engenharia de software 6a - ed - Roger S. Pressman - Portugues - Brasil

Redes de Computadores

Universidade Paulista (UniP)

371 pag.

Document shared on www.docsity.com

ENGENHARIA DE SOFTWARE

Sexta Edição

Roger S. Pressman

Cleber

Parei PÆgina 30

Document shared on www.docsity.com

Engenharia de Software

Document shared on www.docsity.com

Roger S. Pressman é uma autoridade internacionalmente reconhecida nas tecnologias de melhoria do processo de software e engenharia de software. Por mais de três décadas trabalhou como engenheiro de software, gerente, professor, autor e consultor concentrando-se em assuntos de engenharia de software.

Como profissional e gerente da indústria, o Dr. Pressman trabalhou no desenvolvimento de sistemas CAD/CAM para aplicações avançadas de engenharia e de fabricação. Ele também ocupou posições de responsabilidade em programação científica e de sistemas.

Após receber um título de Ph. D. em engenharia na Universidade de Connecticut, o Dr. Pressman começou a dedicar-se à vida acadêmica, tornando-se Bullard Associate Professor of Computer Engineering na Universidade de Bridgeport e diretor do Centro de Projeto e Fabricação Apoiados por Computador (CAD/CAM — Computer Aided Design/Computer Aided Manufacturing) da Universidade.

Dr. Pressman é atualmente presidente da R. S. Pressman & Associates, Inc., uma empresa de consultoria especializada em métodos de engenharia de software e treinamento. Atua como consultor principal; projetou e desenvolveu *Essential Software Engineering*, um currículo completo em vídeo sobre engenharia de software e *Process Advisor*, um sistema auto dirigido para aperfeiçoamento do processo de software. Ambos os produtos são usados por milhares de empresas em todo o mundo. Mais recentemente, trabalhou em colaboração com a QAI Índia para desenvolver uma "eSchool" abrangente de engenharia de software baseada na Internet.

O Dr. Pressman escreveu muitos trabalhos técnicos, é contribuinte regular de periódicos da indústria e é autor de seis livros técnicos. Além de *Engenharia de Software*, escreveu o livro vencedor de um prêmio *A Manager's Guide to Software Engineering* (McGraw-Hill); *Making Software Engineering Happen*, o primeiro livro a tratar de problemas críticos de gestão associados com o aperfeiçoamento de processos de software; e *Software Shock*, que focaliza software e seu impacto nos negócios e na sociedade. Dr. Pressman participou do comitê editorial de vários periódicos da área e durante muitos anos foi editor da coluna "Manager" no *IEEE Software*.

Dr. Pressman é um palestrante bastante conhecido, tendo sido destaque em importantes conferências da indústria. É membro da ACM, IEEE, Tau Beta Pi, Phi Kappa Phi, Eta Kappa Nu e Pi Tau Sigma.

No aspecto pessoal, Dr. Pressman vive no sul da Flórida com sua esposa Bárbara. Atleta durante grande parte da vida, é um respeitado jogador de tênis (NTRP 4.5) e jogador de golfe com handicap de um dígito. Escreveu duas novelas, *The Aymara Bridge* e *The Puppeteer*.

Prefácio XXV
Recursos deste Livro XXIX

CAPÍTULO 1 SOFTWARE E ENGENHARIA DE SOFTWARE 1
1.1 O Papel Evolutivo do Software 2
1.2 Software 4
1.3 A Natureza Mutável do Software 6
1.4 Software Legado 8
1.4.1 A Qualidade do Software Legado 8
1.4.2 Evolução de Software 9
1.5 Mitos do Software 10
1.6 Como Tudo Começa 12
1.7 Resumo 13
REFERÊNCIAS BIBLIOGRÁFICAS 13
PROBLEMAS E PONTOS A CONSIDERAR 13
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS

PARTE UM — O PROCESSO DE SOFTWARE 15

CAPÍTULO 2 PROCESSO: UMA VISÃO GENÉRICA 16
2.1 Engenharia de Software — Uma Tecnologia em Camadas 17
2.2 Um Arcabouço de Processo 18
2.3 O CMMI (Capability Maturity Model Integration) 21
2.4 Padrões de Processo 24
2.5 Avaliação de Processo 27
2.6 Modelos de Processo Pessoal e de Equipe 29
2.6.1 PSP (Personal Software Process — Processo Pessoal de Software) 29
2.6.2 TSP (Team Process Software — Processo de Equipe de Software) 30
2.7 Tecnologia de Processo 31
2.8 Produto e Processo 32
2.9 Resumo 33
REFERÊNCIAS BIBLIOGRÁFICAS 34
PROBLEMAS E PONTOS A CONSIDERAR 34
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 35

CAPÍTULO 3 MODELOS PRESCRITIVOS DE PROCESSO 37
3.1 Modelos Prescritivos 38
3.2 O Modelo em Cascata 38

Document shared on www.docsity.com

3.3 Modelos Incrementais de Processo 39
3.3.1 O Modelo Incremental 40
3.3.2 O Modelo RAD 41
3.4 Modelos Evolucionários de Processo de Software 42
3.4.1 Prototipagem 42
3.4.2 O Modelo Espiral 44
3.4.3 O Modelo de Desenvolvimento Concorrente 46
3.4.4 Um Comentário Final sobre Processos Evolucionários 47
3.5. Modelos Especializados de Processo 48
3.5.1 Desenvolvimento Baseado em Componentes 48
3.5.2 O Modelo de Métodos Formais 49
3.5.3 Desenvolvimento de Software Orientado a Aspectos 49
3.6 O Processo Unificado 51
3.6.1 Um Breve Histórico 51
3.6.2 Fases do Processo Unificado 52
3.6.3 Produtos de Trabalho do Processo Unificado 54
3.7 RESUMO 55
REFERÊNCIAS BIBLIOGRÁFICAS 55
PROBLEMAS E PONTOS A CONSIDERAR 56
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 57

CAPÍTULO 4 DESENVOLVIMENTO ÁGIL 58
4.1 O Que É Agilidade? 59
4.2. O Que É Um Processo Ágil? 60
4.2.1 A Política de Desenvolvimento Ágil 61
4.2.2 Fatores Humanos 61
4.3 Modelos Ágeis de Processo 63
4.3.1 Extreme Programming (XP) 63
4.3.2. DAS — Desenvolvimento Adaptativo de Software (Adaptative Software Development — ASD) 66
4.3.3 DSDM (Dynamic Systems Development Method — Método de Desenvolvimento Dinâmico de Sistemas) 68
4.3.4. Scrum 69
4.3.5. Crystal 71
4.3.6 FDD (Feature Driven Development — Desenvolvimento Guiado por Características) 71
4.3.7. Modelagem Ágil (Agile Modeling — AM) 72
4.4 Resumo 74
REFERÊNCIAS BIBLIOGRÁFICAS 74
PROBLEMAS E PONTOS A CONSIDERAR 75
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 76

PARTE DOIS — PRÁTICA DE ENGENHARIA DE SOFTWARE 77

CAPÍTULO 5 PRÁTICA:UMA VISÃO GENÉRICA 78
5.1. Prática de Engenharia de Software 79
5.1.1 A Essência da Prática 79
5.1.2 Princípios Centrais 80

5.2 Práticas de Comunicação 82
5.3 Práticas de Planejamento 84
5.4 Prática de Modelagem 87
5.4.1 Princípios da Modelagem de Análise 87
5.4.2 Princípios de Modelagem de Projeto 89
5.5 Prática de Construção 91
5.5.1 Princípios e Conceitos de Codificação 91
5.5.2 Princípios de Teste 92
5.6 Implantação 94
5.7 Resumo 95
REFERÊNCIAS BIBLIOGRÁFICAS 96
PROBLEMAS E PONTOS A CONSIDERAR 97
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 97

CAPÍTULO 6 ENGENHARIA DE SISTEMAS 99
6.1 Sistemas Baseados em Computador 100
6.2 A Hierarquia da Engenharia de Sistemas 101
6.2.1 Modelagem de Sistemas 102
6.2.2 Simulação do Sistema 104
6.3 Engenharia de Processo de Negócio: Panorama 104
6.4 Engenharia de Produto: Um Panorama 105
6.5 Modelagem de Sistemas 108
6.5.1 Modelagem Hatley-Pirbhai 108
6.5.2 Modelagem de Sistema com a UML 110
6.6 Resumo 113
REFERÊNCIAS BIBLIOGRÁFICAS 113
PROBLEMAS E PONTOS A CONSIDERAR 114
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 114

CAPÍTULO 7 ENGENHARIA DEREQUISITOS 116
7.1 Uma Ponte para o Projeto e a Construção 117
7.2 Tarefas da Engenharia de Requisitos 118
7.2.1. Concepção 118
7.2.2 Levantamento 118
7.2.3 Elaboração 119
7.2.4 Negociação 119
7.2.5 Especificação 120
7.2.6 Validação 120
7.2.7 Gestão de Requisitos 121
7.3 Início do Processo de Engenharia de Requisitos 122
7.3.1 Identificação dos Interessados 122
7.3.2 Reconhecimento de Diversos Pontos de Vista 122
7.3.3 Trabalho em Busca da Colaboração 123
7.3.4 Formulação das Primeiras Questões 123
7.4 Levantamento de Requisitos 124
7.4.1 Coleta Colaborativa de Requisitos 124

Document shared on www.docsity.com

7.4.2 Implantação da Função de Qualidade 127
7.4.3 Cenários de Usuários 129
7.4.4 Produtos de Trabalho do Levantamento 129
7.5 Desenvolvimento de Casos de Uso 129
7.6 Construção do Modelo de Análise 134
7.6.1 Elementos do Modelo de Análise 134
7.6.2 Padrões de Análise 137
7.7 Negociacão de Requisistos 138
7.8 Validação de Requisitos 140
7.9 Resumo 140
REFERÊNCIAS BIBLIOGRÁFICAS 141
PROBLEMAS E PONTOS A CONSIDERAR 141
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 142

CAPÍTULO 8 MODELAGEM DE ANÁLISE 144

8.1 Análise de Requisitos 145
8.1.1 Objetivos Gerais e Filosofia 145
8.1.2 Regras Práticas de Análise 146
8.1.3 Análise de Domínio 147
8.2 Abordagens de Modelagem de Análise 147
8.3 Conceitos de Modelagem de Dados 148
8.3.1 Objetos de Dados 148
8.3.2. Atributos de Dados 149
8.3.3 Relacionamentos 150
8.3.4 Cardinalidade e Modalidade 150
8.4 Análise Orientada a Objetos 152
8.5 Modelagem Baseada em Cenário 153
8.5.1 Escrita de Casos de Uso 153
8.5.2 Desenvolver um Diagrama de Atividade 157
8.5.3 Diagramas de Raias 158
8.6 Modelagem Orientada a Fluxo 159
8.6.1 Criação de um Modelo de Fluxo de Dados 160
8.6.2 Criação de um Modelo de Fluxo de Controle 162
8.6.3 A Especificação de Controle 163
8.6.4 A Especificação de Processo 164
8.7 Modelagem Baseada em Classe 165
8.7.1 Identificação de Classes de Análise 165
8.7.2 Especificação de Atributos 168
8.7.3 Definição das Operações 169
8.7.4 Modelagem Classe-Responsabilidade-Colaboração (CRC) 171
8.7.5 Associações e Dependências 176
8.7.6 Pacotes de Análise 176
8.8 Criação de Um Modelo Comportamental 177
8.8.1 Identificação dos Eventos no Caso de Uso 178
8.8.2 Representações de Estados 178
8.9 Resumo 181

REFERÊNCIAS BIBLIOGRÁFICAS 182
PROBLEMAS E PONTOS A CONSIDERAR 182
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 184

CAPÍTULO 9 ENGENHARIA DE PROJETO 185

9.1 Projeto No Contexto de Engenharia de Software 186
9.2 Processo de Projeto e Qualidade de Projeto 188
9.3 Conceitos de Projeto 190
9.3.1 Abstração 190
9.3.2 Arquitetura 191
9.3.3 Padrões 191
9.3.4 Modularidade 192
9.3.5 Ocultamento da Informação 193
9.3.6 Independência Funcional 193
9.3.7 Refinamento 194
9.3.8 Refabricação 194
9.3.9 Classes de Projeto 195
9.4 O Modelo de Projeto 197
9.4.1 Elementos de Projeto de Dados 198
9.4.2 Elementos do Projeto Arquitetural 198
9.4.3 Elementos de Projeto da Interface 199
9.4.4 Elementos de Projeto em Nível de Componente 200
9.4.5 Elementos de Projeto em Nível de Implantação 201
9.5 Projeto de Software Baseado em Padrão 202
9.5.1 Descrição de um Padrão de Projeto 202
9.5.2 Uso de Padrões no Projeto 203
9.5.3 Frameworks 203
9.6 Resumo 204
REFERÊNCIAS BIBLIOGRÁFICAS 204
PROBLEMAS E PONTOS A CONSIDERAR 205
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 206

CAPÍTULO 10 PROJETO ARQUITETURAL 207

10.1 Arquitetura de Software 208
10.1.1 O Que É Arquitetura? 208
10.1.2 Por Que a Arquitetura É Importante? 209
10.2 Projeto de Dados 209
10.2.1 Projeto de Dados Arquitetural 209
10.2.2 Projeto de Dados no Nível de componentes 210
10.3 Estilos e Padrões Arquiteturais 211
10.3.1 Uma Breve Taxinomia de Estilos Arquiteturais 212
10.3.2. Padrões Arquiteturais 215
10.3.3 Organização e Refinamento 216
10.4 Projeto Arquitetural 216
10.4.1 Representação do Sistema no Contexto 217
10.4.2 Definição de Arquétipos 218

Document shared on www.docsity.com

10.4.3 Refinar a Arquitetura em Componentes 218
10.4.4. Descrição das Instanciações do Sistema 220
10.5 Avaliação de Alternativas de Projeto Arquitetural 221
10.5.1 Um Método de Análise dos Compromissos da Arquitetura 221
10.5.2 Complexidade Arquitetural 223
10.5.3 Linguagens de Descrição Arquitetural 223
10.6 Mapear Fluxo de Dados para Uma Arquitetura de Software 224
10.6.1 Fluxo de Transformação 224
10.6.2 Fluxo de Transação 225
10.6.3 Mapeamento de Transformação 225
10.6.4 Mapeamento da Transação 231
10.6.5 Refinamento do Projeto Arquitetural 234
10.7 RESUMO 234
REFERÊNCIAS BIBLIOGRÁFICAS 235
PROBLEMAS E PONTOS A CONSIDERAR 235
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 236

CAPÍTULO 11 PROJETO NO NÍVEL DE COMPONENTES 238

11.1 O Que É Um Componente? 239
11.1.1 Uma Visão Orientada a Objetos 239
11.1.2 A Visão Convencional 241
11.1.3 Uma Visão Relacionada a Processo 243
11.2 Projeto de Componentes Baseados em Classes 243
11.2.1 Princípios Básicos de Projeto 243
11.2.2 Diretrizes para Projeto no Nível de Componente 246
11.2.3 Coesão 246
11.2.4 Acoplamento 248
11.3 Condução do Projeto no Nível de Componente 250
11.4 Linguagem de Restrição de Objeto 255
11.5 Projeto de Componentes Convencionais 256
11.5.1 Notação Gráfica de Projeto 257
11.5.2 Notação Tabular de Projeto 258
11.5.3 Linguagem de Projeto de Programas 259
11.5.4 Comparação da Notação de Projeto 260
11.6 RESUMO 261
REFERÊNCIAS BIBLIOGRÁFICAS 261
PROBLEMAS E PONTOS A CONSIDERAR 262
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 262

CAPÍTULO 12 PROJETO DE INTERFACE COM O USUÁRIO 264

12.1 As Regras de Ouro 265
12.1.1 Coloque o Usuário no Controle 265
12.1.2 Reduza a Carga de Memória do Usuário 266
12.1.3 Faça a Interface Consistente 266
12.2 Análise e Projeto de Interface com o Usuário 268
12.2.1 Modelos de Análise e de Projeto de Interface 268
12.2.2 O Processo 270

