penggunaan kembali. Dalam domain sebesar itu, apa cara paling efisien untuk mencari item yang dicari? Teknik pencarian populer yang terbukti sangat efektif adalah yang menyediakan antarmuka web ke repositori. Menggunakan antarmuka web seperti itu, seseorang akan mencari item menggunakan perkiraan pencarian otomatis menggunakan kata kunci, dan kemudian dari hasil ini akan melakukan penelusuran menggunakan tautan yang disediakan untuk mencari item terkait. Perkiraan pencarian otomatis menemukan produk yang tampaknya memenuhi beberapa persyaratan yang ditentukan. Item yang terletak melalui pencarian perkiraan berfungsi sebagai titik awal untuk menjelajahi repositori. Ini berfungsi sebagai titik awal untuk menjelajahi repositori. developer dapat mengikuti tautan ke produk lain sampai ditemukan kecocokan yang cukup baik. Penjelajahan dilakukan dengan menggunakan tautan kata kunci ke kata kunci, kata kunci ke produk, dan tautan produk ke produk. Tautan ini membantu menemukan produk tambahan dan membandingkan atribut detailnya. Menemukan item yang memuaskan dari repositori mungkin memerlukan beberapa iterasi pencarian perkiraan diikuti dengan browsing. Dengan setiap iterasi, developer akan mendapatkan pemahaman yang lebih baik tentang produk yang tersedia dan perbedaannya. Namun, kita harus ingat bahwa item yang akan dicari dapat berupa komponen, desain, model, persyaratan, dan bahkan pengetahuan.

Pemeliharaan Repositori

Pemeliharaan repositori melibatkan memasukkan item baru, menghentikan item yang tidak lagi diperlukan, dan memodifikasi atribut pencarian item untuk meningkatkan efektivitas pencarian. Juga, tautan yang berkaitan dengan item yang berbeda mungkin perlu dimodifikasi untuk meningkatkan efektivitas pencarian. Industri perangkat lunak selalu berusaha mengimplementasikan sesuatu yang belum pernah dilakukan sebelumnya. Saat persyaratan pola muncul, komponen baru yang dapat digunakan kembali diidentifikasi, yang pada akhirnya dapat menjadi lebih atau kurang standar. Namun, seiring kemajuan teknologi, beberapa komponen yang masih dapat digunakan kembali, tidak sepenuhnya memenuhi persyaratan saat ini. Di sisi lain, membatasi penggunaan kembali untuk komponen yang sangat matang, dapat mengorbankan potensi peluang penggunaan kembali. Membuat produk tersedia

sebelum dinilai secara menyeluruh dapat menjadi kontra produktif. Pengalaman negatif cenderung melarutkan kepercayaan pada seluruh kerangka penggunaan kembali.

Penggunaan Kembali Tanpa Modifikasi

Setelah solusi standar muncul, tidak ada modifikasi pada bagian program yang mungkin diperlukan. Seseorang dapat langsung mencolokkan bagian-bagiannya untuk mengembangkan aplikasinya. Penggunaan kembali tanpa modifikasi jauh lebih berguna daripada pustaka program klasik. Ini dapat didukung oleh kompiler melalui tautan ke rutinitas dukungan run-time (generator aplikasi). Generator aplikasi menerjemahkan spesifikasi ke dalam program aplikasi. Spesifikasi biasanya ditulis menggunakan 4GL. Spesifikasi mungkin juga dalam bentuk visual. Programmer akan membuat gambar grafis menggunakan beberapa simbol standar yang tersedia. Mendefinisikan apa itu varian dan apa yang invarian sesuai dengan parameterisasi subrutin untuk membuatnya dapat digunakan kembali. Parameter subrutin adalah varian karena programmer dapat menentukannya saat memanggil subrutin. Bagian dari subrutin yang tidak diparameterisasi, tidak dapat diubah.

Generator aplikasi memiliki keunggulan signifikan dibandingkan program berparameter sederhana. Yang terbesar dari ini adalah bahwa generator aplikasi dapat mengekspresikan informasi varian dalam bahasa yang sesuai daripada dibatasi pada parameter fungsi, konstanta bernama, atau tabel. Keuntungan lainnya termasuk lebih sedikit kesalahan, lebih mudah untuk dipelihara, secara substansial mengurangi upaya pengembangan, dan fakta bahwa seseorang tidak perlu repot dengan detail implementasi. Pembangkit aplikasi menjadi cacat ketika diperlukan untuk mendukung beberapa konsep atau fitur baru. Beberapa generator aplikasi mengatasi hambatan ini melalui mekanisme pelarian. Programmer dapat menulis kode dalam beberapa 3GL melalui mekanisme ini. Generator aplikasi telah berhasil diterapkan ke aplikasi pemrosesan data, antarmuka pengguna, dan pengembangan kompiler. Generator aplikasi kurang berhasil dengan pengembangan aplikasi dengan interaksi yang erat dengan perangkat keras seperti sistem waktu nyata.

14.5 PENGGUNAAN KEMBALI DI TINGKAT ORGANISASI

Penggunaan kembali harus menjadi bagian standar dalam semua kegiatan pengembangan perangkat lunak termasuk spesifikasi, desain, implementasi, pengujian, dll. Idealnya, harus ada aliran komponen yang dapat digunakan kembali. Namun, dalam praktiknya, hal-hal tidak sesederhana itu. Mengekstrak komponen yang dapat digunakan kembali dari proyek yang diselesaikan di masa lalu menghadirkan kesulitan penting yang tidak ditemui saat mengekstraksi komponen yang dapat digunakan kembali dari proyek yang sedang berlangsung—biasanya, developer asli tidak lagi tersedia untuk konsultasi. Pengembangan sistem baru menghasilkan bermacam-macam produk, karena reusability berkisar dari item yang reusability-nya langsung hingga item yang reusability-nya sangat tidak mungkin. Mencapai penggunaan kembali tingkat organisasi memerlukan penerapan langkah-langkah berikut:

- Menilai potensi item untuk digunakan kembali.
- Sempurnakan item agar dapat digunakan kembali dengan lebih baik.
- Masukkan produk ke dalam repositori penggunaan kembali.

Tiga langkah yang diperlukan untuk mencapai penggunaan kembali tingkat organisasi adalah sebagai berikut:

Menilai potensi produk untuk digunakan kembali

Penilaian potensi penggunaan kembali komponen dapat diperoleh dari analisis kuesioner yang diedarkan di antara para pengembang. Kuesioner dapat dirancang untuk menilai reusability komponen. Programmer yang bekerja di domain aplikasi serupa dapat digunakan untuk menjawab kuesioner tentang reusability produk. Bergantung pada jawaban yang diberikan oleh pemrogram, apakah komponen diambil untuk digunakan kembali sebagaimana adanya, dimodifikasi dan disempurnakan sebelum dimasukkan ke dalam repositori penggunaan kembali, atau diabaikan. Contoh kuesioner untuk menilai reusability suatu komponen adalah sebagai berikut:

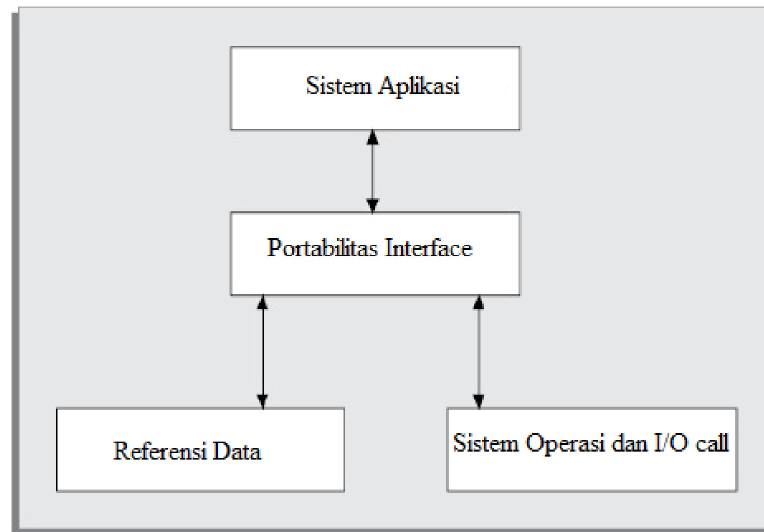
- Apakah fungsionalitas komponen diperlukan untuk implementasi sistem di masa mendatang?
- Seberapa umumkah fungsi komponen dalam domainnya?
- Apakah akan ada duplikasi fungsi dalam domain jika komponen diambil?
- Apakah komponen perangkat keras tergantung?
- Apakah desain komponen cukup dioptimalkan?
- Jika komponen tidak dapat digunakan kembali, apakah dapat didekomposisi untuk menghasilkan beberapa komponen yang dapat digunakan kembali?
- Bisakah kita membuat parameter komponen yang tidak dapat digunakan kembali sehingga menjadi dapat digunakan kembali?