12.3 Análise de Interface
12.3.1 Análise do Usuário 271
12.3.2 Modelagem e Análise de Tarefas 272
12.3.3 Análise de Conteúdo de Mostrador 276
12.3.4 Análise do Ambiente de Trabalho 277
12.4 Passos do Projeto da Interface 277
12.4.1 Aplicação dos Passos do Projeto de Interface 278
12.4.2 Padrões de Projeto de Interface com o Usuário 280
12.4.3 Questões de Projeto 281
12.5 Avaliação de Projeto 284
12.6 Resumo 285
REFERÊNCIAS BIBLIOGRÁFICAS 286
PROBLEMAS E PONTOS A CONSIDERAR 286
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 287

CAPÍTULO 13 ESTRATÉGIAS DE TESTE DE SOFTWARE 288

13.1 Uma Abordagem Estratégica ao Teste de Software 289
13.1.1 Verificação e Validação 289
13.1.2 Organização do Teste de Software 290
13.1.3 Uma Estratégia de Teste de Software para Arquiteturas de Software Convencionais 291
13.1.4 Uma Estratégia de Teste de Software para Arquiteturas Orientadas a Objetos 292
13.1.5 Critérios para Completamento do Teste 293
13.2 Tópicos Estratégicos 293
13.3 Estratégias de Teste para Software Convencional 294
13.3.1 Teste de Unidade 295
13.3.2 Teste de Integração 297
13.4 Estratégias de Teste para Software Orientado a Objetos 302
13.4.1 Teste de Unidade no Contexto OO 302
13.4.2 Teste de Integração no Contexto OO 303
13.5 Teste de Validação 303
13.5.1 Critérios do Teste de Validação 304
13.5.2 Revisão da Configuração 304
13.5.3 Testes Alfa e Beta 304
13.6 Teste de Sistema 305
13.6.1 Teste de Recuperação 306
13.6.2 Teste de Segurança 306
13.6.3 Teste de Estresse 306
13.6.4 Teste de Desempenho 307
13.7 A Arte da Depuração 308
13.7.1 O Processo de Depuração 308
13.7.2 Considerações Psicológicas 309
13.7.3 Abordagens de Depuração 310
13.7.4 Correção do Erro 311
13.8 Resumo 312
REFERÊNCIAS BIBLIOGRÁFICAS 312
PROBLEMAS E PONTOS A CONSIDERAR 313
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 314

Document shared on www.docsity.com

CAPÍTULO 14 TÉCNICAS DE TESTE DE SOFTWARE 315

14.1 Fundamentos do Teste de Software 316
14.2 Testes Caixa-preta e Caixa-branca 318
14.3 Teste Caixa-branca 318
14.4 Teste de Caminho Básico 319
14.4.1 Notação de Grafo de Fluxo 319
14.4.2 Caminhos Independentes de Programa 320
14.4.3 Derivação de Casos de Teste 322
14.4.4 Matrizes de Grafos 324
14.5 Teste de Estrutura de Controle 324
14.5.1 Teste de Condição 325
14.5.2 Teste de Fluxo de Dados 325
14.5.3 Teste de Ciclo 326
14.6 Teste Caixa-preta 327
14.6.1 Métodos de Teste Baseados em Grafo 327
14.6.2 Particionamento de Equivalência 329
14.6.3 Análise de Valor-limite 329
14.6.4 Teste de Matriz Ortogonal 330
14.7 Métodos de Teste Orientados a Objetos 332
14.7.1 Implicações no Projeto de Casos de Teste dos Conceitos OO 333
14.7.2 Aplicabilidade dos Métodos Convencionais de Projeto de Casos de Teste 333
14.7.3 Teste Baseado em Erro 333
14.7.4 Casos de Teste e Hierarquia de Classes 334
14.7.5 Teste com Base em Cenário 334
14.7.6 Teste da Estrutura Superficial e da Estrutura Profunda 336
14.8 Métodos de Teste Aplicáveis ao Nível de Classe 336
14.8.1 Teste Aleatório para Classes OO 336
14.8.2 Teste de Partição no Nível de Classe 337
14.9 Projeto de Casos de Teste Interclasse 338
14.9.1 Teste de Várias Classes 339
14.9.2 Testes Derivados dos Modelos de Comportamento 339
14.10 Teste de Ambientes, Arquiteturas e Aplicações Especializadas 340
14.10.1 Teste de IGU 341
14.10.2 Teste de Arquiteturas Cliente/Servidor 341
14.10.3 Teste da Documentação e Dispositivos de Ajuda 342
14.10.4 Teste de Sistemas de Tempo Real 342
14.11 Padrões de Teste 344
14.12 Resumo 345
REFERÊNCIAS BIBLIOGRÁFICAS 345
PROBLEMAS E PONTOS A CONSIDERAR 346
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 347

CAPÍTULO 15 MÉTRICAS DE PRODUTO PARA SOFTWARE 348

15.1 Qualidade de Software 349
15.1.1 Fatores de Qualidade de McCall 349

15.1.2 Fatores de Qualidade ISO 9126 351
15.1.3 Transição para Visão Quantitativa 351
15.2 Um Arcabouço para Métricas de Produto 352
15.2.1 Medidas, Métricas e Indicadores 352
15.2.2 O Desafio das Métricas Técnicas 352
15.2.3 Princípios de Medição 353
15.2.4 Medições de Software Orientadas a Objetivo 354
15.2.5 Os Atributos de Métricas de Software Efetivas 355
15.2.6 A Paisagem da Métrica de Produto 355
15.3 Métricas Para o Modelo de Análise 357
15.3.1 Métricas Baseadas em Função 357
15.3.2 Métricas de Qualidade de Especificação 359
15.4 Métricas para o Modelo de Projeto 360
15.4.1 Métricas de Projeto Arquitetural 361
15.4.2 Métricas para o Modelo de Projeto OO 363
15.4.4 Métricas Orientadas à Classe — o Conjunto de Métricas MOOD 366
15.4.5 Métricas Propostas por Lorenz e Kidd 367
15.4.6 Métricas de Projeto em Nível de Componente 367
15.4.7 Métricas Orientadas a Operação 369
15.4.8 Métricas de Projeto de Interface 370
15.5 Métricas de Código-fonte 370
15.6 Métricas para Teste 371
15.6.1 Métricas de Halstead Aplicadas ao Teste 371
15.6.2 Métricas para Teste Orientado a Objetos 372
15.7 Métricas de Manutenção 372
15.8 Resumo 373
REFERÊNCIAS BIBLIOGRÁFICAS 374
PROBLEMAS E PONTOS A CONSIDERAR 375
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 376

PARTE TRÊS — APLICAÇÃO DE ENGENHARIA DA WEB 377

CAPÍTULO 16 ENGENHARIA DA WEB 378

16.1 Atributos de Sistemas e Aplicações Baseados na Web 379
16.2 As Camadas de Engenharia da Webapp 381
16.2.1 Processo 381
16.2.2 Métodos 382
16.2.3 Ferramentas e Tecnologia 382
16.3 O Processo de Engenharia Web 383
16.3.1 Definição do Arcabouço 383
16.3.2 Refinamento do Arcabouço 384
16.4 Melhores Práticas de Engenharia Web 385
16.5 Resumo 387
REFERÊNCIAS BIBLIOGRÁFICAS 387
PROBLEMAS E PONTOS A CONSIDERAR 388
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 388

Document shared on www.docsity.com

CAPÍTULO 17 FORMULAÇÃO E PLANEJAMENTO PARA ENGENHARIA DA WEB 389

17.1 Formulação de Sistemas Baseados na Web 390
- 17.1.1 Questões de Formulação 390
- 17.1.2 Coleta de Requisitos para WebApps 391
- 17.1.3 A Ponte para Modelagem de Análise 395

17.2 Planejamento de Projetos de Engenharia Web 395
17.3 A Equipe de Engenharia da Web 396
- 17.3.1 Os Personagens 396
- 17.3.2 Construção da Equipe 397

17.4. Tópicos de Gestão de Projeto para Engenharia Web 398
- 17.4.1 Planejamento de Terceirização de WebApp 398
- 17.4.2 Planejamento da WebApp — Engenharia da Web Interna 401

17.5 Métricas para Engenharia da Web e Webapps 404
- 17.5.1 Métricas para Esforço da Engenharia da Web 404
- 17.5.2 Métricas para Avaliar Valor de Negócio 405

17.6 "Piores Práticas" de Projetos de Webapp 406
17.7 Resumo 407

REFERÊNCIAS BIBLIOGRÁFICAS 407
PROBLEMAS E PONTOS A CONSIDERAR. 407
LEITURAS E INFORMAÇÕES ADICIONAIS 408

CAPÍTULO 18 MODELAGEM DE ANÁLISE PARA APLICAÇÕES DA WEB 409

18.1 Análise de Requisitos para Webapps 410
- 18.1.1 A Hierarquia de Usuário 410
- 18.1.2 Desenvolvimento de Casos de Uso 411
- 18.1.3 Refinamento do Modelo de Caso de Uso 413

18.2 Modelo de Análise para Webapps 414
18.3 O Modelo de Conteúdo 414
- 18.3.1 Definição dos Objetos de Conteúdo 414
- 18.3.2 Relacionamentos e Hierarquia de Conteúdo 415
- 18.3.3 Classes de Análise[4] de WebApps 416

18.4 O Modelo de Interação 417
18.5 O Modelo Funcional 419
18.6 O Modelo de Configuração 420
18.7 Análise de Relacionamento-Navegação 421
- 18.7.1 Análise de Relacionamentos — Questões-chave 422
- 18.7.2 Análise de Navegação 422

18.8 Resumo 423

REFERÊNCIAS BIBLIOGRÁFICAS 424
PROBLEMAS E PONTOS A CONSIDERAR 424
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 425

CAPÍTULO 19 MODELAGEM DE PROJETO PARA APLICAÇÕES WEB 426

19.1 Tópicos de Projeto para Engenharia da Web 427
- 19.1.1 Projeto e Qualidade da WebApp 427
- 19.1.2 Metas de Projeto 429

19.2 A Pirâmide de Projeto da WebE 430
19.3 Projeto de Interface de Webapp 431
- 19.3.1 Princípios e Diretrizes de Projeto de Interface 432
- 19.3.2 Mecanismos de Controle de Interface 435
- 19.3.3 Projeto de Fluxo de Trabalho de Interface 436

19.4 Projeto Estético 437
- 19.4.1 Tópicos de Leiaute 438
- 19.4.2 Tópicos de Projeto Gráfico 438

19.5 Projeto de Conteúdo 439
- 19.5.1 Objetos de Conteúdo 439
- 19.5.2 Tópicos de Projeto de Conteúdo 440

19.6 Projeto de Arquitetura 440
- 19.6.1 Arquitetura de Conteúdo 441
- 19.6.2 Arquitetura da WebApp 443

19.7 Projeto de Navegação 444
- 19.7.1 Semântica Navegacional 444
- 19.7.2 Sintaxe Navegacional 445

19.8 Projeto No Nível de Componente 446
19.9 Padrões de Projeto de Hipermídia 446
19.10 Método de Projeto de Hipermídia Orientado a Objetos (Object- Oriented Hypermedia Design Method — Oohdm) 447
- 19.10.1 Projeto Conceitual para OOHDM 448
- 19.10.2 Projeto Navegacional de OOHDM 448
- 19.10.3 Projeto e Implementação de Interface Abstrata 449

19.11 Métricas de Projeto de Webapp 450
19.12 Resumo 451

REFERÊNCIAS BIBLIOGRÁFICAS 451
PROBLEMAS E PONTOS A CONSIDERAR 453
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 453

CAPÍTULO 20 TESTE DE APLICAÇÕES WEB 455

20.1 Conceitos de Teste de Webapp 456
- 20.1.1 Dimensões de Qualidade 456
- 20.1.2 Erros em um Ambiente de WebApp 457
- 20.1.3 Estratégia de Teste 457
- 20.1.4 Planejamento de Teste 458

20.2 O Processo de Teste — Um Resumo 458
20.3 Teste de Conteúdo 461
- 20.3.1 Objetivos do Teste de Conteúdo 461
- 20.3.2 Teste de Banco de Dados 462

20.4 Teste de Interface com o Usuário 463
- 20.4.1 Estratégia de Teste de Interface 464
- 20.4.2 Mecanismos de Teste de Interface 464

Document shared on www.docsity.com

20.4.3 Teste de Semântica de Interface 466
20.4.4 Testes de Usabilidade 466
20.4.5 Testes de Compatibilidade 467
20.5 Teste no Nível de Componente 468
20.6 Teste de Navegação 470
20.6.1 Sintaxe do Teste de Navegação 470
20.6.2 Teste de Semântica de Navegação 471
20.7 Teste de Configuração 472
20.7.1 Tópicos do Lado do Servidor 472
20.7.2 Tópicos do Lado do Cliente 472
20.8 Teste de Segurança 473
20.9 Teste de Desempenho 474
20.9.1 Objetivos do Teste de Desempenho 474
20.9.2 Teste de Carga 475
20.9.3 Teste de Esforço 476
20.10 Resumo 477
REFERÊNCIAS BIBLIOGRÁFICAS 478
PROBLEMAS E PONTOS A CONSIDERAR 478
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 479

PARTE 4 — GESTÃO DE PROJETOS DE SOFTWARE 481

CAPÍTULO 21 CONCEITOS DE GESTÃO DE PROJETOS 482
21.1 O Espectro de Gestão 483
21.1.1 O Pessoal 483
21.1.2 O Produto 483
21.1.3 O Processo 484
21.1.4 O Projeto 484
21.2 Pessoal 484
21.2.1 Os Interessados 485
21.2.2 Líderes de Equipe 485
21.2.3 A Equipe de Software 486
21.2.4 Equipes Ágeis 488
21.2.5 Problemas de Coordenação e Comunicação 489
21.3 O Produto 490
21.3.1 Escopo do Software 490
21.3.2 Decomposição do Problema 491
21.4 O Processo 491
21.4.1 Fusão do Produto e do Processo 491
21.4.2 Decomposição do Processo 492
21.5 O Projeto 493
21.6 O Princípio W^{5}hh 494
21.7 Práticas Críticas 495
21.8 Resumo 496

REFERÊNCIAS BIBLIOGRÁFICAS 496
PROBLEMAS E PONTOS A CONSIDERAR 497
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 497

CAPÍTULO 22 MÉTRICAS DE PROCESSO E PROJETO 499
22.1 Métricas nos Domínios do Processo e do Projeto 500
22.1.1 Métricas de Processo e Aperfeiçoamento do Processo de Software 500
22.1.2 Métricas de Projeto 502
22.2 Medição de Software 503
22.2.1 Métricas Orientadas a Tamanho 503
22.2.2 Métricas Orientadas a Função 504
22.2.3 Reconciliação de Métricas LOC e FP 505
22.2.4 Métricas Orientadas a Objetos 506
22.2.5 Métricas Orientadas a Casos de Uso 507
22.2.6 Métricas de Projeto de Engenharia da Web 507
22.3 Métricas de Qualidade de Software 509
22.3.1 Medição de Qualidade 509
22.3.2 Eficiência na Remoção de Defeitos 510
22.4 Integração de Métricas no Processo de Software 511
22.4.1 Argumentos Favoráveis a Métricas de Software 512
22.4.2 Estabelecimento de uma Referência 512
22.4.3 Coleta, Cálculo e Avaliação de Métricas 513
22.5 Métricas para Pequenas Organizações 513
22.6 Estabelecimento de Um Programa de Métricas de Software 514
22.7 Resumo 516
REFERÊNCIAS BIBLIOGRÁFICAS 516
PROBLEMAS E PONTOS A CONSIDERAR 517
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 518