Menyempurnakan produk agar dapat digunakan kembali dengan lebih baik

Agar suatu produk dapat digunakan kembali, produk tersebut harus relatif mudah untuk menyesuaikannya dengan konteks yang berbeda. Ketergantungan mesin harus diabstraksikan atau dilokalkan menggunakan teknik enkapsulasi data. Penyempurnaan berikut dapat dilakukan:

- o **Generalisasi nama**: Nama-nama harus bersifat umum, bukan terkait langsung dengan aplikasi tertentu.
- o **Generalisasi operas**i: Operasi harus ditambahkan untuk membuat komponen lebih umum. Juga, operasi yang terlalu spesifik untuk suatu aplikasi dapat dihapus.
- o **Generalisasi pengecualian**: Ini melibatkan pemeriksaan setiap komponen untuk melihat pengecualian mana yang mungkin dihasilkannya. Untuk komponen umum, beberapa jenis: pengecualian mungkin harus ditangani.

Menangani masalah portabilitas: Program biasanya membuat beberapa asumsi mengenai representasi informasi di mesin yang mendasarinya. Asumsi ini secara umum tidak berlaku untuk semua mesin. Program juga sering perlu memanggil beberapa fungsi sistem operasi dan panggilan ini mungkin tidak sama di semua mesin. Juga, program menggunakan beberapa pustaka fungsi, yang mungkin tidak tersedia di semua mesin host. Solusi portabilitas untuk mengatasi masalah ini ditunjukkan pada Gambar 14.1. Solusi portabilitas menyarankan bahwa daripada memanggil sistem operasi dan prosedur I/O secara langsung, versi abstrak dari ini harus dipanggil oleh program aplikasi. Selain itu, semua panggilan terkait platform harus dirutekan melalui antarmuka portabilitas. Satu masalah dengan solusi ini adalah biaya overhead yang signifikan, yang membuatnya tidak dapat diterapkan ke banyak sistem dan aplikasi waktu nyata yang membutuhkan respons yang sangat cepat.



Gambar 14.1 Meningkatkan penggunaan kembali komponen dengan menggunakan antarmuka portabilitas

Terlepas dari semua kekurangan teknik penggunaan kembali yang canggih, pengalaman beberapa organisasi bahwa sebagian besar faktor yang menghambat program penggunaan kembali yang efektif adalah non-teknis. Beberapa faktor tersebut adalah sebagai berikut:

- Perlu komitmen dari manajemen puncak.
- Dokumentasi yang memadai untuk mendukung penggunaan kembali.
- Insentif yang memadai untuk memberi penghargaan kepada mereka yang menggunakan kembali.Baik orang yang menyumbangkan komponen baru yang dapat digunakan kembali dan mereka yang menggunakan kembali komponen yang ada harus diberi penghargaan untuk memulai program penggunaan kembali dan mempertahankannya.
- Menyediakan akses dan informasi tentang komponen yang dapat digunakan kembali. Organisasi sering ragu-ragu untuk memberikan akses terbuka ke repositori penggunaan kembali karena takut komponen penggunaan kembali menemukan cara untuk pesaing mereka.

14.6 RINGKASAN

- Masalah dasar yang harus ditangani untuk memulai program penggunaan kembali yang berarti
  - pembuatan komponen
  - pengindeksan dan penyimpanan komponen
  - pencarian komponen
  - pemahaman komponen
  - adaptasi komponen
  - pemeliharaan repositori.
- Pembuatan komponen yang sangat dapat digunakan kembali adalah masalah yang sangat sulit. Pendekatan yang menjanjikan adalah analisis domain. Analisis domain

bertujuan untuk mengidentifikasi komponen yang dapat digunakan kembali untuk domain masalah.

- Generator aplikasi menerjemahkan spesifikasi masalah ke dalam program aplikasi. Aplikasi generator sangat memudahkan penggunaan kembali dibandingkan dengan cara lain untuk menggunakan kembali komponen.
- Penggunaan kembali di tingkat organisasi memiliki tiga langkah penting yang harus diikuti, ini adalah menilai potensi item untuk digunakan kembali; perbaiki item agar dapat digunakan kembali dengan lebih baik, dan masukkan produk ke dalam repositori penggunaan ulang.

14.7 LATIHAN

1. Sebutkan alasan teknis dan non-teknis utama yang menghambat penggunaan kembali perangkat lunak. Bisakah ada keadaan di mana penggunaan kembali perangkat lunak tidak dapat direkomendasikan?
2. Identifikasi poin-poin penting yang harus diperhatikan oleh developer paket perangkat lunak untuk meningkatkan reusabilitas paket.
3. Mengapa penting bagi organisasi untuk melakukan program penggunaan kembali yang efektif? Apa artifak penting yang dapat digunakan kembali? Mengapa penggunaan kembali komponen perangkat lunak jauh lebih sulit daripada komponen perangkat keras?
4. Identifikasi alasan mengapa penggunaan kembali perangkat lunak matematika sangat berhasil. Juga, identifikasi alasan mengapa penggunaan kembali komponen perangkat lunak selain perangkat lunak matematika sulit dilakukan.
5. Apa yang Anda pahami dengan istilah reuse domain? Jelaskan analisis domain dan bagaimana analisis domain mengarah pada peningkatan penggunaan kembali komponen.
6. Apakah Anda setuju dengan pernyataan: “kode” adalah artefak terpenting yang dapat digunakan kembali selama pengembangan perangkat lunak? Justifikasi jawaban Anda.
7. Apa yang Anda pahami dengan istilah analisis domain dalam konteks desain perangkat lunak? Artefak apa yang dihasilkan setelah analisis domain? Bagaimana analisis domain meningkatkan penggunaan kembali perangkat lunak?
8. Membandingkan kelebihan dan kekurangan program reuse berbasis component library dan yang lain berbasis application generator.
9. Jelaskan bagaimana komponen dapat dibuat secara efektif untuk digunakan kembali.
10. Jelaskan aspek (langkah) penting dalam memulai dan memelihara program penggunaan kembali yang efektif dalam organisasi pengembangan perangkat lunak.
11. Bagaimana Anda dapat meningkatkan penggunaan kembali fungsi yang Anda tulis?
12. Identifikasi tahapan di mana domain penggunaan kembali berlangsung.
13. Jelaskan mengapa penggunaan kembali sulit dalam pengembangan perangkat lunak dibandingkan dengan pengembangan perangkat keras.
14. Jelaskan mengapa penggunaan kembali fungsi matematis lebih mudah dibandingkan dengan penggunaan kembali fungsi non-matematis.
15. Apa yang Anda pahami dengan istilah "klasifikasi segi" dalam konteks penggunaan kembali perangkat lunak? Bagaimana klasifikasi segi menyederhanakan pencarian komponen di toko komponen?

16. Jelaskan bagaimana klasifikasi komponen faceted (skema Prieto-Diaz) dapat digunakan untuk pencarian perkiraan. Apa keuntungan dari pencarian perkiraan dibandingkan pencarian yang tepat?
17. Misalkan tim Anda telah mengembangkan produk perangkat lunak. Bagaimana Anda menilai potensi penggunaan kembali dari fungsi yang dikembangkan?
18. Dalam penggunaan kembali perangkat lunak tingkat organisasi, identifikasi aspek fungsi yang dikembangkan yang menghambat kegunaannya kembali. Bagaimana Anda dapat meningkatkan penggunaan kembali komponen yang telah Anda identifikasi untuk digunakan kembali?
19. Apa yang dimaksud dengan generator aplikasi? Mengapa penggunaan kembali lebih mudah saat menggunakan generator aplikasi dibandingkan dengan pustaka komponen? Apa kekurangan dari sebuah aplikasi generator?
20. Rancang skema untuk menyimpan artefak penggunaan kembali perangkat lunak. Jelaskan bagaimana komponen dapat dicari dalam skema Anda.

BAB 15
TREN YANG MUNCUL

Teknik rekayasa perangkat lunak di masa lalu telah berkembang sebagai tanggapan terhadap tantangan yang diajukan untuk pengembangan program oleh perubahan lingkungan di mana program dijalankan dan juga perubahan jenis aplikasi yang dibutuhkan oleh pengguna. Dengan perubahan lingkungan, yakni perubahan yang terjadi pada berbagai teknologi yang mendasari perangkat keras komputer, perangkat lunak sistem, jaringan, dan perangkat periferal. Mari kita periksa bagaimana lingkungan telah berubah akhir-akhir ini. Ini dapat menunjukkan tantangan yang diajukan pada prinsip-prinsip pengembangan perangkat lunak. Ini pada gilirannya akan memberi kita beberapa wawasan tentang cara teknik rekayasa perangkat lunak berkembang akhir-akhir ini. Perubahan penting terhadap lingkungan yang terjadi dalam dua dekade terakhir antara lain sebagai berikut:

- Harga komputer telah turun drastis dalam periode ini. Pada saat yang sama, mereka menjadi lebih kuat. Sekarang mereka dapat melakukan komputasi lebih cepat dan menyimpan volume data yang jauh lebih besar. Ukuran komputer telah menyusut dan laptop serta palmtop menjadi populer.
- Internet telah menjadi sangat populer. Internet menghubungkan jutaan komputer di seluruh dunia dan menyediakan banyak sekali bagi pengguna.
- Teknik jaringan telah membuat kemajuan pesat. Kecepatan transfer data telah meningkat luar biasa dan pada saat yang sama, biaya jaringan komputer telah turun secara dramatis. Sebagai contoh kecepatan transfer data yang saat ini didukung, desktop kini hadir dengan port jaringan 1Gbps default.
- Ponsel telah secara dramatis menangkap imajinasi semua orang. Tingkat penerimaan yang dicapai ponsel dalam waktu kurang dari satu dekade tampak seperti bab langsung dari buku fiksi ilmiah. Ponsel dengan cepat mengubah dirinya menjadi perangkat komputasi genggam. Selain koneksi fixed line berkecepatan tinggi, GPRS dan LAN nirkabel telah menjadi hal yang umum.
- Selama dekade terakhir, komputasi awan telah menjadi populer. Dalam komputasi awan, aplikasi di-host di cloud yang beroperasi di pusat data. Komputasi awan menjadi semakin populer karena membantu pengguna menjalankan aplikasi canggih tanpa banyak investasi di muka dan juga membebaskannya dari membeli dan memelihara perangkat keras dan perangkat lunak canggih.

Dalam menghadapi perkembangan yang dibahas, developer perangkat lunak menghadapi beberapa tantangan. Berikut adalah beberapa tantangan yang sedang dihadapi oleh developer perangkat lunak.

Tantangan yang dihadapi oleh developer perangkat lunak

Berikut adalah beberapa tantangan yang sedang dihadapi oleh developer perangkat lunak:

- Untuk mengatasi persaingan yang ketat, rumah bisnis dengan cepat mengubah proses bisnis mereka. Hal ini membutuhkan perubahan yang cepat juga terjadi pada perangkat lunak yang mendukung kegiatan proses bisnis. Oleh karena itu, ada permintaan mendesak untuk mempersingkat waktu pengiriman perangkat lunak. Namun, perangkat lunak masih membutuhkan waktu yang sangat lama untuk dikembangkan dan ternyata menjadi hambatan dalam menerapkan perubahan proses

bisnis yang cepat. Untuk mengurangi waktu pengiriman perangkat lunak, perangkat lunak sedang dikembangkan oleh tim yang bekerja dari lokasi yang didistribusikan secara global. Bagaimana perangkat lunak dapat dikembangkan secara efektif menggunakan tim pengembangan yang terdistribusi secara global masih belum jelas dan menimbulkan banyak tantangan. Di sisi lain, perubahan radikal pada prinsip pengembangan perangkat lunak diajukan untuk mempersingkat waktu pengembangan.

- Rumah bisnis mulai bosan dengan biaya perangkat lunak yang luar biasa, pengiriman yang terlambat, dan produk berkualitas buruk. Di sisi lain, biaya perangkat keras menurun dan pada saat yang sama perangkat keras menjadi lebih kuat, canggih, dan andal. Perbedaan biaya perangkat keras dan perangkat lunak menjadi semakin mencolok. Kebijaksanaan mengembangkan setiap perangkat lunak dari awal dipertanyakan. Juga, model pengiriman perangkat lunak alternatif sedang diusulkan untuk mengurangi biaya perangkat lunak.
- Ukuran perangkat lunak semakin meningkat.
- Setelah Internet menjadi sangat populer, banyak produk perangkat lunak sekarang diperlukan untuk berinteraksi dengan Internet. Banyak produk bahkan diharapkan dapat bekerja di Internet. Selain itu, dengan tersedianya jaringan yang cepat, aplikasi terdistribusi menjadi hal yang biasa. Namun, tidak jelas bagaimana perangkat lunak dikembangkan secara efektif dalam konteks platform terdistribusi dan Internet.

Menanggapi tantangan yang dihadapi, tren rekayasa perangkat lunak berikut menjadi nyata:

- Perangkat lunak server-klien
- *Service-Oriented Architecture* (SOA)
- *Software as a Service* (SaaS)

15.1 PERANGKAT LUNAK CLIENT-SERVER

Dalam perangkat lunak klien-server, klien dan server pada dasarnya adalah komponen perangkat lunak. Klien adalah konsumen layanan dan server adalah penyedia layanan. Konsep client-server bukanlah konsep baru. Itu sudah ada di masyarakat sejak lama. Sebagai contoh, seorang guru dapat menjadi klien seorang dokter, dan dokter tersebut pada gilirannya dapat menjadi klien seorang tukang cukur, yang pada gilirannya dapat menjadi klien dari seorang pengacara, dan seterusnya. Dari sini, kita dapat mengamati bahwa server dalam beberapa konteks dapat menjadi klien dalam beberapa konteks lain. Jadi, klien dan server dapat dianggap sebagai peran belaka. Mengingat tingkat popularitas paradigma client-server dalam konteks pengembangan perangkat lunak, harus ada beberapa keuntungan yang diperoleh dari mengadopsi konsep ini. Mari kita membahas keuntungan penting dari paradigma client-server.

Keuntungan dari perangkat lunak client-server

Ada banyak alasan untuk popularitas perangkat lunak server klien. Beberapa alasan penting adalah sebagai berikut:

Concurrency: Perangkat lunak client-server membagi pekerjaan komputasi di antara banyak komponen klien dan server yang berbeda yang dapat berada di mesin yang berbeda. Jadi solusi client-server secara inheren bersamaan dan sebagai hasilnya menawarkan keuntungan dari pemrosesan yang lebih cepat.

Kopling longgar: Komponen klien dan server secara inheren digabungkan secara longgar, membuatnya mudah dipahami dan dikembangkan.

Fleksibilitas: Perangkat lunak klien-server fleksibel dalam arti bahwa klien dan server dapat dipasang dan dihapus jika diperlukan. Juga, klien dapat mengakses server dari mana saja.

Efektivitas biaya: Paradigma client-server biasanya mengarah pada solusi hemat biaya. Klien biasanya berjalan pada komputer desktop yang murah, sedangkan server dapat berjalan pada komputer yang canggih dan mahal. Bahkan untuk menggunakan perangkat lunak yang canggih, seseorang hanya perlu memiliki mesin klien yang murah untuk memanggil server.

Perangkat keras heterogen: Dalam solusi client-server, mudah untuk memiliki server khusus yang dapat secara efisien memecahkan masalah tertentu. Dimungkinkan untuk secara efisien mengintegrasikan platform komputasi heterogen untuk mendukung persyaratan berbagai jenis perangkat lunak server.

Fault-tolerance: Solusi client-server biasanya toleran terhadap kesalahan. Dimungkinkan untuk memiliki banyak server yang menyediakan layanan yang sama. Jika satu server menjadi tidak tersedia, maka permintaan klien dapat diarahkan ke server lain yang berfungsi.

Komputasi seluler: Komputasi seluler secara implisit membutuhkan penggunaan teknik clientserver. Ponsel, akhir-akhir ini, berkembang sebagai komputasi genggam dan perangkat komunikasi dan dilengkapi dengan daya pemrosesan kecil, keyboard, memori kecil, dan layar LCD. Perangkat genggam memiliki daya pemrosesan dan kapasitas penyimpanan yang terbatas, dan oleh karena itu hanya dapat bertindak sebagai klien. Untuk melakukan tugas non-sepele, komputer genggam mungkin hanya dapat mendukung antarmuka pengguna yang diperlukan untuk menempatkan permintaan pada beberapa server jarak jauh.

Penyediaan layanan aplikasi: Ada banyak produk perangkat lunak aplikasi yang sangat mahal untuk dimiliki. Pendekatan berbasis client-server dapat digunakan untuk membuat produk perangkat lunak ini terjangkau untuk digunakan. Dalam pendekatan ini, application service provider (ASP) akan memilikinya, dan pengguna akan membayar ASP berdasarkan biaya per satuan waktu penggunaan.

Pengembangan berbasis komponen: Paradigma client-server sangat cocok dengan pengembangan perangkat lunak berbasis komponen. Pengembangan perangkat lunak berbasis komponen menjanjikan pencapaian pengurangan biaya dan waktu pengiriman yang substansial dan pada saat yang sama mencapai peningkatan keandalan produk. Pengembangan berbasis komponen mirip dengan cara peralatan perangkat keras dibangun dengan biaya yang efektif. developer perangkat keras mencapai penghematan biaya, tenaga, dan waktu dalam pengembangan peralatan oleh integrating pre-built components (ICs) yang dibeli langsung di printed circuit board (PCB). Dalam paradigma komponen, pengembangan perangkat lunak terdiri dari pengintegrasian komponen perangkat lunak yang sudah ada dan menulis hanya bagian yang hilang.

Seperti yang telah dibahas, keuntungan dari paradigma perangkat lunak client-server sangat banyak. Tidak heran bahwa paradigma client-server telah menjadi sangat populer. Namun, sebelum kita membahas lebih detail tentang teknologi ini, penting untuk mengetahui kekurangan penting darinya juga.

Kekurangan perangkat lunak client-server

Ada beberapa kelemahan pengembangan perangkat lunak client-server. Kerugian utama adalah:

Keamanan: Dalam aplikasi monolitik, menangani masalah keamanan jauh lebih mudah dibandingkan dengan implementasi client-server. Perangkat lunak berbasis client-server menyediakan banyak fleksibilitas. Sebagai contoh, klien dapat terhubung ke server dari mana saja. Ini memudahkan peretas untuk membobol sistem. Oleh karena itu, memastikan keamanan sistem server klien adalah tugas yang sangat menantang.