CAPÍTULO 23 ESTIMATIVA DE PROJETOS DE SOFTWARE 519
23.1 Observações sobre Estimativa 520
23.2 O Processo de Planejamento de Projeto 521
23.3 Escopo e Viabilidade do Software 521
23.4 Recursos 522
23.4.1 Recursos Humanos 522
23.4.2 Recursos de Software Reusáveis 523
24.4.3 Recursos de Ambiente 523
23.5 Estimativa do Projeto de Software 524
23.6 Técnicas de Decomposição 525
23.6.1 Dimensionamento do Software 525
23.6.2 Estimativa Baseada no Problema 526
23.6.3 Um Exemplo de Estimativa Baseada em LOC 527
23.6.4 Um Exemplo de Estimativa Baseada em FP 528
23.6.5 Estimativa Baseada em Processo 529
23.6.6 Um Exemplo de Estimativa Baseada em Processo 530
23.6.7 Estimativas com Casos de Uso 530

Document shared on www.docsity.com

23.6.8 Um Exemplo de Estimativa Baseada em Casos de Uso 531
23.6.9 Acomodação das Estimativas 532
23.7 Modelos de Estimativa Empíricos 533
23.7.1 Estrutura dos Modelos de Estimativa 534
23.7.2 O Modelo COCOMO II 534
23.7.3 A Equação de Software 535
23.8 Estimativa de Projetos Orientados a Objetos 536
23.9 Técnicas Especializadas de Estimativa 537
23.9.1 Estimativa para Desenvolvimento Ágil 537
23.9.2 Estimativa para Projetos de Engenharia Web 538
23.10 A Decisão de Fazer/Comprar 539
23.10.1 Criação de uma Árvore de Decisão 539
23.10.2 Terceirização 540
23.11 Resumo 541
REFERÊNCIAS BIBLIOGRÁFICAS 542
PROBLEMAS E PONTOS A CONSIDERAR 542
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 543

CAPÍTULO 24 CRONOGRAMAÇÃO DE PROJETO DE SOFTWARE 544

24.1 Conceitos Básicos 545
24.2 Cronogramação de Projeto 546
24.2.1 Princípios Básicos 547
24.2.2 O Relacionamento entre Pessoal e Esforço 548
24.2.3 Distribuição de Esforço 549
24.3 Definição de um Conjunto de Tarefas para o Projeto de Software 550
24.3.1 Um Exemplo de Conjunto de Tarefas 551
24.3.2 Refinamento das Tarefas Principais 551
24.4 Definição de uma Rede de Tarefas 552
24.5 Cronogramação 553
24.5.1 Gráficos de tempo 554
24.5.2 Acompanhamento do Cronograma 555
24.5.3 Acompanhamento do Progresso de um Projeto OO 556
24.6 Análise do Valor Agregado 557
24.7 Resumo 559
REFERÊNCIAS BIBLIOGRÁFICAS 559
PROBLEMAS E PONTOS A CONSIDERAR 559
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 560

CAPÍTULO 25 GESTÃO DE RISCO 562

25.1 Estratégias de Risco Proativas *versus* Reativas 563
25.2 Riscos de Software 563
25.3 Identificação de Riscos 564
25.3.1 Avaliação do Risco Global do Projeto 565
25.3.2 Componentes e Fatores de Risco 566
25.4 Previsão de Risco 566
25.4.1 Desenvolvimento de uma Tabela de Risco 567

25.4.2 Avaliação do Impacto do Risco 569
25.5 Refinamento de Risco 570
25.6 Atenuação, Monitoração e Gestão de Risco 571
25.7 O Plano RMMM 573
25.8 Resumo 574
REFERÊNCIAS BIBLIOGRÁFICAS 574
PROBLEMAS E PONTOS A CONSIDERAR 575
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 576

CAPÍTULO 26 GESTÃO DE QUALIDADE 577

26.1 Conceitos de Qualidade 578
26.1.1 Qualidade 578
26.1.2 Controle de Qualidade 579
26.1.3 Garantia da Qualidade 579
26.1.4 Custo da Qualidade 579
26.2 Garantia da Qualidade de Software 580
26.2.1 Panorama Histórico 581
26.2.2 Atividades de SQA 581
26.3 Revisões de Software 582
26.3.1 Impacto no Custo de Defeitos de Software 583
26.3.2 Amplificação e Remoção de defeitos 583
26.4 Revisões Técnicas Formais 585
26.4.1 A Reunião de Revisão 585
26.4.2 Relatório e Manutenção de Registros das Revisões 586
26.4.3 Diretrizes de Revisão 586
26.4.4 Revisões Guiadas por Amostras 587
26.5 Abordagens Formais para SQA 589
26.6 Garantia Estatística de Qualidade de Software 589
26.6.1 Um Exemplo Genérico 590
26.6.2 Seis Sigma para Engenharia de Software 591
26.7 Confiabilidade de Software 592
26.7.1 Medidas de Confiabilidade e Disponibilidade 592
26.7.2 Segurança de Software 593
26.8 As Normas de Qualidade ISO 9000 594
26.9 O Plano de SQA 595
26.10 Resumo 596
REFERÊNCIAS BIBLIOGRÁFICAS 596
PROBLEMAS E PONTOS A CONSIDERAR 597
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 598

CAPÍTULO 27 GESTÃO DE MODIFICAÇÕES 599

27.1 Gestão de Configuração de Software 600
27.1.1 Um Cenário SCM 600
27.1.2 Elementos de um Sistema de Gestão de Configuração 601
27.1.3 Referenciais 602
27.1.4 Itens de Configuração de Software 603

Document shared on www.docsity.com

27.2 O Repositório SCM 604
27.2.1 O Papel do Repositório 604
27.2.2 Características e Conteúdo Geral 605
27.2.3 Características de SCM 606
27.3 O Processo de SCM 606
27.3.1 Identificação de Objetos na Configuração de Software 607
27.3.2 Controle de Versão 608
27.3.3 Controle de Modificação 609
27.3.4 Auditoria de Configuração 612
27.3.5 Preparação de Relatórios de Estado 612
27.4 Gestão de Configuração para Engenharia da Web 613
27.4.1 Tópicos de Gestão de Configuração para WebApps 613
27.4.2 Objetos de Configuração de WebApp 614
27.4.3 Gestão de Conteúdo 615
27.4.4 Gestão de Modificação 617
27.4.5 Controle de Versão 619
27.4.6 Auditoria e Preparação de Relatório 620
27.5 Resumo 621
REFERÊNCIAS BIBLIOGRÁFICAS 621
PROBLEMAS E PONTOS A CONSIDERAR 622
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 623

PARTE 5 — TÓPICOS AVANÇADOS EM ENGENHARIA DE SOFTWARE 625

CAPÍTULO 28 MÉTODOS FORMAIS 626
28.1 Conceitos Básicos 627
28.1.1 Deficiências de Abordagens Menos Formais 627
28.1.2 Matemática no Desenvolvimento de Software 628
28.1.3 Conceitos de Métodos Formais 628
28.2 Preliminares Matemáticas 631
28.2.1 Conjuntos e Especificação Construtiva 631
28.2.2 Operadores de Conjuntos 632
28.2.3 Operadores Lógicos 634
28.2.4 Seqüências 634
28.3 Aplicação de Notação Matemática para Especificação Formal 635
28.4 Linguagens de Especificação Formal 636
28.5 Linguagem de Restrição de Objeto (Object Constraint Language, OCL) 637
28.5.1 Um Breve Panorama da Sintaxe e Semântica de OCL 637
28.5.2 Um Exemplo Usando OCL 639
28.6 A Linguagem de Especificação Z 640
28.6.1 Um Breve Panorama da Sintaxe e Semântica de Z 640
28.6.2 Um Exemplo Usando Z 641
28.7 Os Dez Mandamentos dos Métodos Formais 643
28.8 Métodos Formais — a Estrada à Frente 644

28.9 Resumo 644
REFERÊNCIAS BIBLIOGRÁFICAS 645
PROBLEMAS E PONTOS A CONSIDERAR 645
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 646

CAPÍTULO 29 ENGENHARIA DE SOFTWARE SALA LIMPA 647
29.1 A Abordagem Sala Limpa 648
29.1.1 A Estratégia Sala Limpa 648
29.1.2 O Que Torna Sala Limpa Diferente? 650
29.2 Especificação Funcional 651
29.2.1 Especificação Caixa-preta 652
29.2.2 Especificação Caixa de Estado 652
29.2.3 Especificação Caixa-clara 652
29.3 Projeto Sala Limpa 653
29.3.1 Refinamento e Verificação de Projeto 653
29.3.2 Vantagens da Verificação de Projeto[4] 656
29.4 Teste Sala Limpa 658
29.4.1 Teste Estatístico de Uso 658
29.4.2 Certificação 659
29.5 Resumo 659
REFERÊNCIAS BIBLIOGRÁFICAS 660
PROBLEMAS E PONTOS A CONSIDERAR 661
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 661

CAPÍTULO 30 ENGENHARIA DE SOFTWARE BASEADA EM COMPONENTES 663
30.1 Engenharia de Sistemas Baseada em Componentes 664
30.2 O Processo CBSE 665
30.3 Engenharia de Domínio 666
30.3.1 O Processo de Análise de Domínio 666
30.3.2 Funções de Caracterização 667
30.3.3 Modelagem Estrutural e Pontos da Estrutura 668
30.4 Desenvolvimento Baseado em Componentes 668
30.4.1 Qualificação, Adaptação e Composição de Componentes 669
30.4.2 Engenharia de Componentes 671
30.4.3 Análise e Projeto para Reuso 672
30.5 Classificação e Recuperação de Componentes 673
30.5.1 Descrição de Componentes Reusáveis 673
30.5.2 O Ambiente de Reuso 674
30.6 Questões Econômicas Relacionadas à CBSE 675
30.6.1 Impacto na Qualidade, Produtividade e Custo 675
30.6.2 Análise de Custos Usando Pontos Estruturais 676
30.7 Resumo 677
REFERÊNCIAS BIBLIOGRÁFICAS 677
PROBLEMAS E PONTOS A CONSIDERAR 678
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 679

Document shared on www.docsity.com

CAPÍTULO 31 REENGENHARIA 681

31.1 Reengenharia de Processo do Negócio 682
31.1.1 Processos de Negócio 682
31.1.2 Um Modelo BPR 683
31.2 Reengenharia de Software 684
31.2.1 Manutenção de Software 684
31.2.2 Um Modelo de Processo de Reengenharia de Software 685
31.3 Engenharia Reversa 688
31.3.1 Engenharia Reversa para Entender Dados 689
31.3.2 Engenharia Reversa para Entender o Processamento 690
31.3.3 Engenharia Reversa das Interfaces com o Usuário 690
31.4 Reestruturação 691
31.4.1 Reestruturação de Código 692
31.4.2 Reestruturação dos Dados 692
31.5 Engenharia Avante 693
31.5.1 Engenharia Avante para Arquiteturas Cliente/Servidor 694
31.5.2 Engenharia Avante para Arquiteturas Orientadas a Objetos 694
31.5.3 Engenharia Avante de Interfaces com o Usuário 695
31.6 A Economia da Reengenharia 696
31.7 Resumo 696
REFERÊNCIAS BIBLIOGRÁFICAS 697
PROBLEMAS E PONTOS A CONSIDERAR 698
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 698

CAPÍTULO 32 A ESTRADA ADIANTE 700

32.1 A Importância do Software — Revisitada 701
32.2 O Alcance da Modificação 701
32.3 Pessoas e a Maneira como Elas Constroem Sistemas 702
32.4 O "Novo" Processo de Engenharia de Software 703
32.5 Novos Modos de Representar Informação 704
32.6 Tecnologia como Condutora 706
32.7 As Responsabilidades do Engenheiro de Software 707
32.8 Conclusão 708
REFERÊNCIAS BIBLIOGRÁFICAS 709
PROBLEMAS E PONTOS A CONSIDERAR 709
LEITURAS E FONTES DE INFORMAÇÃO ADICIONAIS 709

ÍNDICE ANALÍTICO 711

PREFÁCIO

Quando um software de computador é bem-sucedido — quando satisfaz às necessidades das pessoas que o usam, tem desempenho sem falhas por um longo período, é fácil de modificar e ainda mais fácil de usar — ele pode e efetivamente modifica as coisas para melhor. Mas, quando o software falha — quando seus usuários ficam insatisfeitos, quando tem tendência a erros, quando é difícil de modificar e ainda mais difícil de usar — podem e efetivamente acontecem coisas desagradáveis. Todos nós desejamos construir softwares que tornem as coisas melhores evitando os problemas que espreitam na sombra dos esforços malsucedidos. Para obter sucesso, precisamos de disciplina quando o software é projetado e construído. Precisamos de uma abordagem de engenharia.

Nos 25 anos que se passaram desde que a primeira edição deste livro foi escrita, a engenharia de software evoluiu de uma idéia obscura praticada por um número relativamente pequeno de fanáticos para uma legítima disciplina de engenharia. Hoje, ela é reconhecida como um assunto que merece pesquisa séria, estudo consciencioso e debate acalorado. Em toda a indústria, engenheiro de software substitui programador como cargo preferido. Modelos de processos de software, métodos de engenharia de software e instrumentos de software têm sido adotados com sucesso em um amplo espectro de aplicações na indústria.

Apesar de gerentes e profissionais reconhecerem a necessidade de uma abordagem mais disciplinada para o software, eles continuam a debater a forma pela qual essa disciplina deve ser aplicada. Muitos indivíduos e empresas ainda desenvolvem software ao acaso, mesmo quando constroem sistemas para servir às tecnologias mais avançadas da atualidade. Muitos profissionais e estudantes desconhecem os métodos modernos. Em decorrência disso, a qualidade do software que produzimos é sofrível e coisas ruins acontecem. Além disso, continua o debate e a controvérsia sobre a verdadeira natureza da abordagem de engenharia de software. O estado atual da engenharia de software é um estudo de contrastes. As atitudes mudaram, houve progresso, mas muito resta a ser feito antes que a disciplina alcance maturidade total.

A sexta edição de *Engenharia de Software* tem o objetivo de servir como guia para uma disciplina de engenharia em maturação. A sexta edição, como as cinco edições que a precederam, destina-se a estudantes e profissionais, mantendo sua característica de guia para o profissional da indústria e introdução abrangente para o estudante de graduação ou pós-graduação.

A sexta edição é consideravelmente mais do que uma simples atualização. O livro foi revisto extensivamente e reestruturado para enfatizar novos e importantes processos e práticas de engenharia de software. Além disso, um novo "sistema de apoio", ilustrado na próxima página, fornece um conjunto abrangente de recursos para estudantes, instrutores e profissionais para complementar o conteúdo do livro. Esses recursos são apresentados como parte do site www.mhhe.com/pressman, especificamente projetado para *Engenharia de Software*.

A Sexta Edição. Os 32 capítulos da sexta edição foram organizados em cinco partes. Isso foi feito para compartimentalizar os tópicos e apoiar os instrutores que podem não ter tempo para completar o livro inteiro em um período. A Parte 1, *O Processo de Software*, apresenta diferentes visões do processo de software, considerando todos os modelos importantes de processo e tratando do debate entre as filosofias de processo prescritivas e ágeis. A Parte 2, *Prática de Engenharia de Software*, apresenta métodos de análise, projeto e teste com ênfase em técnicas orientadas a objetos e modelagem UML. Como métodos orientados a objetos são agora amplamente usados na indústria, o conteúdo da Parte 4 da quinta edição ("Engenharia de software orientada a objetos") foi agora totalmente integrado em todas as discussões da prática de engenharia de software nesta edição. A Parte 3, *Aplicação de Engenharia da Web*, apresenta uma abordagem completa de engenharia para análise, projeto e teste de aplicações da Web. A Parte 4, *Gestão de Projetos de Software*, apresenta

Document shared on www.docsity.com

tópicos que são relevantes para aqueles que planejam, gerenciam e controlam um projeto de software. A Parte 5, *Tópicos Avançados de Engenharia de Software*, apresenta capítulos dedicados que tratam de métodos formais, engenharia de software sala limpa, engenharia de software baseada em componentes, reengenharia e tendências futuras.