Server bisa menjadi bottleneck: Server bisa menjadi bottleneck karena banyak klien mungkin mencoba terhubung ke server pada saat yang bersamaan. Masalah ini muncul karena fleksibilitas mengingat setiap klien dapat terhubung kapan saja diperlukan.

Kompatibilitas: Klien dan server mungkin tidak kompatibel satu sama lain. Karena komponen klien dan server mungkin diproduksi oleh vendor yang berbeda, mereka mungkin tidak kompatibel sehubungan dengan tipe data, bahasa, representasi nomor, dll.

Inkonsistensi: Replikasi server berpotensi menimbulkan masalah karena setiap kali ada replikasi data, ada bahaya data menjadi tidak konsisten.

15.2 ARSITEKTUR CLIENT-SERVER

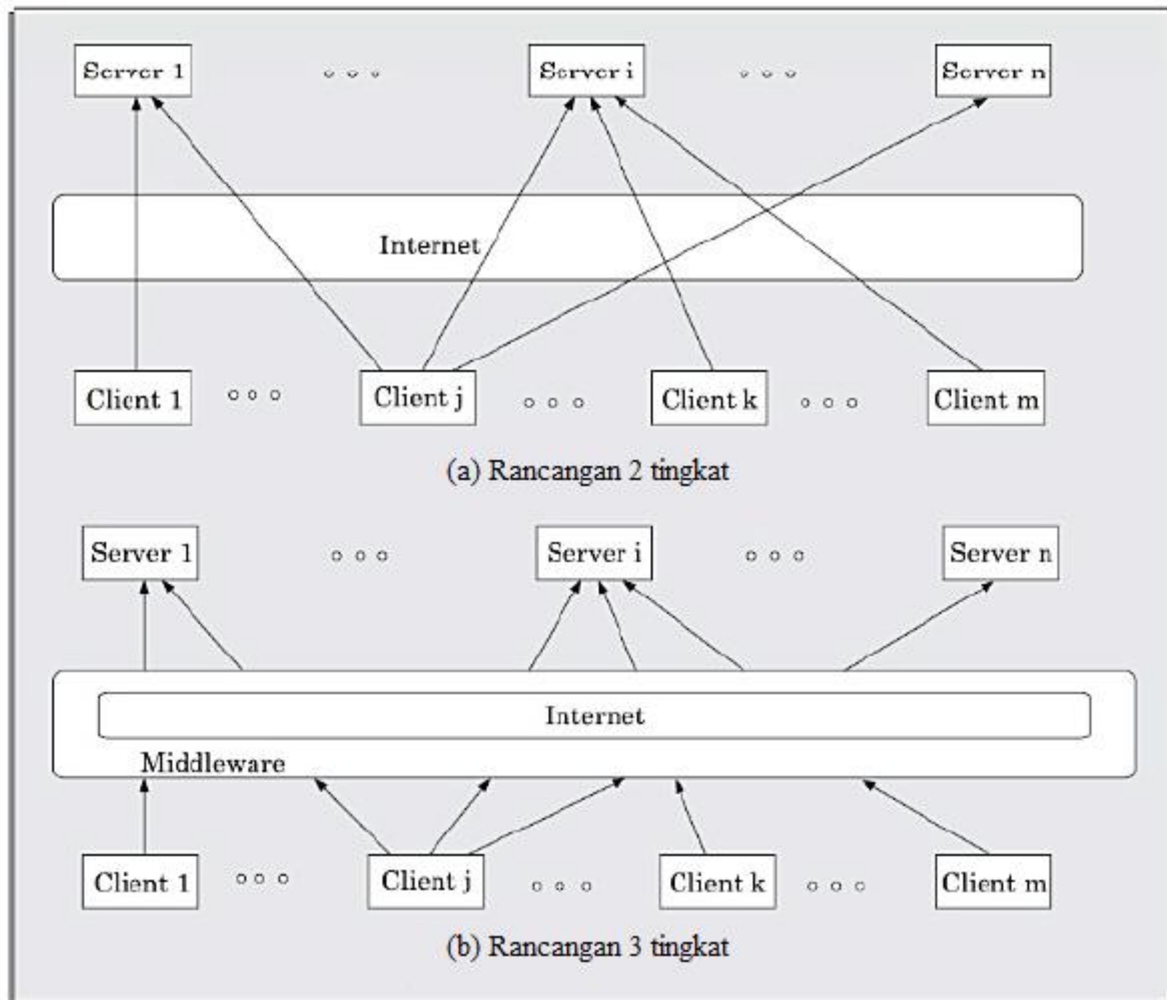
Cara paling sederhana untuk menghubungkan klien dan server adalah dengan menggunakan arsitektur dua tingkat yang ditunjukkan pada Gambar 15.1(a). Dalam arsitektur dua tingkat, setiap klien dapat memperoleh layanan dari server mana pun dengan mengirimkan permintaan melalui jaringan.

Keterbatasan arsitektur client-server dua tingkat

Arsitektur dua tingkat untuk aplikasi client-server adalah solusi yang jelas secara intuitif, tetapi ternyata tidak dapat digunakan secara praktis. Masalah utama adalah bahwa komponen klien dan server biasanya dibuat oleh vendor yang berbeda, yang mungkin mengadopsi solusi antarmuka dan implementasi mereka sendiri. Akibatnya, komponen yang berbeda mungkin tidak saling berinteraksi (berbicara) dengan mudah.

Arsitektur client-server tiga tingkat

Arsitektur tiga tingkat mengatasi keterbatasan utama arsitektur twotier. Dalam arsitektur tiga tingkat, middleware ditambahkan antara klien dan komponen server seperti yang ditunjukkan pada Gambar 15.1(b). Middleware melacak semua server. Ini juga menerjemahkan permintaan klien ke dalam bentuk server yang dapat dimengerti. Misalnya, klien dapat mengirimkan permintaannya ke middleware dan melepaskannya karena middleware akan mengakses data dan mengembalikan jawabannya ke klien.



Gambar 15.1 Arsitektur client-server dua tingkat dan tiga tingkat.

Fungsi middleware

Aktivitas penting dari middleware meliputi:

- Middleware melacak alamat server. Berdasarkan permintaan klien, karena itu dapat dengan mudah menemukan server yang diperlukan.
- Itu dapat menerjemahkan antara format data klien dan server dan sebaliknya.

Dua standar middleware yang populer adalah:

- Common Object Request Broker Architecture (CORBA)
- COM/DCOM

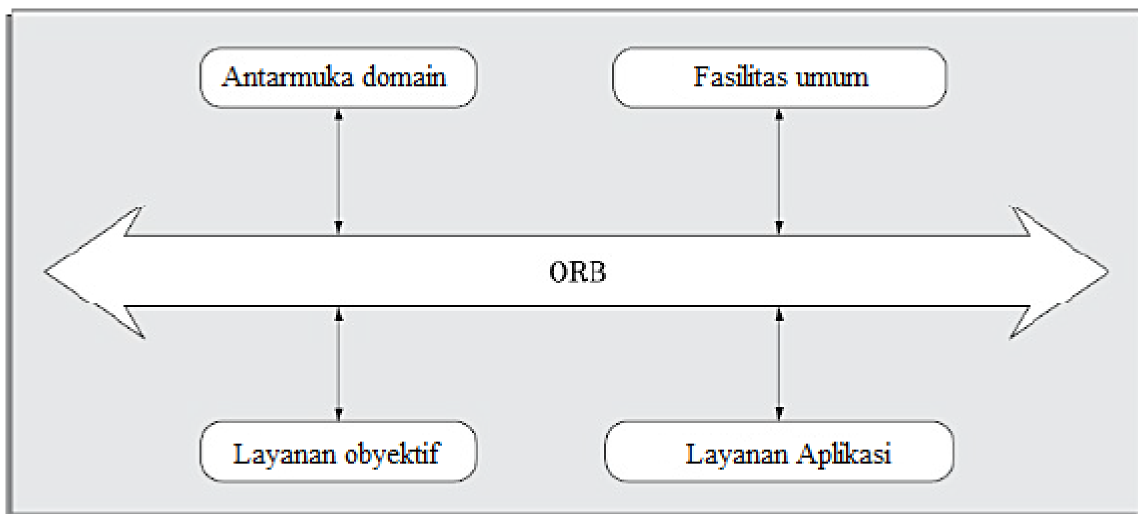
CORBA dipromosikan oleh Object Management Group (OMG), sebuah konsorsium dari sejumlah besar industri komputer seperti IBM, HP, Digital, dll. Namun, OMG bukanlah badan standar. OMG sebenarnya tidak memiliki wewenang untuk membuat atau menegakkan standar. Itu hanya mencoba mempopulerkan solusi yang baik dengan harapan bahwa jika solusi menjadi sangat populer, pada akhirnya akan menjadi standar. COM/DCOM dipromosikan terutama oleh Microsoft.

15.3 CORBA

Common Object Request Broker Architecture (CORBA) adalah spesifikasi arsitektur standar untuk middleware. Menggunakan implementasi CORBA, klien dapat secara transparan meminta layanan dari objek server, yang dapat berada di mesin yang sama atau melintasi jaringan. CORBA mengotomatiskan banyak tugas pemrograman jaringan umum seperti pendaftaran objek, lokasi, dan aktivasi; meminta demultiplexing; pembingkaian dan penanganan kesalahan; parameter marshalling dan demarshalling; dan pengiriman operasi.

Model Referensi CORBA

Model referensi CORBA telah ditunjukkan pada Gambar 15.2.



Gambar 5.12 Model referensi COBRA

15.4 ORB

ORB juga dikenal sebagai **bus objek**, karena ORB mendukung komunikasi di antara berbagai komponen yang melekat padanya. Ini mirip dengan bus pada printed circuit board (PCB) di mana perbedaan komponen perangkat keras (IC) berkomunikasi. Perhatikan bahwa karena analogi ini, bahkan simbol bus dari domain perangkat keras digunakan untuk mewakili ORB (lihat Gambar 15.2). ORB menangani permintaan klien untuk layanan apa pun, dan bertanggung jawab untuk menemukan objek yang dapat mengimplementasikan permintaan, meneruskan parameternya, memanggil metodenya, dan mengembalikan hasil permintaan. Klien tidak harus mengetahui di mana objek server yang diperlukan berada, bahasa pemrogramannya, sistem operasinya, atau aspek lain apa pun yang bukan merupakan bagian dari antarmuka objek.

Antarmuka domain

Antarmuka ini menyediakan layanan yang berkaitan dengan domain aplikasi tertentu. Beberapa layanan domain telah digunakan, termasuk domain manufaktur, telekomunikasi, medis, dan keuangan.

Layanan objek : Ini adalah antarmuka domain-independen yang digunakan oleh banyak program objek terdistribusi. Misalnya, layanan yang menyediakan penemuan layanan lain yang tersedia hampir selalu diperlukan terlepas dari domain aplikasi. Dua contoh layanan objek yang memenuhi peran ini adalah sebagai berikut:

Layanan Penamaan: Ini memungkinkan klien untuk menemukan objek berdasarkan nama. Layanan penamaan juga disebut layanan halaman putih.

Layanan Perdagangan: Ini memungkinkan klien untuk menemukan objek berdasarkan properti mereka. Layanan perdagangan juga disebut layanan halaman kuning. Menggunakan layanan perdagangan layanan tertentu dapat dicari. Ini mirip dengan mencari layanan seperti bengkel mobil di direktori halaman kuning. Mungkin ada layanan lain yang dapat disediakan oleh layanan objek seperti layanan keamanan, layanan siklus hidup, dan sebagainya.

Fasilitas umum

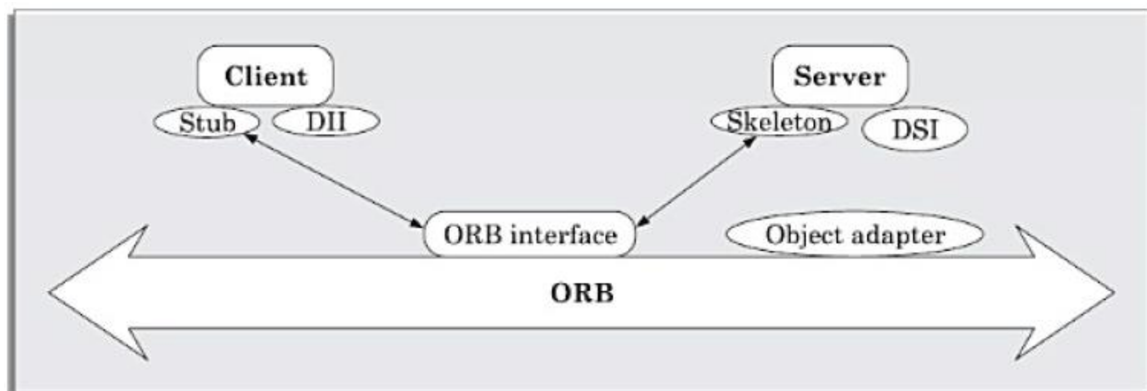
Seperti antarmuka layanan objek, antarmuka ini juga berorientasi horizontal, tetapi tidak seperti layanan objek, antarmuka ini berorientasi pada aplikasi pengguna akhir. Contoh fasilitas tersebut adalah fasilitas istributed document component facility (DDCF), fasilitas umum dokumen majemuk berdasarkan OpenDoc. DDCF memungkinkan presentasi dan pertukaran objek berdasarkan model dokumen, misalnya, memfasilitasi penautan objek spreadsheet ke dalam dokumen laporan.

Antarmuka aplikasi

Ini adalah antarmuka yang dikembangkan secara khusus untuk aplikasi tertentu.

Arsitektur CORBA ORB

Representasi Gambar 15.3 disederhanakan karena tidak menunjukkan berbagai komponen ORB. Sekarang mari kita bahas komponen penting dari arsitektur CORBA dan bagaimana mereka beroperasi. ORB harus mendukung sejumlah besar fungsi agar dapat beroperasi secara konsisten dan efektif. Dalam desain ORB yang dipikirkan dengan cermat, ORB mengimplementasikan banyak fungsi ini sebagai modul yang dapat dipasang untuk menyederhanakan desain dan implementasi ORB dan membuatnya efisien.



Gambar 15.2 Rancangan COBRA ORB

ORB

Komponen CORBA yang paling mendasar adalah object request broker (ORB) yang tugasnya memfasilitasi komunikasi antar objek. Tanggung jawab utama ORB adalah mengirimkan permintaan klien ke server dan mendapatkan respons kembali ke klien. ORB mengabstraksi kompleksitas permintaan layanan di seluruh jaringan dan membuat permintaan layanan oleh klien menjadi mulus dan mudah. ORB menyederhanakan pemrograman terdistribusi dengan memisahkan klien dari detail pemanggilan layanan. Ini membuat permintaan klien tampak seperti panggilan prosedur lokal. Ketika klien memanggil

operasi, ORB bertanggung jawab untuk menemukan implementasi objek, mengaktifkannya secara transparan jika: diperlukan, mengirimkan permintaan ke objek, dan mengembalikan respons apa pun ke pemanggil. ORB memungkinkan objek untuk menyembunyikan detail implementasinya dari klien. Berbagai aspek program yang disembunyikan (disarikan) dari klien termasuk bahasa pemrograman, sistem operasi, perangkat keras host, dan lokasi objek.

Rintisan dan kerangka

Menggunakan implementasi CORBA, klien dapat berkomunikasi ke server dalam dua cara—dengan menggunakan stub atau dengan menggunakan dynamic invocation interface (DII). Rintisan membantu pemanggilan layanan statis, di mana klien meminta layanan tertentu menggunakan parameter yang diperlukan. Dalam pemanggilan layanan dinamis, klien tidak perlu mengetahui sebelumnya tentang parameter yang diperlukan dan ini ditentukan pada saat run time. Meskipun pemanggilan layanan dinamis lebih fleksibel, pemanggilan layanan statis lebih efisien daripada pemanggilan layanan dinamis. Permintaan layanan oleh klien melalui rintisan cocok ketika antarmuka antara klien dan server diperbaiki dan tidak berubah seiring waktu. Jika antarmuka diketahui sebelum mulai mengembangkan klien dan bagian server, maka stub dapat digunakan secara efektif untuk pemanggilan layanan. Bagian rintisan berada di komputer klien dan bertindak sebagai proxy untuk server yang mungkin berada di komputer jarak jauh. Itulah alasan mengapa rintisan juga dikenal sebagai proxy.

Adaptor objek

Permintaan layanan melalui dynamic invocation interface (DII) secara transparan mengakses repositori antarmuka (OA). Ketika sebuah objek dibuat, ia mendaftarkan informasi tentang dirinya sendiri dengan OA. DII mendapatkan informasi yang relevan dari IR dan memberi tahu klien tentang antarmuka yang digunakan.

Implementasi CORBA

Ada beberapa implementasi CORBA yang tersedia untuk digunakan. Berikut ini adalah beberapa yang populer.

- Visibroker adalah perangkat lunak dari Borland yang mungkin merupakan implementasi CORBA paling populer. Browser Netscape mendukung Visibroker. Oleh karena itu, aplikasi CORBA dapat dijalankan dengan menggunakan web browser Netscape. Dengan kata lain, browser Netscape dapat bertindak sebagai klien untuk aplikasi CORBA. Netscape sangat populer dan ada beberapa juta salinan yang diinstal pada desktop di seluruh dunia.
- Orbix dari teknologi Iona.
- Java IDL.

Pengembangan Perangkat Lunak di CORBA

Mari kita periksa bagaimana perangkat lunak dapat dikembangkan di CORBA. Sebelum mengembangkan aplikasi client-server, solusinya dibagi menjadi dua bagian yaitu bagian client dan bagian serv. Selanjutnya, antarmuka klien dan server yang tepat ditentukan. Untuk menentukan antarmuka, interface definition language (IDL) digunakan. IDL sangat mirip dengan C++ dan Java kecuali bahwa ia tidak memiliki pernyataan yang dapat dieksekusi. Menggunakan IDL hanya antarmuka data antara klien dan server yang dapat ditentukan. Ini mendukung pewarisan sehingga antarmuka dapat digunakan kembali dalam aplikasi yang sama atau di berbagai aplikasi. Ini juga mendukung pengecualian.