Além de muitos capítulos novos e significativamente revisados, a sexta edição introduz mais de 120 quadros que (1) permitem ao leitor acompanhar uma equipe de projeto (fictícia) à medida que ela planeja e desenvolve a engenharia de um sistema baseado em computador; (2) fornecem discussões complementares de tópicos selecionados; (3) delineam "conjuntos de tarefas" que descrevem o fluxo de trabalho de atividades selecionadas de engenharia de software; e (4) sugerem ferramentas automatizadas relevantes para os tópicos do capítulo.

A organização em cinco partes permite a um instrutor "aglutinar" tópicos com base no tempo disponível e na necessidade dos alunos. Um curso completo em um período pode ser construído em volta de uma ou mais dessas cinco partes. Por exemplo, um "curso de métodos" poderia enfatizar apenas as Partes 1 e 2; um curso de desenvolvimento baseado na Web poderia enfatizar as partes 1 e 3; um "curso de gestão" enfatizaria as Partes 1 e 4. Ao organizar a sexta edição deste modo, tentei dar aos instrutores um certo número de opções de ensino.

Agradecimentos. Meu trabalho na seis edições de *Engenharia de Software* foi o mais longo projeto técnico continuado de minha vida. Mesmo quando a escrita termina, informação extraída da literatura técnica continuar a ser assimilada e organizada. Por essa razão, agradeço imensamente aos muitos autores de livros, trabalhos e artigos (tanto em papel quanto em meio eletrônico) que me forneceram conhecimento profundo, idéias e comentários adicionais durante os últimos 25 anos.

Agradecimento especial vai para Tim Lethbridge da University of Ottawa que realizou uma revisão extremamente detalhada da sexta edição, ajudou-me no desenvolvimento de exemplos em UML e OCL e desenvolveu o abrangente estudo de caso que acompanha este livro. Sua assistência e comentários foram extremamente valiosos. Especial agradecimento, vai também, para Bruce Maxim da University of Michigan-Dearborn que me ajudou no desenvolvimento do site que acompanha este livro. Bruce é responsável por muito do conteúdo pedagógico. Finalmente, desejo agradecer aos revisores da sexta edição. Seus profundos comentários e críticas inteligentes foram de valor incalculável:

Mark Ardis
Rose-Hulman Institute
Xiaoxia Cao
Shanghai University
Nimmagadda Chalamaiah
Jawaharlal Nehru Technological University
Sergiu Dascalu
University of Nevada, Reno
Harry Delugach
University of Alabama, Huntsville
Premkumar Devanbu
University of California, Davis
Lipika Dey
I.I.T., Delhi
Osama Eljabiri
New Jersey Institute of Technology
Gerald Gannon
Arizona State University
David Gustafson
Kansas State University
Qingchun Hu
East China University of Science and Technology
Shi-Ming Huang
National Chung Cheng University
Clinton Jeffery
New Mexico State University
Barbara Jennings
Colorado School of Mines
Venkatesh Kamat
Goa University
Jo Ann Lane
San Diego State University
Minglu Li
Shanghai Jiao Tong University
Robert Lingard
California State University, Northridge
Jiang B. Liu
Bradley University
WY Liu
City University of Hongkong
Banshidhar Majhi
National Institute of Technology
Aditya P Mathur
Purdue University

John D. McGregor
Clemson University
Hong Mei
Peking University
Ahmed Naumaan
University of Minnesota
Joey Paquet
Concordia University
Deepak Phatak
Indian Institute of Technology Bombay
James Purtilo
University of Maryland
Tong Seng, Jon Quah
Nanyang Technological University
K.V. S.V.N. Raju
Andhra University
D. Janaki Ram
Indian Institute of Technology, Madras
J. L. Rana
Maulana Azad National Institute of Technology (MANIT)
Ahmed Salem
California State University, Salem
Hee Beng Kuan Tan
Nanyang Technological University
Chris Teng
San Jose State University
Flora Tsai
Nanyang Technological University
David Umphress
Auburn University
Muthanna Gowramma Venkateshmurthy
Visveswaraiah Technological University
Liang Wang
Renmin University of China
Laura Williams
North Carolina State University
Junmin Ye
Central China Normal University
Renkun Ying
Tsinghua University

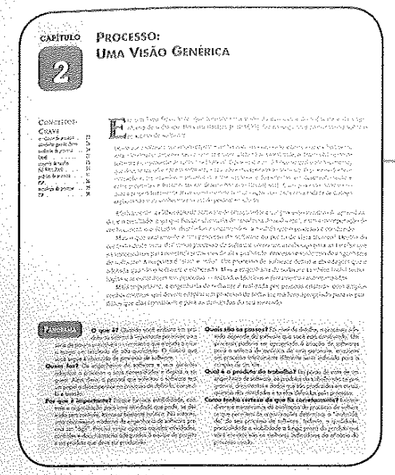
O conteúdo da sexta edição de *Engenharia de Software* foi moldado por profissionais, professores e estudantes universitários que usaram edições anteriores do livro e gastaram tempo para dar suas sugestões, críticas e idéias. Agradeço com alegria a cada um de vocês. Além disso, meus agradecimentos pessoais vão para nossos muitos clientes em todo o mundo, que certamente me ensinaram tanto ou mais quanto eu jamais pude ensinar-lhes.

À medida que as edições deste livro evoluíram, meus filhos, Mathew and Michael, cresceram de meninos para homens. Sua maturidade, caráter e sucesso no mundo real têm sido uma inspiração para mim. Nada me encheu mais de orgulho. E, finalmente, para Bárbara, meu amor e agradecimento por encorajar ainda outra edição de "o livro".

Roger S. Pressman

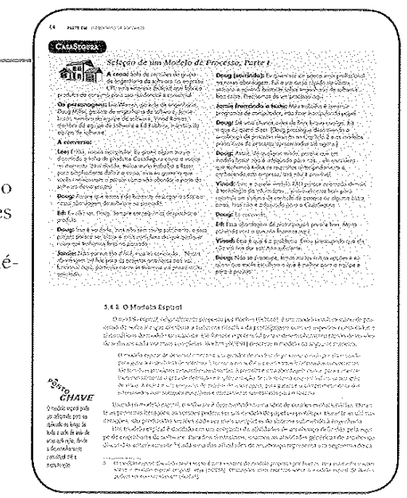
Document shared on www.docsity.com

Recursos deste Livro



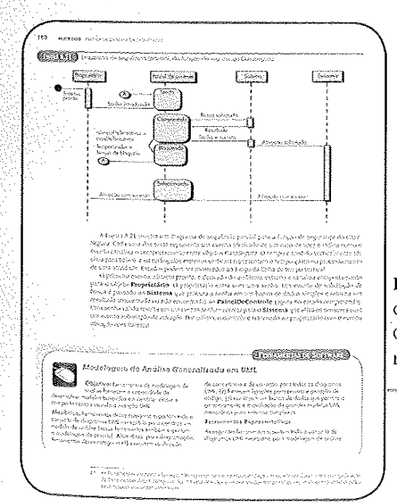
Abertura de Capítulo

Introduz os tópicos que você aprenderá e fornece o conhecimento básico.
Panorama fornece uma visão geral do que o capítulo trata.
Conceitos-chave são indexados de modo que o leitor possa rapidamente chegar ao conceito sobre o qual deseja aprender.



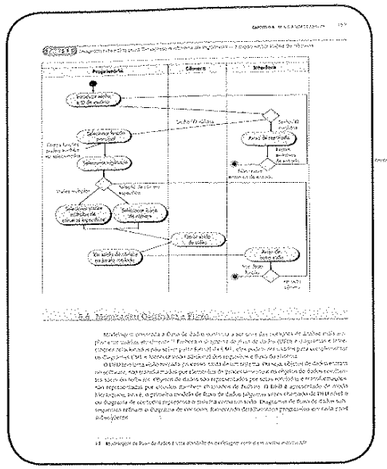
Sabor Prático

Novo Estudo de Caso "CasaSegura" é desenvolvido ao longo do texto. Esses diálogos apresentam situações do mundo real que mostram como membros de uma equipe de software interagem e aplicam princípios e métodos-chave de engenharia de software.



Ferramentas de Software são quadros que ajudam a identifcar quais ferramentas de software são adequadas a tarefas específicas. Os quadros incluem também uma lista de ferramentas representativas.

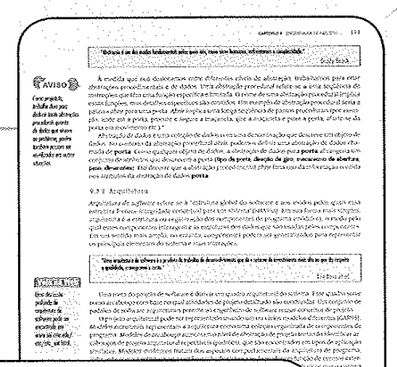
Document shared on www.docsity.com



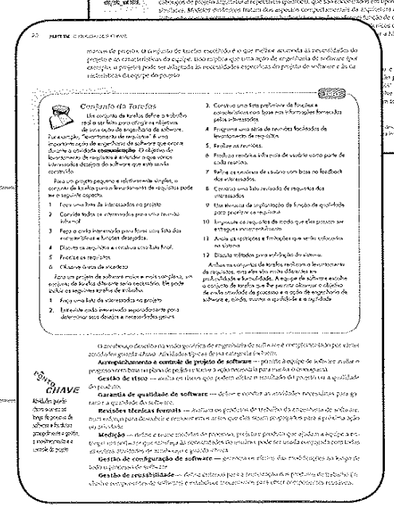
Diagramas UML são usados para ilustrar métodos de análise e projeto importantes tanto para software convencional quanto para aplicações da Web.

Excelente Pedagogia

Citações intercaladas ao longo do livro tornam a leitura divertida e interessante.



Quadros Info apresentam informação que complementa e destaca o tópico que está sendo discutido.



Ícones **Ponto Chave** salientam importantes conceitos a serem lembrados.

Veja na Web

Endereços que vão conduzi-lo a recursos da Internet.

Ícones **Veja na Web** conduzem os leitores para onde mais informação relevante pode ser encontrada na Web.

? Onde posso encontrar a resposta?

Ícones de **Interrogação** formulam questões comuns que são respondidas no corpo do texto.

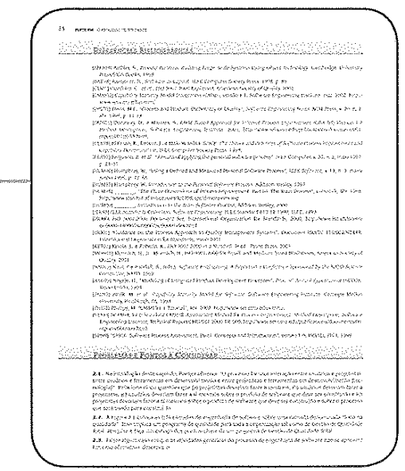
AVISO

Sugestão prática do mundo real da engenharia de software.

Ícones de Aviso fornecem diretrizes pragmáticas que podem ajudá-lo a tomar a decisão correta ou evitar problemas comuns que aparecem durante o desenvolvimento de software.

Material no Final do Capítulo

Um **Resumo** revê brevemente os pontos altos de cada capítulo.
Diversas **Referências** para literatura significativa lhe dá a vantagem de encontrar rapidamente mais informação.
Problemas e Pontos a Considerar são seções de problemas que reforçam importantes conceitos de engenharia de software.
Leituras Adicionais fornecem endereços on-line para mais informação em profundidade.



Document shared on www.docsity.com

Software e Engenharia de Software

CAPÍTULO 1

Conceitos-Chave

características do software . . . 4
categorias de aplicação 6
curva de falhas 4
definição de software 4
desafios 8
deterioração 5
evolução 9
histórico 2
software legado 8
mitos 10

Você já notou que a invenção de uma tecnologia pode ter efeitos profundos e inesperados em outras tecnologias aparentemente não relacionadas, em empresas comerciais, nas pessoas e até na cultura como um todo? Esse fenômeno é freqüentemente chamado de "lei das conseqüências não-pretendidas".

Hoje, o software de computadores é a tecnologia única mais importante no palco mundial. É também um importante exemplo da lei das conseqüências não-pretendidas. Ninguém na década de 1950 poderia ter previsto que o software fosse se tornar uma tecnologia indispensável para negócios, ciência e engenharia; que o software fosse permitir a criação de novas tecnologias (por exemplo, engenharia genética), a extensão de tecnologias existentes (por exemplo, telecomunicações), e o declínio de antigas tecnologias (por exemplo, a indústria tipográfica); que o software se tornaria a força motriz por trás da revolução do computador pessoal; que produtos de software em pequenas embalagens seriam comprados pelos consumidores em centros comerciais da vizinhança; que uma empresa de software fosse se tornar maior e mais influente que a maioria das empresas da era industrial; que uma vasta rede guiada por softwares, chamada Internet, evoluiria e modificaria tudo, desde a pesquisa em bibliotecas até a maneira de os consumidores comprarem e, até mesmo, os hábitos de marcar encontro dos adultos jovens (e não tão jovens).

Ninguém poderia ter previsto que o software estaria embutido em sistemas de toda a espécie: transporte, médico, telecomunicações, militar, industrial, entretenimento, máquinas de escritório — a lista é quase sem fim. E, se formos acreditar na lei das conseqüências não-pretendidas, existem muitos efeitos que nós ainda não podemos prever.

Finalmente, ninguém poderia ter previsto que milhões de programas de computador tivessem de ser corrigidos, adaptados e aperfeiçoados à medida que o tempo passasse, e que o ônus de realizar essas atividades de "manutenção" absorveria mais pessoas e mais recursos que todo o trabalho aplicado na criação de novos softwares.

Panorama

O que é? Software de computador é o produto que os profissionais de software constroem e, depois, mantêm ao longo do tempo. Abrange programas que executam em computadores de qualquer tamanho e arquitetura, conteúdo que é apresentado ao programa a ser executado e documentos tanto em forma impressa quanto virtual que combinam todas as formas de mídia eletrônica.

Quem faz? Engenheiros de software constroem e mantêm, e praticamente todas as pessoas do mundo industrializado usam direta ou indiretamente.

Por que é importante? Porque afeta praticamente todos os aspectos de nossas vidas e tornou-se difundido no nosso comércio, na nossa cultura e nas nossas atividades do dia-a-dia.

Quais são os passos? Você constrói software de computadores como constrói qualquer produto bem-sucedido, aplicando um processo ágil e adaptável que leva a um resultado de alta qualidade e que satisfaz às necessidades das pessoas que vão usar o produto. Você aplica uma abordagem de engenharia de software.

Qual é o produto do trabalho? Do ponto de vista do engenheiro de software, o produto do trabalho são os programas, conteúdo (dados) e documentos que compõem um software de computador. Mas, do ponto de vista do usuário, o produto do trabalho é a informação resultante que, de algum modo, torna melhor o mundo do usuário.

Como tenho certeza de que fiz corretamente? Leia o restante deste livro, selecione as idéias que são aplicáveis ao software que você constrói e aplique-as ao seu trabalho.

Document shared on www.docsity.com

> "Idéias e descobertas tecnológicas são as forças propulsoras do crescimento econômico."
>
> **The Wall Street Journal**

À medida que a importância do software cresceu, a comunidade de software tem continuamente tentado desenvolver tecnologias que tornem mais fácil, mais rápido e menos dispendioso construir e manter programas de computador de alta qualidade. Algumas dessas tecnologias são voltadas para um domínio de aplicação específico (por exemplo, projeto e implementação de sites da Web); outros enfocam um domínio tecnológico (por exemplo, sistemas orientados a objetos ou programação orientada a aspectos) e outros, ainda, são de base mais ampla (por exemplo, sistemas operacionais tais como o LINUX). No entanto, ainda temos de desenvolver uma tecnologia de software que faça isso tudo, e a probabilidade de surgir uma dessas no futuro é pequena. Mesmo assim, as pessoas apostam seus empregos, sua segurança e suas próprias vidas em softwares de computador. É melhor que esteja correto.

Este livro apresenta um arcabouço que pode ser usado por aqueles que constroem software de computadores — pessoas que precisam acertar. O arcabouço abrange um processo, um conjunto de métodos e ferramentas, que nós chamamos de *engenharia de software*.

> "Na sociedade moderna, o papel da engenharia é fornecer sistemas e produtos que melhoram os aspectos materiais da vida humana, tornando assim a vida mais fácil, menos perigosa, mais segura e mais agradável."
>
> **Richard Fairley e Mary Willshire**

1.1 O Papel Evolutivo do Software

Ponto Chave

O software é tanto o produto quanto o veículo para entrega do produto.

Hoje em dia o software assume um duplo papel. Ele é o produto e, ao mesmo tempo, o veículo para entrega do produto. Como produto ele disponibiliza o potencial de computação presente no hardware do computador ou, mais amplamente, por uma rede de computadores acessível pelo hardware local. Quer resida em um telefone celular, quer opere em um computador de grande porte, o software é um transformador de informações — produzindo, gerindo, adquirindo, modificando, exibindo ou transmitindo informações que podem ser tão simples como um único bit ou tão complexas quanto uma apresentação multimídia. Como veículo usado para entrega do produto, o software age como base para o controle do computador (sistemas operacionais), para a comunicação da informação (redes) e para a criação e o controle de outros programas (ferramentas e ambientes de software).

O software entrega o mais importante produto da nossa época — a informação. Ele transforma dados pessoais (por exemplo, as transações financeiras de uma pessoa) de modo que os dados possam ser mais úteis em um determinado contexto; organiza informações comerciais para melhorar a competitividade; fornece um portal para as redes de informação de âmbito mundial (por exemplo, a Internet) e proporciona os meios para obter informação em todas as suas formas.

Veja na Web

Dê uma olhada no passado da indústria de software em www.softwarehistory.org.

A importância do software de computadores tem passado por mudanças significativas em pouco mais de 50 anos. Melhora surpreendente no desempenho do hardware, profundas modificações na arquitetura de computadores, aumento significativo na memória e na capacidade de armazenamento, e uma variedade de opções incomuns de entrada e saída levaram a sistemas baseados em computador mais sofisticados e complexos. Sofisticação e complexidade podem produzir resultados magníficos quando um sistema é bem-sucedido, mas também podem causar enormes problemas para quem precisa construir sistemas complexos.

Livros populares publicados nas décadas de 1970 e 1980 fornecem uma visão histórica útil da percepção de mudanças dos computadores e do software, e do seu impacto em nossa cultura. Osborne [OSB79] caracterizou como uma "nova revolução industrial". Toffler [TOF80] chamou o advento da microeletrônica de "terceira onda de mudança" da história humana e Naisbitt [NAI82] previu a transformação de uma sociedade industrial para uma "sociedade da informação". Feigenbaum e McCorduck [FEI83] sugeriram que a informação e o conhecimento (controlados por computadores) seriam o centro do poder no século XXI, e Stoll [STO89] ponderou que a "comunidade eletrônica", criada por redes e software, era a chave para a troca de conhecimento em todo o mundo. Todos eles estavam certos.

Aviso

Se você tiver tempo, dê uma olhada em um ou mais desses livros clássicos. Preste atenção no que esses especialistas erraram quando previram eventos e tecnologias futuros. Seja humilde: nenhum de nós pode realmente saber o futuro dos sistemas que construímos.

No início dos anos 90, Toffler [TOF90] descreveu uma "mudança de poder" na qual antigas estruturas de poder (governamental, educacional, industrial, econômica e militar) se desintegravam, visto que computadores e software levam a uma "democratização do conhecimento". Yourdon [YOU92] temeu que as empresas norte-americanas pudessem perder sua vantagem competitiva nos negócios relacionados a software e previu "o declínio e a queda do programador norte-americano". Hammer e Champy [HAM93] previram que as tecnologias da informação desempenhariam um papel fundamental na "reengenharia das empresas". Em meados dos anos 90, a difusão dos computadores e do software provocou uma onda de livros "anti-computadores" (por exemplo, *Resisting the Virtual Life*, editado por James Brook e Iain Boal, e *The Future Does Not Compute* por Stephen Talbot). Esses autores demonificaram o computador, enfatizando preocupações legítimas, mas ignorando os profundos benefícios que deles já haviam sido obtidos [LEV95].

> "Os computadores tornam mais fácil fazer uma série de coisas, mas a maior parte das coisas que eles facilitam não precisa ser feita."
>
> **Andy Rooney**

Na segunda metade da década de 1990, Yourdon [YOU96] reavaliou as perspectivas para o profissional de software e sugeriu "a ascensão e a ressurreição" do programador norte-americano. À medida que a Internet cresceu em importância, a mudança de tendência de Yourdon mostrou ser correta. Ao final do século XX, o foco deslocou-se mais uma vez, agora sob o impacto da "bomba-relógio" do ano 2000 (por exemplo, [YOU98a], [KAR99]). Apesar de as previsões de catástrofe do ano 2000 terem sido exageradas, sua popularização concorreu para a invasão do software em nossas vidas.

Veja na Web

Para obter comentários sobre uma ampla gama de tópicos relacionados a softwares visite www.yourdon.com.

Conforme o segundo milênio progredia, Johnson [JOH01] discutiu o poder do "surgimento" — fenômeno que explica o que acontece quando as interconexões entre entidades relativamente simples resulta em um sistema que "se auto-organiza para formar comportamento mais inteligente e mais adaptativo". Yourdon [YOU02] revisitou os trágicos eventos de 11 de setembro para discutir o impacto contínuo do terrorismo global na comunidade de TI. Wolfram [WOL02] apresentou um tratado sobre "uma nova espécie de ciência", que postula uma teoria unificadora baseada principalmente em simulações de software sofisticadas. Daconta e seus colegas [DAC03] discutiram a evolução da "Web semântica" e os modos pelos quais ela vai mudar a maneira pela qual as pessoas interagem com as redes globais.

> "Quando mergulhei no futuro, tão longe quanto o olho humano pode alcançar, tive a visão do mundo e de toda a maravilha em que poderia se tornar."
>
> **Tennyson**

Hoje em dia uma enorme indústria de software tornou-se fator dominante nas economias do mundo industrializado. O programador solitário de antigamente foi substituído por uma equipe de especialistas em software, cada um se concentrando numa parte da tecnologia necessária para produzir uma aplicação complexa. No entanto, as questões que eram feitas ao programador solitário são as mesmas questões que são feitas quando modernos sistemas baseados em computador são construídos:[1]

- Por que leva tanto tempo para concluir o software?
- Por que os custos de desenvolvimento são tão altos?
- Por que não podemos achar todos os erros antes de entregar o software aos clientes?
- Por que gastamos tanto tempo e esforço mantendo programas existentes?

1 Em um excelente livro de ensaios sobre os negócios de software, Tom DeMarco [DEM95] argumenta o contrário. Afirma ele: "Em vez de perguntar por que o software custa tanto, precisamos começar a perguntar o que fizemos para tornar possível que o software de hoje custe tão pouco? A resposta a essa questão nos ajudará a continuar o extraordinário nível de conquistas que sempre distinguiu a indústria de software".

Document shared on www.docsity.com

- Por que continuamos a ter dificuldade em avaliar o progresso enquanto o software é desenvolvido e mantido?

Essas e muitas outras questões demonstram a preocupação da indústria com o software e a maneira pela qual ele é desenvolvido — preocupação que tem levado à adoção da prática de engenharia de software.

1.2 SOFTWARE

? Como deveríamos definir software?

Em 1970, menos de 1% do público poderia ter definido o que "software de computadores" significava. Hoje, a maioria dos profissionais e muitos membros do público em geral pensam que entendem de software. Mas será verdade?

A definição de software em um livro-texto poderia ter a seguinte forma: *Software são (1) instruções (programas de computadores) que quando executadas fornecem as características, função e desempenho desejados; (2) estruturas de dados que permitem aos programas manipular adequadamente a informação; e (3) documentos que descrevem a operação e o uso dos programas.* Não há dúvida de que definições mais completas poderiam ser oferecidas, mas nós precisamos mais do que uma definição formal.

PONTO CHAVE
Software é elaborado e não manufaturado.

Para entendermos de software (e, em última análise, de engenharia de software), é importante examinar as características do software que o tornam diferente de outras coisas que os seres humanos produzem. O software é um elemento de um sistema lógico e não de um sistema físico. Assim, ele possui características que são consideravelmente diferentes daquelas do hardware:

1. *O software é desenvolvido ou passa por um processo de engenharia; não é fabricado no sentido clássico.*

PONTO CHAVE
Software não se degasta, mas se deteriora.

Apesar de existirem algumas semelhanças entre o desenvolvimento de softwares e a fabricação de hardware, as duas atividades são fundamentalmente diferentes. Em ambas, a alta qualidade é conseguida por um bom projeto, mas a fase de fabricação do hardware pode gerar problemas de qualidade, que são inexistentes (ou facilmente corrigidos) para o software. Ambas as atividades são dependentes de pessoas, mas a relação entre as pessoas envolvidas e o trabalho realizado é inteiramente diferente (veja o Capítulo 24). As duas atividades requerem a construção de um "produto", porém as abordagens são diferentes. Os custos do software são concentrados na engenharia. Isso significa que os projetos de software não podem ser geridos como se fossem projetos de fabricação.

2. *Software não "se desgasta".*

A Figura 1.1 mostra a taxa de falhas como uma função do tempo para o hardware. O relacionamento, freqüentemente chamado de "curva da banheira", indica que o hardware

FIGURA 1.1
Curva de falhas para o hardware

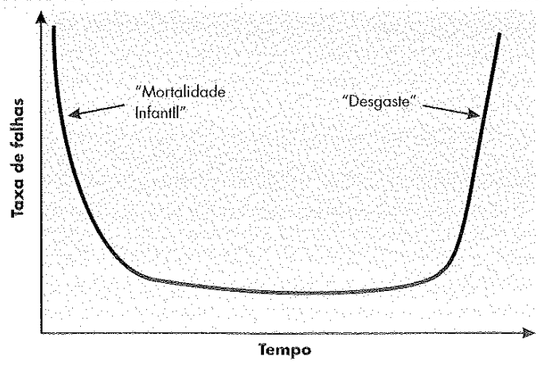
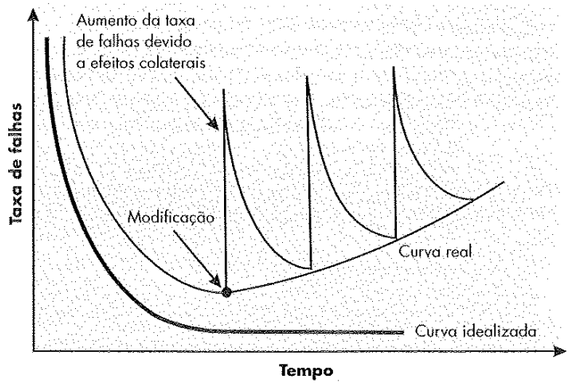


FIGURA 1.2
Curva de falhas para o software



AVISO
Se você deseja reduzir a deteriorização do software, terá de melhorar o projeto do software (Caps. 9–12).

exibe taxas de falha relativamente altas no início de sua vida (essas falhas são freqüentemente atribuíveis a defeitos de projeto ou de fabricação). Os defeitos são depois corrigidos e a taxa de falhas cai para um nível constante (idealmente bastante baixo) por um certo período de tempo. Com o passar do tempo, no entanto, a taxa de falhas aumenta novamente, visto que os componentes de hardware sofrem o efeito cumulativo de poeira, vibração, maus-tratos, gradientes de temperatura e muitos outros males ambientais. Falando simplesmente, o hardware começa a se *desgastar*.

O software não é suscetível aos males ambientais que causam o desgaste do hardware. Teoricamente, entretanto, a curva da taxa de falhas para o software deveria tomar a forma da "curva idealizada" mostrada na Figura 1.2. Defeitos não detectados causarão altas taxas de falhas no início da vida de um programa. Todavia, eles são corrigidos (idealmente sem a introdução de outros erros) e a curva se achata, conforme mostrado. A curva idealizada é uma simplificação grosseira dos modelos reais de falhas (veja o Capítulo 26 para mais informações) para softwares. Todavia, a implicação é clara — o software não se desgasta. Mas se *deteriora*!

PONTO CHAVE
Os métodos de engenharia de software tentam reduzir a amplitude dos dentes e a inclinação da curva real da Figura 1.2.

Essa aparente contradição pode ser melhor explicada considerando a "curva real" mostrada na Figura 1.2. Durante sua vida,[2] o software passará por modificações. Conforme as modificações são feitas, é provável que erros sejam introduzidos causando um dente na curva da taxa de falhas, conforme mostra a Figura 1.2. Antes que a curva possa voltar ao seu estado original de taxa de falhas estável, outra modificação é solicitada causando novo dente na curva. A taxa de falhas mínima começa a subir lentamente — o software está se deteriorando em razão das modificações.

Outro aspecto do desgaste ilustra a diferença entre hardware e software. Quando um componente de hardware se desgasta, é substituído por uma peça sobressalente. Não há peças sobressalentes de software. Toda falha de software indica um erro no projeto, ou no processo pelo qual o projeto foi traduzido para código de máquina executável. Assim, a manutenção de software envolve consideravelmente mais complexidade do que a manutenção de hardware.

PONTO CHAVE
A maioria dos softwares continua a ser construída sob encomenda.

3. *Apesar da indústria estar se movendo em direção à montagem baseada em componentes, a maior parte dos softwares continua a ser construída sob encomenda.*

Considere a maneira pela qual o hardware de controle para um produto baseado em computador é projetado e construído. O engenheiro de projeto desenha um esquema simples do circuito digital, faz uma análise fundamental para assegurar a função pretendida e, depois, consulta os catálogos de componentes digitais. Cada circuito integrado tem um código de componente, uma função definida e validada, uma interface bem delineada e um

2 De fato, a partir do momento que o desenvolvimento começa, e muito antes da primeira versão ser liberada, modificações podem ser requisitadas pelo cliente.

Document shared on www.docsity.com

conjunto padrão de diretrizes de integração. Depois que cada componente é selecionado, ele pode ser requisitado do estoque.

À medida que uma disciplina da engenharia evolui, uma coleção de componentes de projeto padronizados é criada. Parafusos padronizados e circuitos integrados de linha são apenas dois dos milhares de componentes-padrão que são usados por engenheiros mecânicos e eletricistas quando projetam novos sistemas. Os componentes reutilizáveis foram criados para que o engenheiro possa se concentrar nos elementos realmente inovadores de um projeto, isto é, as partes do projeto que representam algo novo no mundo do hardware, o reuso de componentes é natural no processo de engenharia. No mundo do software, é algo que está apenas começando a ser obtido em ampla escala.

> "Idéias são blocos construtivos de idéias."
>
> Jason Zebehazy

Um componente de software deve ser projetado e implementado de modo que possa ser reusado em muitos programas diferentes. Componentes modernos reutilizáveis encapsulam tanto os dados quanto o processamento aplicado aos dados, permitindo ao engenheiro de software criar novas aplicações a partir de partes reusáveis.[3] Por exemplo, as atuais interfaces gráficas com os usuários são construídas com componentes reutilizáveis que permitem a criação de janelas gráficas, menus retráteis (*pull-down*) e uma ampla variedade de mecanismos de interação. Os detalhes de estruturas de dados e de processamento necessários para construir a interface estão contidos em uma biblioteca de componentes reutilizáveis para construção de interface.

1.3 A Natureza Mutável do Software

Hoje em dia, sete amplas categorias de software de computadores apresentam desafios contínuos para os engenheiros de software:

Software de sistemas. Software de sistemas é uma coleção de programas escritos para servir a outros programas. Alguns — por exemplo, compiladores, editores e utilitários para gestão de arquivos — processam estruturas de informação complexas mas determinadas.[4] Outras aplicações de sistemas (por exemplo, componentes de sistemas operacionais, acionadores, software de rede e processadores de telecomunicações) processam dados amplamente indeterminados. Em ambos os casos, a área de software de sistemas é caracterizada por interação intensa com o hardware do computador; uso intenso por múltiplos usuários; operação concorrente que requer ordenação, compartilhamento de recursos e sofisticada gestão de processo; estruturas de dados complexas e interfaces externas múltiplas.