Setelah antarmuka klien-server ditentukan dalam IDL, kompiler IDL digunakan untuk mengkompilasi spesifikasi IDL. Bergantung pada apakah bahasa target di mana aplikasi akan dikembangkan adalah Java, C++, C, dll., Kompiler IDL yang berbeda seperti IDL2Java, IDL2C++, IDL2C dll. dapat digunakan sesuai kebutuhan. Ketika spesifikasi IDL dikompilasi, itu menghasilkan kode kerangka untuk rintisan dan kerangka. Rintisan dan kerangka berisi definisi antarmuka dan hanya badan metode yang perlu ditulis oleh programmer yang mengembangkan komponen.

Komunikasi antar-ORB

Awalnya, CORBA hanya dapat mengintegrasikan komponen yang berjalan pada LAN yang sama. Namun, pada aplikasi tertentu, komponen aplikasi yang berbeda harus dijalankan di jaringan yang berbeda. Kekurangan CORBA 1.X ini telah dihapus oleh CORBA 2.0. CORBA 2.0 mendefinisikan standar interoperabilitas umum. General inter-orb protocol (GIOP) adalah meta-protokol abstrak. Ini menentukan sintaks transfer standar dan satu set format pesan untuk permintaan objek. GIOP adalah dirancang untuk bekerja pada banyak protokol transport yang berbeda. Dalam implementasi terdistribusi, setiap ORB harus mendukung GIOP yang dipetakan ke transportasi lokalnya. GIOP dapat digunakan oleh hampir semua transportasi aliran byte berorientasi koneksi. GIOP secara populer diimplementasikan pada TCP/IP yang dikenal sebagai internet inter-ORB protocol (IIOP).

15.5 COM/DCOM

COM

Gagasan utama dalam component object model(COM) adalah bahwa vendor yang berbeda dapat menjual komponen biner. Aplikasi dapat dikembangkan dengan mengintegrasikan komponen off-the-shelf. COM dapat digunakan untuk mengembangkan aplikasi komponen pada satu komputer. Konsep yang digunakan sangat mirip dengan CORBA. Komponen tersebut dikenal sebagai objek biner. Ini dapat dihasilkan menggunakan bahasa seperti Visual Basic, Delphi, Visual C++ dll. Bahasa-bahasa ini memiliki fitur yang diperlukan untuk membuat komponen COM. Komponen COM adalah objek biner dan ada dalam bentuk .exe atau .dll (dynamic link library). Komponen .exe memiliki keberadaan yang terpisah. Tetapi komponen .dll COM adalah server dalam proses, yang ditautkan ke suatu proses. Misalnya, ActiveX adalah server tipe dll, yang dimuat di sisi klien.

DCOM

Distributed component object model (DCOM) adalah perpanjangan dari component object model (COM). Pembatasan bahwa klien dan server berada di komputer yang sama dilonggarkan di sini. Jadi, DCOM dapat beroperasi di komputer berjaringan. Menggunakan DCOM, pengembangan mudah dibandingkan dengan CORBA. Banyak kerumitan tersembunyi dari programmer.

15.6 *SERVICE ORIENTED ARCHITECTURE* (SOA)

Prinsip orientasi layanan berakar pada perancangan berorientasi objek. Banyak yang mengklaim bahwa orientasi layanan akan menggantikan orientasi objek; yang lain berpikir bahwa keduanya adalah paradigma yang saling melengkapi. SOA memandang perangkat lunak sebagai menyediakan satu set layanan. Setiap layanan terdiri dari layanan yang lebih kecil. Mari kita pahami dulu apa itu layanan perangkat lunak. Layanan diimplementasikan dan

disediakan oleh komponen untuk digunakan oleh developer aplikasi. Layanan adalah perilaku yang ditentukan secara kontraktual. Artinya, komponen yang menyediakan layanan menjamin bahwa perilakunya sesuai dengan spesifikasi. Beberapa contoh layanan adalah sebagai berikut, Mengisi aplikasi online, melihat laporan mutasi bank online, dan melakukan pemesanan online. Layanan yang berbeda dalam aplikasi berkomunikasi satu sama lain.

Layanan mandiri. Artinya, suatu layanan tidak bergantung pada konteks atau keadaan layanan lainnya. Aplikasi yang mengintegrasikan layanan yang berbeda bekerja dalam arsitektur sistem terdistribusi. Ide utama di balik SOA adalah untuk membangun aplikasi dengan menyusun layanan perangkat lunak.

SOA terutama memanfaatkan Internet dan memunculkan standarisasi untuk interoperabilitas di antara berbagai layanan. Aplikasi dibangun menggunakan layanan yang tersedia di Internet, dan hanya menulis yang hilang. Ada beberapa kesamaan antara layanan dan komponen, yaitu sebagai berikut:

- **Reuse**: Baik komponen dan layanan digunakan kembali di beberapa aplikasi.
- **Generik**: Komponen dan layanan biasanya cukup umum untuk berguna untuk berbagai aplikasi.
- **Composable**: Baik layanan dan komponen terintegrasi bersama untuk mengembangkan aplikasi.
- **Dienkapsulasi**: Baik komponen dan layanan tidak dapat diselidiki melalui antarmuka mereka.
- **Pengembangan dan pembuatan versi independen:** Baik komponen dan layanan dikembangkan secara independen oleh vendor yang berbeda dan juga terus berkembang secara independen.
- **Kopling longgar**: Kedua aplikasi yang dikembangkan menggunakan paradigma komponen dan paradigma SOA memiliki kopling longgar yang melekat padanya.

Namun, ada beberapa perbedaan antara komponen dan paradigma SOA, yaitu sebagai berikut:

- Granularity (ukuran) layanan dalam paradigma SOA seringkali 100 hingga 1.000 kali lebih besar daripada komponen paradigma komponen.
- Layanan dapat dikembangkan dan dihosting di mesin terpisah.
- Biasanya komponen dalam paradigma komponen dibeli untuk digunakan sesuai kebutuhan (kepemilikan). Di sisi lain, layanan biasanya tersedia dalam pengaturan bayar per penggunaan.

Alih-alih layanan menyematkan panggilan satu sama lain dalam kode sumbernya, layanan menggunakan protokol yang terdefinisi dengan baik yang menjelaskan bagaimana layanan dapat berbicara satu sama lain. Arsitektur ini memfasilitasi pakar proses bisnis untuk menyesuaikan aplikasi sesuai kebutuhan. Untuk memenuhi kebutuhan bisnis baru, pakar proses bisnis dapat menghubungkan dan mengurutkan layanan dalam proses yang dikenal sebagai orkestrasi.

SOA menargetkan potongan fungsionalitas yang cukup besar untuk dirangkai untuk membentuk layanan baru. Artinya, layanan besar dapat dikembangkan dengan mengintegrasikan layanan perangkat lunak yang ada. Semakin besar potongannya, semakin sedikit antarmuka yang dibutuhkan. Ini mengarah pada perkembangan yang lebih cepat. Namun, potongan yang sangat besar mungkin terbukti sulit untuk digunakan kembali.

Service-oriented Architecture (SOA): Nitty Gritty

Paradigma SOA menggunakan layanan yang dapat di-host di komputer yang berbeda. Komputer dan layanan yang berbeda mungkin berada di bawah kendali pemilik yang berbeda. Untuk memfasilitasi pengembangan aplikasi, SOA harus menyediakan sarana untuk menawarkan, disc over, berinteraksi dengan dan menggunakan kemampuan layanan untuk mencapai hasil yang diinginkan.

SOA melibatkan layanan plug-in secara statis dan dinamis untuk membangun perangkat lunak. SOA player—BEA Aqua logic, Oracle Web services manager, HP Systinet Registry, MS .Net, IBM Web Sphere, Iona Artrix, Java composite application suite. Layanan web dapat digunakan untuk mengimplementasikan arsitektur berorientasi layanan. Layanan web dapat membuat blok bangunan fungsional dapat diakses melalui protokol Internet standar terlepas dari platform dan bahasa pemrograman.

Salah satu asumsi utama SOA adalah bahwa sekali pasar untuk layanan berkembang, layanan dapat dibeli untuk mengembangkan aplikasi baru. Untuk membangun aplikasi, seseorang akan menggunakan layanan siap pakai dan mungkin membangun beberapa. Ketika layanan digunakan di sejumlah besar aplikasi, secara otomatis kualitas akan meningkat dan juga harga akan berkurang. Ketika sebuah layanan digunakan oleh sejumlah besar aplikasi, biaya penggunaan layanan tersebut menjadi mendekati nol. Dengan demikian biaya pembuatan aplikasi yang menggunakan layanan yang banyak digunakan juga akan mendekati nol, karena semua layanan perangkat lunak yang diperlukan sudah ada dan biayanya mendekati nol, hanya orkestrasi layanan ini yang diperlukan untuk menghasilkan aplikasi.