Veja na Web

Uma das mais abrangentes bibliotecas de shareware/freeware pode ser encontrada em shareware.cnet.com.

Software de aplicação. O software de aplicação consiste de programas isolados que resolvem uma necessidade específica do negócio. Aplicações nessa área processam dados comerciais ou técnicos de um modo que facilita as operações ou gestão/tomada de decisões técnicas do negócio. Além das aplicações convencionais de processamento de dados, o software de aplicação é usado para controlar funções do negócio em tempo real (por exemplo, processamento de transações no ponto-de-venda, controle de processo de fabricação em tempo real).

Software científico e de engenharia. Antigamente caracterizado por algoritmos "number crunching" (que processam números), as aplicações de software científico e de engenharia vão da astronomia à vulcanologia, da análise automotiva de tensões à dinâmica orbital do ônibus espacial, e da biologia molecular à manufatura automatizada. Todavia, as aplicações modernas na área científica de engenharia estão se afastando dos algoritmos numéricos convencionais. Projeto apoiado por computadores, simulação de sistemas e outras aplicações interativas começaram a adquirir características de tempo real e até de software de sistemas.

3 A engenharia de software baseada em componentes é apresentada no Capítulo 30.

4 O software é *determinado* se a ordem e a ocasião das entradas, processamento e saídas é previsível. Ele é *indeterminado* se a ordem e a ocasião das entradas, processamento e saídas não puderem ser previstas antecipadamente.

Software embutido. O software embutido reside dentro de um produto ou sistema e é usado para implementar e controlar características e funções para o usuário final e para o próprio sistema. O software embutido pode realizar funções muito limitadas e particulares (por exemplo, o controle de teclado para um forno de microondas) ou fornecer função significativa e capacidade de controle (por exemplo, funções digitais em um automóvel tais como controle de combustível, mostradores do painel e sistemas de frenagem etc.).

Software para linhas de produtos. Projetado para fornecer uma capacidade específica a ser usada por muitos clientes diferentes, o software para linhas de produtos pode focalizar um mercado limitado e especial (por exemplo, produtos de controle de estoque) ou dirigir-se ao mercado de consumo de massa (por exemplo, processamento de texto, planilhas, gráficos por computador, multimídia, entretenimento, gestão de banco de dados, aplicações financeiras pessoais e empresariais).

Aplicações da Web. Aplicações da Web, "ApsWeb", cobrem uma ampla gama de aplicações. Na sua forma mais simples, ApsWeb podem ser pouco mais que um conjunto de arquivos ligados por hipertexto que apresentam informações usando texto e poucos gráficos. No entanto, conforme as aplicações de comércio eletrônico (*e-commerce*) e B2B crescem em importância, as ApsWeb evoluem para sofisticados ambientes computacionais que fornecem não apenas características isoladas, funções de computação e conteúdo para o usuário final, mas também estão integradas ao banco de dados da empresa e às aplicações do negócio.

Software para inteligência artificial. O software para inteligência artificial (AI) faz uso de algoritmos não-numéricos para resolver problemas complexos que não são passíveis de computação ou análise direta. Aplicações nessa área incluem robótica, sistemas especialistas, reconhecimento de padrões (de imagem e de voz), redes neurais artificiais, prova de teoremas e jogos.

> "Não há um computador que tenha bom senso."
>
> Marvin Minsky

Milhões de engenheiros de software em todo o mundo estão trabalhando duro em projetos de uma ou mais dessas categorias. Em alguns casos, novos sistemas estão sendo construídos, mas em outros aplicações existentes são corrigidas, adaptadas e aperfeiçoadas. É comum para um jovem engenheiro de software trabalhar em um programa que é mais velho do que ele! Gerações passadas de pessoal de software deixaram um legado em cada uma dessas categorias que foram discutidas. Tomara que o legado deixado por esta geração torne mais fácil o encargo de futuros engenheiros de software. No entanto, novos desafios apareceram no horizonte:

Computação ubíqua. O rápido crescimento de redes sem fio pode levar logo à verdadeira computação distribuída. O desafio para os engenheiros de software será desenvolver sistemas e softwares de aplicação que permitam que pequenos dipositivos, computadores pessoais e sistemas empresariais comuniquem-se através de grandes redes.

NetSourcing. A World Wide Web (WWW) está rapidamente se transformando em uma ferramenta da computação e também em um fornecedor de conteúdo. O desafio para os engenheiros de software é arquitetar aplicações simples (como planejamento financeiro pessoal) e sofisticadas que forneçam benefícios para um mercado global de usuários finais visados.

> "Você não pode sempre prever, mas pode sempre se preparar."
>
> Anônimo

Software aberto. Uma tendência crescente que resulta na distribuição de código-fonte para sistemas de aplicação (por exemplo, sistemas operacionais, bancos de dados e ambientes de desenvolvimento), de modo que os clientes possam fazer modificações locais. O desafio para os engenheiros de software é construir código-fonte que seja auto-descritível e, mais importante,

Document shared on www.docsity.com

desenvolver técnicas que permitam ao cliente e aos desenvolvedores saber que modificações foram feitas e como essas modificações se manifestam no software.

A "nova economia". A insanidade ponto-com que tomou os mercados financeiros durante o final da década de 1990 e o surto que seguiu-se no início dos anos 2000 levaram muitas pessoas de negócio a acreditar que a nova economia está morta. A nova economia está viva e muito bem, mas vai evoluir lentamente. Ela será caracterizada pela comunicação e distribuição em massa. Andy Lippman [LIP02] nota isso quando escreve:

> Estamos entrando em uma era caracterizada por comunicações entre máquinas distribuídas e pessoas dispersas, em vez de ser principalmente uma conexão entre dois indivíduos ou entre um indivíduo e uma máquina. A abordagem antiga para telefonia era sobre "conexões para"; a onda seguinte é sobre "conexões entre". Os sistemas Napster, de mensagens instântaneas, de mensagens curtas e BlackBerries são exemplos disso.

O desafio para os engenheiros de software é construir aplicações que facilitem a comunicação de massa e a distribuição de produtos em massa usando conceitos que estão agora apenas se formando.

Cada um desses "novos desafios" vai, sem dúvida, obedecer à lei das conseqüências não-pretendidas e terá efeitos (para empresários, engenheiros de software e usuários finais) que não podem ser hoje previstos. No entanto, os engenheiros de software podem se preparar para instanciar um processo que seja suficientemente ágil e adaptável para acomodar grandes modificações em tecnologia e regras de negócio que certamente virão na próxima década.

> "O próprio computador vai fazer uma transição histórica de algo que é usado para tarefas analíticas . . . para algo que pode provocar emoção."
>
> **David Vaskevitch**

1.4 Software Legado

? O que é software legado?

Centenas de milhares de programas de computador caem em um dos sete amplos domínios de aplicação — software de sistemas, software de aplicação, software científico e de engenharia, software embutido, produtos de software, ApsWeb e aplicações para inteligência artificial — discutidos na Seção 1.3. Alguns deles são software do estado da arte — recém-lançados para indivíduos, indústria e governo. Mas outros programas são mais antigos, em alguns casos *muito* mais antigos.

Esses programas antigos — freqüentemente referidos como *software legado* — têm sido foco de atenção e preocupação contínuas desde a década de 1960. Dayani-Fard e seus colegas [DAY99] descrevem o software legado do seguinte modo:

> Sistemas de software legado (...) foram desenvolvidos décadas atrás e têm sido continuamente modificados para satisfazer a mudanças nos requisitos do negócio e nas plataformas de computação. A proliferação de tais sistemas está causando dor de cabeça para grandes organizações que os consideram dispendiosos de manter e arriscados de evoluir.

Liu e seus colegas [LIU98] estendem essa descrição observando que "muitos sistemas legados permanecem dando suporte a funções importantes do negócio e são indispensáveis ao negócio". Assim, software legado é caracterizado por longevidade e criticalidade para o negócio.

1.4.1 A Qualidade do Software Legado

? Que devo fazer se encontrar um sistema legado que exiba má qualidade?

Infelizmente, existe uma característica adicional que pode estar presente no software legado — *má qualidade*.[5] Sistemas legados, algumas vezes, têm projetos não-extensíveis, código complicado, documentação pobre ou inexistente, casos de teste e resultados que nunca foram arquivados, um histórico de modificações mal gerido — a lista pode ser bem longa. E, no entanto, esses

5 Nesse caso, a qualidade é julgada com base no pensamento moderno de engenharia de software — critério um pouco injusto, tendo em vista que alguns conceitos e princípios modernos de engenharia de software podem não ter sido bem entendidos na época em que o software legado foi desenvolvido.

sistemas apóiam "funções importantes do negócio e são indispensáveis ao negócio" [LIU98]. O que pode ser feito?

A única resposta razoável pode ser não fazer nada, pelo menos até que o sistema legado precise passar por modificações significativas. Se o software legado satisfaz às necessidades de seus usuários e funciona confiavelmente, não está danificado e não precisa ser consertado. No entanto, à medida que o tempo passa, os sistemas legados freqüentemente evoluem, por uma ou mais das seguinte razões:

? Que tipos de modificações são feitas em sistemas legados?

- O software precisa ser adaptado para satisfazer às necessidades do novo ambiente ou tecnologia computacional.
- O software precisa ser aperfeiçoado para implementar novos requisitos do negócio.
- O software precisa ser estendido para torná-lo interoperável com os sistemas ou bancos de dados mais modernos.
- O software precisa ser rearquitetado para torná-lo viável em um ambiente de rede.

Quando esses modos de evolução ocorrem, um sistema legado precisa ser "reengenheirado" (Capítulo 31) de modo que permaneça viável no futuro. O objetivo da engenharia de software moderna é "conceber metodologias que sejam fundamentadas na noção de evolução", ou seja, a noção de que "sistemas de software modificam-se continuamente, novos sistemas de software são construídos a partir dos antigos e. . . todos precisam interoperar e cooperar uns com os outros" [DAY99].

AVISO

Todo engenheiro de software precisa reconhecer que modificações são naturais. Não tente combatê-las.

1.4.2 Evolução de Software

Independentemente do tamanho, complexidade ou domínio de aplicação, o software de computador vai evoluir com o tempo. As modificações (freqüentemente denominadas *manutenção de software*) orientam esse processo e ocorrem quando erros são corrigidos, quando o software é adaptado a um novo ambiente, quando o cliente solicita novas características ou funções e quando a aplicação é "reengenheirada" para fornecer benefícios em um contexto moderno. San Williams [WIL02] descreve isso quando escreve:

> À medida que os programas de larga escala como o Windows e o Solaris expandem-se bastante no intervalo de 30 a 50 milhões de linhas de código, gerentes de projetos bem-sucedidos aprenderam a dedicar tanto tempo para desatar os nós de código legado quanto para adicionar novo código. Colocado de modo simples, em uma década que viu o desempenho médio do microchip de PC aumentar cem vezes, a inabilidade do software de aumentar na mesma velocidade foi de um pequeno segredo a um embaraço de toda a indústria.

Durante os últimos 30 anos, Manny Lehman [por exemplo, LEH97a] e seus colegas realizaram análises detalhadas de software e sistemas de qualidade industrial em um esforço para desenvolver uma *teoria unificada para evolução de software*. Os detalhes desse trabalho vão além do escopo deste livro,[6] mas as leis subjacentes que foram derivadas são dignas de nota [LEH97b].

Lei da Modificação Contínua (1974). Sistemas tipo-E[7] devem ser continuamente adaptados ou então eles se tornam progressivamente menos satisfatórios.

Lei da Complexidade Crescente (1974). À medida que um sistema tipo-E evolui sua complexidade aumenta, a menos que seja realizado esforço para mantê-la ou reduzi-la.

Lei da Auto-Regulação (1974). O processo de evolução de sistemas tipo-E é auto-regulatório, com medidas de distribuição do produto e do processo perto do normal.

Lei da Conservação da Estabilidade Organizacional (1980). A velocidade média da atividade global efetiva em um sistema evolutivo tipo-E é invariante ao longo do tempo de vida do produto.

Lei da Conservação da Familiaridade (1980). À medida que um sistema tipo-E evolui, todos os que estão a ele associados, desenvolvedores, pessoal de vendas e usuários, por exemplo,

6 O leitor interessado deve ver [LEH97a] para uma discussão abrangente da evolução do software.

7 *Sistemas tipo-E* são softwares que foram implementados em um sistema de computação do mundo real e, portanto, evoluirão com o tempo.

Document shared on www.docsity.com

devem manter o domínio de seu conteúdo e comportamento para conseguir evolução satisfatória. O crescimento excessivo diminui esse domínio. Assim, o crescimento médio incremental permanece invariante à medida que o sistema evolui.

Lei do Crescimento Contínuo (1980). O conteúdo funcional de sistemas tipo-E deve ser continuamente aumentado para manter a satisfação do usuário ao longo do tempo de vida do sistema.

? Por que os sistemas legados evoluem com o passar do tempo?

Lei da Qualidade Declinante (1996). A qualidade de sistemas tipo-E parecerá estar declinando a menos que eles sejam rigorosamente mantidos e adaptados às modificações no ambiente operacional.

Lei de Realimentação do Sistema (1996). Os processos de evolução tipo-E constituem sistemas de realimentação em multiníveis, em multiciclos, por multiagentes e precisam ser tratados como tal para conseguir aperfeiçoamento significativo sobre qualquer base razoável.

As leis que Lehman e seus colegas definiram são uma parte inerente da realidade de um engenheiro de software. No resto deste livro, discutimos modelos de processo de software, métodos de engenharia de software e técnicas gerenciais que visam manter a qualidade à medida que o software evolui.

1.5 Mitos do Software

Mitos de Software — crenças sobre softwares e sobre o processo usado para construí-los — podem ser encontrados desde os primeiros dias da computação. Os mitos têm uma quantidade de atributos que os têm tornado traiçoeiros. Por exemplo, os mitos parecem ser afirmações de fatos razoáveis (algumas vezes contendo elementos verdadeiros), têm um aspecto intuitivo e são freqüentemente divulgados por profissionais experientes que "entendem do assunto".

> "Na falta de padrões expressivos, uma nova indústria, como a de softwares, passa a depender de folclore."
>
> Tom DeMarco

Hoje, a maioria dos profissionais especializados em engenharia de software reconhece os mitos pelo que eles são — atitudes enganosas que causaram sérios problemas tanto para gerentes quanto para pessoal técnico. No entanto, atitudes e hábitos antigos são difíceis de modificar e remanescentes de mitos de software ainda são acreditados.

Mitos da gerência. Os gerentes com responsabilidade sobre o software, como os gerentes na maioria das atividades, estão freqüentemente sob pressão para obedecer a orçamentos, manter cronogramas no prazo e melhorar a qualidade. Como uma pessoa que está se afogando e se agarra em um pedaço de madeira, um gerente de software freqüentemente se agarra à crença em um mito de software, se isso puder diminuir a pressão (ainda que temporariamente).

Mito: *Já temos um livro que está cheio de padrões e procedimentos para elaborar o software. Isso não fornece ao meu pessoal tudo o que ele precisa saber?*

Realidade: O livro de padrões pode muito bem existir, mas será que é usado? Os profissionais de software sabem da sua existência? Ele reflete as práticas modernas da engenharia de software? É completo? É adaptável? Está voltado para melhorar o prazo de entrega, mantendo o foco na qualidade? Em muitos casos, a resposta a todas essas perguntas é não.

Veja na Web

A Rede de Gerentes de Projetos de Software pode ajudá-lo a descartar esse e outros mitos. Ela pode ser encontrada em www.spmn.com.

Mito: *Se nos atrasarmos no cronograma, podemos adicionar mais programadores e ficar em dia (algumas vezes denominado* conceito da Horda mongólica*).*

Realidade: O desenvolvimento de softwares não é um processo mecanizado como o de fabricação. Nas palavras de Brooks [BRO75]: "Adicionar pessoas a um projeto de software atrasado atrasa-o ainda mais". A princípio, essa afirmação pode parecer contra-intuitiva. Todavia, sempre que novas pessoas são adicionadas, o pessoal que estava trabalhando precisa gastar tempo orientando os recém-chegados, reduzindo assim a quantidade de tempo empregado no esforço de desenvolvimento produtivo. Pessoas podem ser adicionadas, mas apenas de forma planejada e bem coordenada.

Mito: *Se eu decidir terceirizar um projeto de software vou poder relaxar e deixar que aquela firma o elabore.*

Realidade: Se uma organização não sabe como gerir e controlar projetos de software internamente, certamente terá problemas quando terceirizar esses projetos.

Mitos do cliente. Um cliente que encomenda software para computador pode ser uma pessoa na mesa vizinha, um grupo técnico em outra sala, o departamento de promoção/vendas, ou uma outra empresa que encomendou software sob contrato. Em muitos casos, o cliente acredita em mitos de software, porque os gerentes e profissionais de software fazem pouco para corrigir essa desinformação. Mitos levam a falsas expectativas (pelo cliente) e, em última análise, à insatisfação com o desenvolvedor.

AVISO

"Trabalhe bastante para entender o que deve ser feito antes de começar. Você poderá não ser capaz de elaborar todos os detalhes, mas quanto mais você souber, menos risco correrá."

Mito: *O estabelecimento geral de objetivos é suficiente para iniciar a escrita de programas — podemos fornecer os detalhes posteriormente.*

Realidade: Embora uma declaração abrangente e estável dos requisitos nem sempre seja possível, uma definição inicial malfeita é a principal causa de esforços malsucedidos de software. Uma descrição formal e detalhada do domínio da informação, da função, do comportamento, do desempenho, das interfaces, das restrições de projeto e dos critérios de validação é essencial. Essas características podem ser determinadas somente depois de intensa comunicação entre o cliente e o desenvolvedor.

Mito: *Os requisitos de projeto mudam contiuamente, mas as mudanças podem ser facilmente acomodadas porque o software é flexível.*

Realidade: É verdade que os requisitos de software mudam, mas o impacto da mudança varia com a época em que é introduzida. Quando mudanças são solicitadas antecipadamente (antes que o projeto e a codificação tenham começado), o impacto no custo é relativamente baixo.[8] No entanto, à medida que o tempo passa, o impacto de custo cresce rapidamente — recursos foram comprometidos, a estrutura do projeto foi estabelecida e a mudança pode causar conseqüências que exijam recursos adicionais e grandes modificações no projeto.

Mitos do profissional. Os mitos que ainda têm crédito entre os profissionais de software sobreviveram a mais de 50 anos de cultura de programação. Durante os primeiros dias do software, a programação era vista como uma forma de arte. Maneiras e atitudes antigas custam a morrer.

AVISO

Sempre que você pensar que não tem tempo para a atividade de engenharia de software, pergunte a si mesmo: "Vamos ter tempo para fazer tudo de novo?".

Mito: *Quando escrevemos um programa e o fazemos funcionar, nosso trabalho está completo.*

Realidade: Alguém disse um dia que "quanto mais cedo você começar a escrever código, mais vai demorar para acabar". Dados da indústria indicam que entre 60% e 80% de todo o esforço despendido em software vai ser despendido depois de ele ser entregue ao cliente pela primeira vez.

Mito: *Até que eu esteja com o programa "rodando" não tenho como avaliar a sua qualidade.*

Realidade: Um dos mecanismos mais eficazes de garantia de qualidade de software pode ser aplicado a partir do início de um projeto — a *revisão técnica formal.* Revisões de software (descritas no Capítulo 26) são um "filtro de qualidade" que descobriu-se ser mais eficaz do que testes para encontrar alguns tipos de erros de software.

Mito: *O único produto de trabalho que pode ser entregue para um projeto de software bem-sucedido é o programa executável.*

Realidade: Um programa executável é apenas uma parte de uma *configuração de software* que inclui vários elementos. A documentação fornece a base para uma engenharia bem-sucedida e, mais importante, orienta para o suporte ao software.

8 Muitos engenheiros de software têm adotado uma abordagem "ágil" que acomoda incrementalmente as modificações, controlando assim impacto e custo. Métodos ágeis são discutidos no Capítulo 4.

Document shared on www.docsity.com

Mito: *A engenharia de software vai nos fazer criar documentação volumosa e desnecessária que certamente nos atrasará.*

Realidade: A engenharia de software não se relaciona à criação de documentos. Refere-se à criação de qualidade. Melhor qualidade leva à redução de retrabalho. E menor retrabalho resulta em tempos de entrega mais rápidos.

Muitos dos profissionais de software reconhecem a falácia dos mitos descritos. Lamentavelmente, ações e métodos habituais levam a práticas de gestão e técnicas de baixa qualidade, mesmo quando a realidade exige melhor abordagem. O reconhecimento das realidades do software é o primeiro passo em direção à formulação de soluções práticas para a engenharia de software.

1.6 Como Tudo Começa

Todo projeto de software é iniciado por alguma necessidade do negócio — a necessidade de corrigir um defeito em uma aplicação existente; a necessidade de adaptar um sistema legado para mudar o ambiente do negócio; a necessidade de estender as funções e características de uma aplicação existente; ou a necessidade de criar um novo produto, serviço ou sistema.

No início de um projeto de engenharia de software, a necessidade do negócio é muitas vezes expressa informalmente, como parte de uma simples conversa. A conversa apresentada em *CasaSegura* é típica.

Com exceção de uma referência passageira, o software praticamente não é mencionado como parte da conversa. No entanto, o software será tudo ou nada para a linha de produtos *CasaSegura*. O esforço de engenharia terá sucesso somente se o software *CasaSegura* tiver sucesso. O mercado aceitará o produto somente se o software embutido dentro dele atender às necessidades dos clientes (ainda que não fixadas). Prosseguiremos com a engenharia de software do *CasaSegura* nos capítulos seguintes.

CasaSegura[9]

Como o Projeto Começa

A cena: Sala de reunião na empresa CPI, uma empresa (fictícia) que fabrica produtos de consumo para uso residencial e comercial.

Os personagens: Mal Golden, gerente sênior, desenvolvimento de produtos; Lisa Perez, gerente de marketing; Lee Warren, gerente de engenharia; Joe Camalleri, vice-presidente executivo, desenvolvimento de negócios.

A conversa:

Joe: Tudo bem, Lee, o que é isto que eu ouvi sobre vocês desenvolverem o quê? Uma caixa sem-fio universal genérica?

Lee: É muito interessante, quase do tamanho de uma caixa de fósforo pequena. Pode-se ligá-la a sensores de todas as naturezas, a uma câmera digital, a praticamente qualquer coisa. Usa o protocolo sem fio 802.11b. Permite acessar a saída de um dispositivo sem fio. Estamos pensando que ela vai levar a toda uma nova geração de produtos.

Joe: Você concorda, Mal?

Mal: Sim. De fato, com as vendas tão achatadas como elas têm estado este ano, precisamos de algo novo. Lisa e eu estivemos fazendo um pouco de pequisa de mercado e achamos que temos uma linha de produtos que pode ser grande.

Joe: De que tamanho... Acima do nível mínimo?

Mal (evitando um comprometimento direto): Conte a ele a nossa idéia, Lisa.

Lisa: É toda uma nova geração do que chamamos de "produtos de gestão doméstica". Nós o chamamos de *CasaSegura*. Eles usam a nova interface sem-fio, fornecem a usuários domésticos ou de pequenos negócios um sistema que é controlado pelo seu PC – segurança residencial, vigilância residencial, controle de eletrodomésticos e dispositivos. Você sabe, desligar o ar-condicionado da residência enquanto está indo para casa, essa espécie de coisa.

Lee (entrando na conversa): A engenharia fez um estudo de viabilidade técnica dessa idéia, Joe. É exeqüível a baixo custo de fabricação. A maior parte do hardware é de prateleira. Software já é mais complicado, mas nada que não possamos fazer.

Joe: Interessante. Agora, eu perguntei sobre o nível mínimo.

Mal: Os PCs têm penetrado 60% de todos os domicílios dos EUA. Se pudéssemos dar a essa coisa o preço adequado, ela poderia ser um eletrodoméstico campeão. Ninguém mais tem a nossa caixa sem-fio — ela é de nossa propriedade intelectual. Vamos ter uma vantagem de dois anos sobre a concorrência. Receita? Pode ser tanto quanto 30 a 40 milhões de dolares no segundo ano.

Joe (sorrindo): Vamos levar isso para o nível acima. Eu estou interessado.

9 O projeto *CasaSegura* será usado ao longo deste livro para ilustrar o trabalho interno de uma equipe de projeto à medida que ela constrói um produto de software. A empresa, o projeto e o pessoal são totalmente fictícios, mas as situações e os problemas são reais.

1.7 Resumo

O software tornou-se o elemento-chave na evolução de sistemas e produtos baseados em computador, e uma das tecnologias mais importantes em todo o mundo. Ao longo dos últimos 50 anos, o software evoluiu de um ferramental especializado em solução de problemas e análise de informações para um produto de indústria. Mas ainda temos problemas na construção de software de alta qualidade no prazo e dentro do orçamento. O software — programas, dados e documentos — dirigido a uma ampla variedade de tecnologias e aplicações, continua a obedecer a uma série de leis que permanecem as mesmas há cerca de 30 anos. O intuito da engenharia de software é fornecer uma estrutura para a construção de software com alta qualidade.

Referências Bibliográficas

[BRO75] Brooks, F., *The Mythical Man-Month*, Addison-Wesley, 1975.

[DAC03] Daconta, M., L. Obrst, e Smith, K., *The Semantic Web*, Wiley, 2003.

[DAY99] Dayani-Fard, H., *et al.*, "Legacy Software Systems: Issues, Progress, and Challenges". IBM Technical Report: TR-74.165-k, abr. 1999, http://www.cas.ibm.com/toronto/publications/TR-74.165/k/legacy.html.

[DEM95] DeMarco, T., *Why does Software Cost so Much?*, Dorset House, 1995.

[FEI83] Feigenbaum, E. A. e McCorduck, P., *The Fifth Generation*, Addison-Wesley, 1983.

[HAM93] Hammer, M., e Champy, J., *Reengineering the Corporation*, HarperCollins Publishers, 1993.

[JOH01] Johnson, S., *Emergence: The Connected Lives of Ants, Brains, Cities, and Software*, Scribner, 2001.

[KAR99] Karlson, E. e Kolber, J., *A Basic Introduction to Y2K: How the Year 2000 Computer Crisis Affects You*, Next Era Publications, 1999.

[LEH97a] _____, e Belady L., *Program Evolution: Processes of Softwre Change*, Academic Press, 1997.

[LEH97b] _____, *et al.*, "Metrics and Laws of Software Evolution – The Nineties View", *Proceedings of the 4th International Software Metrics Symposium (METRICS´97)*, IEEE, 1997, pode ser feito download em http://www.ece.utexas.edu/~perry/work/papers/feast1.pdf.

[LEV95] Levy, S., "The Luddites Are Back", *Newsweek*, p. 55, 12 jun. 1995.

[LIP02] Lippman, A., "Round 2.0", *Context Magazine*, ago. 2002, http://www.contextmag.com/.

[LIU98] Liu, K. , *et al.*, "Report on the First SEBPC Workshop on Legacy Systems", Durham University, fev. 1998, http://www.dur.ac.uk/CSM/SABA/legacy-wksp/report.html.

[NAI82] Naisbitt, J., *Megatrends*, Warner Books, 1982.

[OSB79] Osborne, A., *Running Wild — The Next Industrial Revolution*, Osborne/McGraw-Hill, 1979.

[STO89] Stoll, C., *The Cuckoo's Egg*, Doubleday, 1989.

[TOF80] Toffler, A., *The Third Wave*, Morrow, 1980.

[TOF90] _____, *Powershift*, Bantam Publishers, 1990.

[WIL02] Williams, S., "A Unified Theory of Software Evolution", salon.com, 2002, http://www.salon.com/tech/feature/2002/04/08/lehman/index.html.

[WOL02] Wolfram, S., *A New Kind of Science*, Wolfram Media, Inc., 2002.

[YOU92] Yourdon, E., *The Decline and Fall of the American Programmer*, Yourdon Press, 1992.

[YOU96] _____, *The Rise and Resurrection of the American Programmer*, Yourdon Press, 1996.

[YOU98a] _____, Yourdon J., *Time Bomb 2000*, Prentice-Hall, 1998.

[YOU98b] _____, *Death March Projects*, Prentice-Hall, 1998.

[YOU02] _____, *Byte Wars*, Prentice-Hall, 2002.

Problemas e Pontos a Considerar

1.1 Dê no mínimo cinco exemplos adicionais de como a lei das conseqüências não-pretendidas se aplica ao software de computador.

Document shared on www.docsity.com

1.2 Dê alguns exemplos (tanto positivos quanto negativos) que indicam o impacto do software na nossa sociedade. Leia uma das referências anteriores a 1990 fornecidas na Seção 1.1 e indique em que as previsões do autor estavam corretas e em que estavam erradas.

1.3 Desenvolva as suas próprias respostas para as questões feitas na Seção 1.1. Discuta-as com os colegas.

1.4 A definição para software apresentada na Seção 1.2 aplica-se a sites Web? Se você responder sim, indique as principais diferenças entre um site e um software convencional se houver.

1.5 Muitas aplicações modernas modificam-se freqüentemente — antes de elas serem apresentadas aos seus usuários finais ou depois da primeira versão ser colocada em uso. Sugira alguns modos para se construir software que não se deteriore com as modificações.

1.6 Considere as sete categorias de software apresentadas na Seção 1.3. Pode a mesma abordagem de engenharia de software ser aplicada a cada uma delas? Justifique a sua resposta.

1.7 Selecione um dos novos desafios apresentados na Seção 1.3 (ou uma das mais novas modificações que apareceram depois que este livro foi impresso) e descreva em uma folha de papel a tecnologia e os desafios propostos para a engenharia de software.

1.8 Descreva a *Lei da Conservação da Estabilidade Organizacional* (Seção 1.4.2) com suas próprias palavras.

1.9 Descreva a *Lei da Conservação da Familiaridade* (Seção 1.4.2) com suas próprias palavras.

1.10 Descreva a *Lei da Qualidade Declinante* (Seção 1.4.2) com suas próprias palavras.

1.11 À medida que o software torna-se mais difundido, os riscos para o público (por causa de programas errados) tornam-se uma preocupação crescente e significativa. Desenvolva um cenário catastrófico realístico (diferente do erro do ano 2000) em que a falha de um programa de computador pôde provocar grande prejuízo (econômico ou humano).

1.12 Consulte o grupo de notícias da Internet *comp.risks* e prepare um resumo dos riscos para o público que foram recentemente discutidos. Uma fonte alternativa é a *Software Engineering Notes* publicada pela ACM.

Leituras e Fontes de Informação Adicionais[10]

Literalmente, milhares de livros são escritos sobre software de computador. A grande maioria aborda linguagens de programação ou aplicações de software, mas alguns discutem o software propriamente dito. Pressman e Herron (*Software Shock*, Dorset House, 1991) apresentaram uma discussão pioneira (dirigida a leigos) do software e da maneira pela qual os profissionais o elaboram.