15.7 *SOFTWARE AS A SERVICE* (SAAS)

Dalam konteks ini, SaaS membuat alasan untuk membayar per penggunaan perangkat lunak daripada memiliki perangkat lunak untuk digunakan. SaaS adalah model pengiriman perangkat lunak dan melibatkan pelanggan untuk membayar perangkat lunak apa pun per satuan waktu penggunaan, dengan harga yang mencerminkan penawaran dan permintaan pasar.

Seperti yang bisa kita lihat, SaaS menggeser “kepemilikan” perangkat lunak dari pelanggan ke penyedia layanan. Pemilik perangkat lunak menyediakan pemeliharaan, operasi teknis harian, dan dukungan untuk perangkat lunak. Layanan diberikan kepada klien berdasarkan jumlah penggunaan. Penyedia layanan adalah vendor yang meng-host perangkat lunak dan memungkinkan pengguna mengeksekusi biaya permintaan per unit penggunaan. Ini juga mengalihkan tanggung jawab untuk manajemen perangkat keras dan perangkat lunak dari pelanggan ke penyedia. Biaya penyediaan layanan perangkat lunak berkurang karena semakin banyak pelanggan yang berlangganan layanan ini. Elemen outsourcing dan penyediaan layanan aplikasi tersirat dalam model SaaS. Juga, itu membuat perangkat lunak dapat diakses oleh sejumlah besar pelanggan yang tidak mampu membeli perangkat lunak secara langsung. Targetkan "ekor panjang" pelanggan kecil.

Jika kita membandingkan SaaS dengan SOA, kita dapat mengamati bahwa SaaS adalah model pengiriman perangkat lunak, sedangkan SOA adalah model konstruksi perangkat lunak. Terlepas dari perbedaan yang signifikan, baik SOA dan SaaS mendukung model arsitektur yang terkait erat. SaaS dan SOA saling melengkapi. SaaS membantu menawarkan komponen untuk

digunakan SOA. SOA membantu membantu mewujudkan SaaS dengan cepat. Juga, enabler utama SaaS dan SOA adalah teknologi Internet dan layanan web.

15.8 RINGKASAN

- Beberapa asumsi dasar perangkat lunak berubah. Ini mengarah ke beberapa paradigma berbeda untuk pengembangan dan pengiriman perangkat lunak.
- Pengembangan berbasis komponen diharapkan dapat mengurangi waktu pengembangan, biaya dan pada saat yang sama meningkatkan kualitas.
- SaaS mengubah cara perangkat lunak dikirimkan.
- SOA secara fundamental akan mengubah cara kita membangun sistem perangkat lunak. Dalam paradigma SOA, aplikasi dapat dibangun dengan mengatur layanan yang ada, dan hanya menulis yang hilang.

15.9 LATIHAN

1. Pilih opsi yang benar:
   (a) Apa kemungkinan alasan di balik popularitas baru-baru ini dari gaya pengembangan perangkat lunak client-server?
      (i) Komputer menjadi kecil, terdesentralisasi, dan murah
      (ii) Jaringan menjadi terjangkau, andal, dan efisien
      (iii) Sistem client-server membagi pekerjaan komputasi di antara banyak mesin yang terpisah
      (iv) Semua hal di atas
   (a) Manakah dari fungsi berikut yang dilakukan oleh middleware?
      (i) Mengidentifikasi server baik dari id atau jenis layanannya
      (ii) Konversi antara protokol klien dan protokol server
      (iii) Menyampaikan permintaan klien ke server dan respons server ke klien
      (iv) Semua hal di atas
   (b) Manakah dari fungsi berikut yang dilakukan oleh broker permintaan objek (ORB)?
      (i) Mengirimkan permintaan klien ke server dan mendapatkan respons kembali ke klien
      (ii) Lokasi dan kemungkinan aktivasi objek jarak jauh
      (iii) Definisi antarmuka
      (iv) Semua hal di atas
   (c) Sebelum mengembangkan aplikasi client-server di CORBA, antarmuka antara bagian klien dan server harus ditentukan menggunakan:
      (i) Bahasa definisi antarmuka
      (ii) Antarmuka pemanggilan dinamis
      (iii) ORB
      (iv) Tidak satu pun di atas
   (d) Dalam CORBA jika antarmuka server diketahui tidak berubah terhadap waktu, maka akan lebih efisien untuk digunakan
      (i) Rintisan dan kerangka
      (ii) DSI dan DII
      (iii) RMI
      (iv) Tidak satu pun di atas

2. Apa kelebihan perangkat lunak client-server dibandingkan dengan perangkat lunak monolitik? Juga, mengidentifikasi kelemahan dari perangkat lunak client-server.
3. Apa yang dimaksud dengan Common Object Request Broker Architecture (CORBA)? Menjelaskan arsitektur CORBA.
4. Sebutkan tiga fungsi dari Object Request Broker (ORB).
5. Sebutkan setidaknya tiga ORB komersial.
6. Jelaskan apa itu rintisan.
7. Jelaskan Dynamic Invocation Interface (DII) di CORBA.
8. Jelaskan apa yang dimaksud dengan Component Object Model (COM) dan distributed component object model (DCOM).
9. Menjelaskan komunikasi antar ORB.
10. Apa keuntungan dari pengembangan perangkat lunak client-server?
11. Mengapa arsitektur two-tier bukan arsitektur client-server yang praktis? Bagaimana arsitektur tiga tingkat mengatasi masalah arsitektur dua tingkat.
12. Apa fungsi middleware dalam arsitektur tiga tingkat? Sebutkan dua standar middleware yang populer.
13. Bagaimana Dynamic Invocation Interface (DII) mengetahui format data atau data persis apa yang dibutuhkanoleh server untuk menyediakan layanan dan bagaimana klien mengenali data tersebut?
14. Apa itu protokol General Inter-ORB Protocol (GIOP)? Apa saja fitur GIOP?
15. Tandai pernyataan berikut sebagai Benar atau Salah. Justifikasi jawaban Anda.
    (a) Toleransi kesalahan lebih sulit diberikan dalam aplikasi monolitik dibandingkan dengan implementasi client-servernya.
    (b) Pengembangan perangkat lunak berbasis client-server lebih aman daripada perangkat lunak monolitik.
    (c) Arsitektur client-server dua tingkat dapat digunakan untuk mengembangkan sistem berbasis komponen komersial secara efektif.
    (d) Komponen adaptor objek di CORBA bertanggung jawab untuk menerjemahkan format data klien ke dalam format data server dan sebaliknya.
    (e) CORBA adalah nama produk perangkat lunak yang memfasilitasi pengembangan solusi client-server.
    (f) Solusi client-server berbasis CORBA dibatasi untuk dijalankan pada satu Local Area Network (LAN).
    (g) Menggunakan Internet Inter-ORB Protocol (IIOP), browser web seperti Netscape dapat berfungsi sebagai klien CORBA.
16. Bedakan antara solusi perangkat lunak monolitik dan client-server. Berikan contoh untuk masing-masing.

DAFTAR PUSTAKA

Albrecht, A.J. and J.E. Gaffney, “Software function, lines of code, and development effort prediction: a software science validation,” IEEE Trans. on Software Engineering, 9(6), pp. 639–647.

Alhir, Sinan Si., UML in Nutshell, OReily, 1998.

Boehm, B.W., Software Engineering Economics, Prentice Hall, Englewood Cliffs, New Jersey, 1981.

Boehm, Barry, B. Clark and E. Horowitz, et al. “Cost Models for Future Software Life Cycle Processes: COCOMO 2.0,” Annals of Software Engineering, 1995.

Booch, Grady, Object-Oriented Design with Applications, Benjamin Cummings, Menlo Park, California, 1991.

Booch, Rumbaugh, Jacobson, The UML User’s Guide, Addison-Wesley, Reading, Mass., 1999.

Brooks F., The Mythical Man-Month, Addison-Wesley, Reading, Mass., 1975. Constantine, L.L. and E. Yourdon, Structured Design, Prentice Hall, Englewood Cliffs, New Jersey, 1979.

Conte, S.D., H.E. Dunsmore and V.Y. Shen, Software Engineering Metrics and Models, Benjamin/Cummings Publishing Company Inc., Menlo Park, California, 1986.

Corrado Bohm and Giuseppe Jacopini, “Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules”, Communications of the ACM, 9(5), May 1966.

Crosby, Philip B., Quality is Free, McGraw-Hill, New York, 1979.

DeMarco, T., Structured Analysis and System Specification, Yourdon Press, New York, 1978.

Dijkstra, E.W., “Goto Statement Considered Harmful,” Communications of the ACM, 11(3), pp. 147–148, 1968. Eliason, Alan L., Systems Development Analysis and Implementation, Little Brown and Company (Canada) Ltd., 1987.

Fairley, Richard, Managing and Leading Software Project, IEEE Press, 2009.

Fowler, Martin, UML Distilled, 2nd ed., Addison-Wesley, Reading Mass., 2000.

Gane, C. and T. Sarson, Structured Systems Analysis, Prentice Hall, Englewood Cliffs, New Jersey, 1979.

George A., Miller, “The Magical Number Seven Plus or Minus Two: Some Limits on Our Capacity for Processing Information,” The Psychological Review, 63(2), pp. 8197, 1956.

Ghezzi, Carlo, Mehdi Jazayeri and Dino Mandrioli, Fundamentals of Software Engineering, Pearson Education Inc., 2003.

Guttag, J., “Notes on Type Abstraction,” IEEE Transactions on Software Engineering, 6(1), pp. 13–23, 1980.

Guttag, J., J. Horning, J.M. Wing, “The Larch Family of Specification Languages,” IEEE Software, 2(5), pp. 24–36, 1985.

Harel, D. et al., “Statemate: A working environment for the development of complex reactive systems,” IEEE Transactions on Software Engineering, 16(3), 403–414, April 1990.

Hatley, D. and I. Pirbhai, Strategies for Real-Time System Specifications, Dorset House, New York, 1987.

Hoare, C.A.R., “Programming Sorcery or Science?” IEEE Transactions on Software Engineering, pp. 5–16, April 1994.

Humphrey, Watts S., Introduction to the Personal Software Process, Addison Wesley, Longman, 1997. IEEE Recommended Practice for Software Requirements Specifications, IEEE Std. 830–1998. IEEE Standard Glossary of Software Engineering Terminology, IEEE Std. 610.12–1990.

Ince, Darrel, ISO 9001 and Software Quality Assurance, McGraw-Hill, New York, 1994.

Ince, David (Ed.), Software Quality and Reliability, Chapman and Hall Publishing, London, 1991.

Jackson, M.A., Principles of Program Design, Academic Press Inc., Orlando, Florida, 1975.

Jacobson, Booch, Rumbaugh, The Unified Software Development Process, Addison-Wesley, Reading Mass., 1999.

Jacobson, I. and Christerson, M., Object-oriented Software Engineering—A Use Case-Driven Approach, Addison-Wesley, England, 1992.

Jalote, Pankaj, An Integrated Approach to Software Engineering, Third Edition,

Jelinski, Z. and P. Moranda, “ Software Reliability Research,” in Freiberger W. (Ed.), Statistical Computer Reliability Evaluation, Academic Press, pp. 465– 484, 1972.

Jensen, R.W., “A Comparison of the Jensen and COCOMO Schedule and Cost Estimation Models,” in Proc. of International Society of Parametric Analysis, pp. 96–106, 1984.

Jones, C.B., Software Development: A rigorous approach, Prentice Hall, Englewood Cliffs, New Jersey, 1980.

Lamb, David Alex, Software Engineering Planning for Change, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

Larman, Craig, Applying UML and Patterns—An Introduction to Object- Oriented Analysis and Design, Pearson Press, 1998.

Lawrence Pfleeger, Shari, Software Engineering Theory and Practice, Fourth Edition, Prentice Hall, Englewood Cliffs, New Jersey, 2009.

Leathrum, J.F., Foundations of Software Design, Reston Publishing Company, Virginia, USA, 1983.

Lehman, M.M. and L.A. Belady, "Programs Life Cycles and Laws of Software Evolution," Proceedings of IEEE, pp. 1060–1076, September 1980.

Ludolph, Frank, Model-based user interface designSuccessive transformations of a task object model, in User interface designbridging the gap from user requirements to design, CRC Press, 1998.

Martin, James and Carma McClure, Structured Techniques: The Basis for CASE, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

Martin, James and James J. Odell, Principles of Object-Oriented Analysis and Design, Prentice Hall, Englewood Cliffs, New Jersey, 1992.

McCabe, T., "A Complexity Measure," IEEE Transactions on Software Engineering, 4(2) December 1976.

Miller, G.A., "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," in the Psychological Review, 63(2), pp. 81-97, March 1956.

Mund, G.B., R. Mall and S. Sarkar, "An Efficient Dynamic Program Slicing Technique," Journal of Information and Software Technology, Elsevier Press, 44(2), pp. 123–132, March 2002.

Nielson, J. and R.L. Mack, Usability Inspection Methods, John Wiley, 1994.

Orr, K., Structured Requirements Definition, Ken Orr and Associates, Australia, 1981.

Parnas, D., "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM, 15(2), pp. 105–358, 1972.

Pressman, Roger S., Software Engineering, 7th Ed., McGraw-Hill, New York, 2009.

Putnam, L.H., "A General Empirical Solution to Macro Software Sizing and Estimation Problem," IEEE Transactions on Software Engineering, 4(3), pp. 345–361, 1978.

Rosenberg D., Use Case-Driven Object Modelling with UML, Addison- Wesley, Reading, Mass., 2000.

Rumbaugh, J., I. Jacobson and G. Booch, The UML Reference Manual, Addison-Wesley, Reading, Mass., 1999.

Rumbaugh, J., M. Blaha, et al., Object-Oriented Modelling and Design, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

Sackman H., W.J. Erikson and E.E. Grant “Exploratory Experimentation Studies Comparing On-line and Off-line Programming Performance,” Communications of the ACM, 11(1), pp. 3–11, 1968.

Scheifler, Robert W., Jim Gettys and R. Newman, X Window System—C Library and Protocol Reference—C Library and , DEC Press, Bedford, Mass., 1988.

Shlaer, Sally and Stephen J. Mellor, Object Lifecycles: Modeling the World in States, Prentice Hall, Englewood Cliffs, New Jersey, 1992.

Smith, David J. and Kenneth B. Wood, Engineering Quality Software, Elsevier Science Publishing, New York, 1987.

Somerville, Ian, Software Engineering, 9th ed., Addison-Wesley, Reading, Mass., 1992.

Ward, P. and S. Mellor, Structured Development of Real-Time Systems, Prentice Hall, Englewood Cliffs, New Jersey, 1985.

Warnier, J.D., Logical Construction of Programs, Van Nostrand Reinhold, New York, 1977.

Wirfs-Brock, Rebecca, Brian Wilkerson, Lauren Wiener, Designing Object Oriented Software, Prentice Hall, Englewood Cliffs, New Jersey, 1990.

Wirth, N., “Program Development by Stepwise Refinement,” Communications of the ACM, 14(4), pp. 221–227, 1971.

Rekayasa Perangkat Lunak

Migunani, S.Kom, M.Kom

Biodata Penulis

Penulis yang Lahir pada tahun 1976 ini memang memiliki minat yang tentang IT sejak lama. Penulis juga sudah banyak melakukan Seminar dan Workshop tentang perkembangan Ilmu Komputer. Penulis kelahiran semarang yang bergelar Magister ini sekarang sedang melakukan studinya untuk menyelesaikan gelar Doktornya.

Selain mengajar di Universitas Sains Dan Teknologi Komputer Fakultas Studi Akademik jurusan Teknik Informatika ini, kegiatan penulis memberkan pelatihan atau workshop dan seminar-seminar dibidang teknologi informasi. Beberapa buku yang sudah penulis buat digunakan untuk bahan ajar pengajar di Universitas Sains Dan Teknologi Komputer. Penelitian pertama penulis tahun 2002 tentang Jurnal Teknik Informasi Dinamik sampai saat ini terhitung hingga puluhan Publikasi ilmiah yang sudah dikerjakan penulis. Penulis lulusan Magister Universitas Diponegoro ini mendapatkan penghargaan sebagai mahasiswa Cumlaude dan Terbaik tahun 2011 serta mendapatan pernghargaan The Best Of Fifth Paper Award ditahun yang sama.

Penulis juga menjadi beberapa narasumber dalam seminar nasional maupun internasional, seperti Visiting Lecturer from STEKOM University with Oles Honchar Dnipro National University (Ukraine) menjadi pembicara utama dalam seminar Internasional, dan mejadi narasumber webinar nasional "Peran Machine Learning Dalam Transformasi Digital, Teori Dan Penerapannya" dan masih banyak Seminar lagi yang penulis lakukan. Banyak Organisasi profesi ilmiah yang diikuti penulis, tahun 2016 Asosiasi Perguruan Tinggi Komputer (APTIKOM) sampai Sekarang, tahun 2017 hingga sekarang Ikatan Dosen Republik Indonesia, tahun 2010 Paguyuban Wakil Ketua / Rektor III, dan yang paling baru merupakan salah satu anggota Perkumpulan Dosen Perguruan Tinggi Nusantara (Pdptn) Provinsi Jawa Tengahpada tahun 2020 hingga sekarang. Cita-cita penulis adalah memajukan dunia IT melalui menulis dan mengajar. Motto hidup penulis “Hidup akan lebih berarti saat termotivasi, memiliki arah dan tujuan, serta mengejarnya”

PENERBIT :
YAYASAN PRIMA AGUS TEKNIK
Jl. Majapahit No. 605 Semarang
Telp. (024) 6723456. Fax. 024-6710144
Email : penerbit_ypat@stekom.ac.id

Rekayasa Perangkat Lunak

Migunani, S.Kom, M.Kom

PENERBIT :
YAYASAN PRIMA AGUS TEKNIK
Jl. Majapahit No. 605 Semarang
Telp. (024) 6723456. Fax. 024-6710144
Email : penerbit_ypat@stekom.ac.id