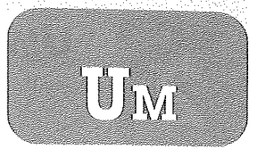
O livro campeão de vendas da Negroponte (*Being Digital*, Alfred A. Knopf, 1995) fornece um panorama da computação e de seu impacto global no vigésimo primeiro século. DeMarco [DEM 95] produziu uma coleção de ensaios divertidos e reveladores sobre software e o processo pelo qual ele é desenvolvido. Livros de Norman (*The Invisible Computer*, MIT Press, 1998) e de Bergman (*Information Appliances and Beyond*, Academic Press/Morgan Kaufmann, 2000) sugerem que o impacto, amplamente disseminado, do computador pessoal entrará em declínio à medida que os aparelhos de informação e a difusão da computação conectarem todos do mundo industrializado e quase todos os aparelhos "que eles possuam" a uma nova infra-estrutura de Internet.

Minasi (*The Software Conspiracy: Why Software Companies Put Out Faulty Products, How They Can Hurt You, and What You Can Do*, McGraw-Hill, 2000) argumenta que a "onda moderna" de erros de software pode ser eliminada e sugere modos de consegui-lo. Compaine (*Digital Divide: Facing a Crisis or Creating a Myth*, MIT Press, 2001) argumenta que a "divisão" entre as pessoas que têm acesso a fontes de informação (por exemplo, a Internet) e aquelas que não têm está reduzindo-se à medida que avançamos na primeira década deste século.

Uma grande variedade de fontes de informação sobre tópicos relacionados a softwares e gestão está disponível na Internet. Uma lista atualizada dessas referências, que são relevantes para o software, pode ser encontrada no site do autor:
htpp://www.mhhe.com/pressman.

10 A seção *Leituras e Fontes de Informação Adicionais* apresentada na conclusão de cada capítulo apresenta um breve panorama das fontes impressas que podem ajudar a expandir o seu entendimento dos principais tópicos apresentados no capítulo. Criamos um site abrangente para apoiar este livro em *http://www.mhhe.com/pressman*. Entre os vários tópicos tratados nesse site estão, capítulo a capítulo, recursos de engenharia de software e informação com base na Web que pode complementar o material apresentado em cada capítulo.

PARTE



O Processo de Software

Nesta parte de *Engenharia de Software*, você aprenderá sobre o processo que fornece uma estrutura para a prática da engenharia de software. Essas são as questões que serão abordadas nos capítulos seguintes:

- O que é um processo de software?
- Quais são as atividades genéricas de arcabouço (estruturais) que estão presentes em todo processo de software?
- Como os processos são modelados e quais os padrões de processo?
- Quais são os modelos de processo prescritivos e quais são seus pontos fortes e fracos?
- Quais características de modelos incrementais os tornam adequados a modernos projetos de software?
- O que é o processo unificado?
- Por que "agilidade" é uma palavra-guia no trabalho moderno de engenharia de software?
- O que é desenvolvimento ágil de software e como ele difere dos modelos de processo mais tradicionais?

Quando essas questões forem respondidas você estará melhor preparado para entender o contexto no qual a prática de engenharia de software é aplicada.

Document shared on www.docsity.com

CAPÍTULO 2

PROCESSO: UMA VISÃO GENÉRICA

CONCEITOS-CHAVE

arcabouço de processo 20
atividades guarda-chuva ... 20
avaliação de processo 29
CMMI 21
conjunto de tarefas 20
ISO 9001:2000 28
padrões de processo 24
PSP 29
tecnologia de processo 31
TSP 30

Em um livro fascinante, que fornece uma visão da economia do software e da engenharia de software, Howard Baetjer, Jr. [BAE98], faz os seguintes comentários sobre o processo de software:

> Desde que o software, como todo capital, é conhecimento incorporado, e como esse conhecimento está inicialmente disperso, tácito, latente e incompleto na sua totalidade, o desenvolvimento de software é um processo de aprendizado social. O processo é um diálogo no qual o conhecimento, que deve se transformar em software, é reunido e incorporado ao software. O processo fornece interação entre usuários e projetistas, entre usuários e ferramentas em desenvolvimento e entre projetistas e ferramentas em desenvolvimento [tecnologia]. É um processo iterativo no qual a própria ferramenta serve como meio de comunicação, com cada nova rodada de diálogo explicitando mais conhecimento útil do pessoal envolvido.

Efetivamente, a elaboração de software de computador é um processo iterativo de aprendizado, e o resultado, algo que Baetjer chamaria de "essência do software", é uma incorporação de conhecimentos coletados, destilados e organizados, à medida que o processo é conduzido.

Mas o que exatamente é um processo de software do ponto de vista técnico? Dentro do contexto deste livro, definimos *processo de software* como um arcabouço para as tarefas que são necessárias para construir softwares de alta qualidade. *Processo* é sinônimo de engenharia de software? A resposta é "sim" e "não". Um processo de software define a abordagem que é adotada quando o software é elaborado. Mas a engenharia de software também inclui tecnologias que constituem um processo — métodos técnicos e ferramentas automatizadas.

Mais importante, a engenharia de software é realizada por pessoas criativas, com amplos conhecimentos, que devem adaptar um processo de software maduro apropriado para os produtos que elas constroem e para as demandas do seu mercado.

PANORAMA

O que é? Quando você elabora um produto ou sistema é importante percorrer uma série de passos previsíveis — um roteiro que o ajuda a criar a tempo um resultado de alta qualidade. O roteiro que você segue é chamado de *processo de software*.

Quem faz? Os engenheiros de software e seus gerentes adaptam o processo a suas necessidades e depois o seguem. Além disso, o pessoal que solicitou o software tem um papel a desempenhar no processo de defini-lo, construí-lo e testá-lo.

Por que é importante? Porque fornece estabilidade, controle e organização para uma atividade que pode, se deixada sem controle, tornar-se bastante caótica. No entanto, uma abordagem moderna de engenharia de software precisa ser "ágil". Precisa exigir apenas aquelas atividades, controles e documentação adequados à equipe de projeto e ao produto que deve ser produzido.

Quais são os passos? Em nível de detalhe, o processo adotado depende do software que você está construindo. Um processo poderia ser apropriado à criação de softwares para o sistema de aviônica de uma aeronave, enquanto um processo inteiramente diferente seria indicado para a criação de um site.

Qual é o produto do trabalho? Do ponto de vista de um engenheiro de software, os produtos do trabalho são os programas, documentos e dados produzidos em conseqüência das atividades e tarefas definidas pelo processo.

Como tenho certeza de que fiz corretamente? Existem diversos mecanismos de avaliação do processo de software que permitem às organizações determinar a "maturidade" do seu processo de software. Todavia, a qualidade, pontualidade e viabilidade a longo prazo do produto que você constrói são os melhores indicadores da eficácia do processo usado.

2.1 ENGENHARIA DE SOFTWARE — UMA TECNOLOGIA EM CAMADAS

Apesar de centenas de autores terem desenvolvido definições pessoais de *engenharia de software*, uma definição proposta por Fritz Bauer [NAU69] na conferência pioneira sobre o assunto ainda serve de base para a discussão:

> [Engenharia de software é] a criação e a utilização de sólidos princípios de engenharia a fim de obter softwares econômicos que sejam confiáveis e que trabalhem eficientemente em máquinas reais.

Quase todo leitor ficará tentado a ampliar essa definição. Ela diz pouco sobre os aspectos técnicos da qualidade do software; não inclui diretamente a necessidade de satisfação do cliente ou pontualidade; não faz menção à importância de medidas e unidades; não fala sobre a importância de um processo amadurecido. Ainda assim, a definição de Bauer nos dá uma linha básica. Quais são os "sólidos princípios de engenharia" que podem ser aplicados ao desenvolvimento de softwares de computador? Como construímos softwares "economicamente" e de modo que eles sejam "confiáveis"? O que é necessário para criar programas de computador que trabalhem "eficientemente" em não apenas uma, mas em várias "máquinas reais"? Essas são questões que continuam a desafiar os engenheiros de software.

> "Mais que uma atividade ou um corpo de conhecimentos, a engenharia é um verbo, uma palavra de ação, uma forma de abordar um problema."
>
> **Scott Whitmire**

? *Como definir engenharia de software?*

O IEEE [IEE93] desenvolveu uma definição mais abrangente ao estabelecer:

> Engenharia de software: (1) aplicação de uma abordagem sistemática, disciplinada e quantificável, para o desenvolvimento, operação e manutenção do software; isto é, a aplicação da engenharia ao software. (2) O estudo de abordagens como as de (1).

E no entanto, o que é "sistemático, disciplinado e quantificável" para uma equipe de software pode ser cansativo para outra. Precisamos de disciplina, mas precisamos também de adaptabilidade e agilidade.

PONTO CHAVE

A engenharia de software inclui um processo, métodos e ferramentas.

A engenharia de software é uma tecnologia em camadas. De acordo com a Figura 2.1, qualquer abordagem de engenharia (inclusive a engenharia de software) deve se apoiar num compromisso organizacional com a qualidade. Gestão de Qualidade Total, Seis Sigmas (Six Sigma) e filosofias análogas levam à cultura de um processo contínuo de aperfeiçoamento, e é essa cultura que, em última análise, leva ao desenvolvimento de abordagens cada vez mais efetivas para a engenharia de software. A base em que se apóia é o *foco na qualidade*.

O alicerce da engenharia de software é a camada de *processo*. O processo de engenharia de software é o adesivo que mantém unidas as camadas de tecnologia e permite o desenvolvimento racional e oportuno de softwares de computador. O processo define um arcabouço [PAU93] que deve ser estabelecido para a efetiva utilização da tecnologia de engenharia de software. Os processos de software formam a base para o controle gerencial de projetos de software e estabelecem o contexto no qual os métodos técnicos são aplicados, os produtos de trabalho (modelos, documentos, dados,

FIGURA 2.1

Engenharia de Software em camadas

Document shared on www.docsity.com

Veja na Web

Cross Talk é um jornal que fornece informações pragmáticas sobre processo, métodos e ferramentas. Ele pode ser encontrado em www.stsc.hill.af.mil.

relatórios, formulários etc.) são produzidos, os marcos são estabelecidos, a qualidade é assegurada e as modificações são adequadamente geridas.

Os *métodos* de engenharia de software fornecem a técnica de "como fazer" para construir softwares. Eles abrangem um amplo conjunto de tarefas que incluem comunicação, análise de requisitos, modelagem de projeto, construção de programas, testes e manutenção. Os métodos de engenharia de software repousam num conjunto de princípios básicos que regem cada área da tecnologia e incluem atividades de modelagem e outras técnicas descritivas.

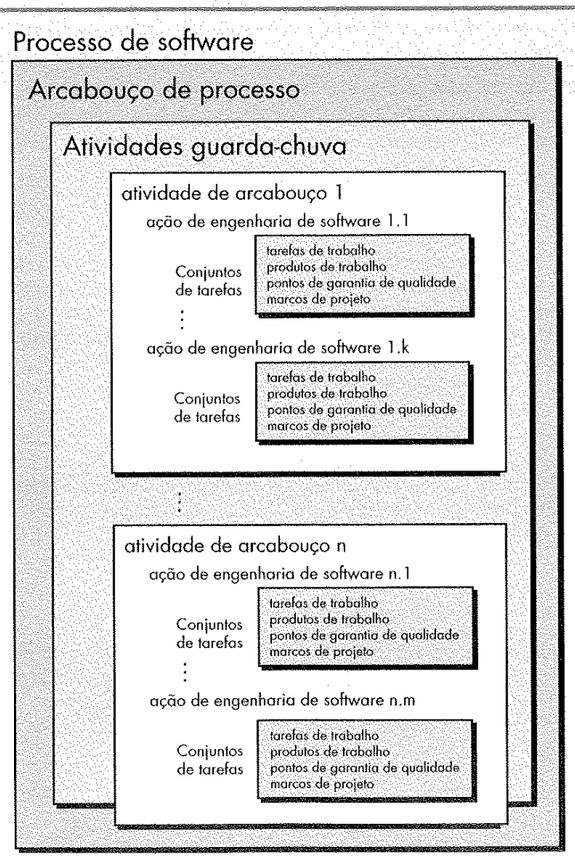
As *ferramentas* de engenharia de software fornecem apoio automatizado ou semi-automatizado para o processo e para os métodos. Quando ferramentas são integradas de modo que a informação criada por uma ferramenta possa ser usada por outra, um sistema de apoio ao desenvolvimento de software, chamado *engenharia de software apoiada por computador*, é estabelecido.

2.2 Um Arcabouço de Processo

Um *arcabouço de processo* estabelece o alicerce para um processo de software completo pela identificação de um pequeno número de *atividades de arcabouço* aplicáveis a todos os projetos de software, independentemente de seu tamanho ou complexidade. Além disso, o arcabouço de processo engloba um conjunto de *atividades guarda-chuva* que são aplicáveis durante todo o processo de software.

Figura 2.2

Um arcabouço de processo de software



Com referência à Figura 2.2, cada atividade de arcabouço é preenchida por um conjunto de *ações de engenharia de software* — uma coleção de tarefas relacionadas que produz um produto importante do trabalho de engenharia de software (por exemplo, o *projeto* é uma ação de engenharia de software). Cada ação é preenchida por *tarefas de trabalho* individuais que realizam alguma parte do trabalho determinado pela ação.

> "Um processo define quem está fazendo *o quê*, *quando* e *como* para alcançar um certo objetivo."
>
> **Ivar Jacobson, Grad Booch e James Rumbaugh**

O seguinte *arcabouço de processo genérico* (usado como base para a descrição dos modelos de processo nos capítulos subseqüentes) é aplicável à grande maioria dos projetos de software:

? Quais são as cinco atividades genéricas do arcabouço de processo?

Comunicação. Essa atividade de arcabouço envolve alta comunicação e colaboração com o cliente (e outros interessados)[1] e abrange o levantamento de requisitos e outras atividades relacionadas.

Planejamento. Essa atividade estabelece um plano para o trabalho de engenharia de software que se segue. Descreve as tarefas técnicas a ser conduzidas, os riscos prováveis, os recursos que serão necessários, os produtos de trabalho a ser produzidos e um cronograma do trabalho.

Modelagem. Essa atividade inclui a criação de modelos que permitam ao desenvolvedor e ao cliente,entender melhor os requisitos do software e o projeto que vai satisfazer a esses requisitos.

Construção. Essa atividade combina geração de código (quer manual ou automática) e os testes necessários para revelar erros no código.

Implantação. O software (como entidade completa ou incremento parcialmente completo) é entregue ao cliente, que avalia o produto entregue e fornece *feedback* com base na avaliação.

Essas cinco atividades genéricas de arcabouço podem ser usadas durante o desenvolvimento de pequenos programas, durante a criação de grandes aplicações para a Internet e para a engenharia de grandes e complexos sistemas baseados em computador. Os detalhes do processo de software serão muito diferentes em cada caso, mas as atividades de arcabouço permanecem as mesmas.

> "Einstein argumentou que deve haver uma explicação simplificada da natureza, porque Deus não é inconstante ou arbitrário. Nenhuma dessas fés conforta o engenheiro de software. Muito da complexidade que ele deve controlar é complexidade arbitrária."
>
> **Fred Brooks**

Ponto Chave

Diferentes projetos necessitam de diferentes conjuntos de tarefas. a equipe de software escolhe o conjunto de tarefas com base no problema e características do projeto.

Usando um exemplo derivado do arcabouço de processo genérico, a *atividade de modelagem* é composta de duas ações de engenharia de software — *análise* e *projeto*. A análise[2] engloba um conjunto de tarefas de trabalho (por exemplo, levantamento, elaboração, negociação, especificação e validação de requisitos) que conduzem à criação de modelos de análise (e/ou especificação de requisitos). O projeto engloba tarefas de trabalho (projeto de dados, projeto arquitetural, projeto de interface e projeto de componentes) que criam um modelo (e/ou uma especificação de projeto).[3]

Com referência novamente à Figura 2.2, cada ação de engenharia de software é representada por um número diferente de *conjuntos de tarefas* — cada um como uma coleção de tarefas de trabalho de engenharia de software, produtos de trabalho relacionados, pontos de garantia de qualidade e

1 Um "interessado" é alguém que tem interesse no resultado bem-sucedido do projeto — gerentes de negócio, usuários finais, engenheiros de software, pessoal de apoio etc. Rob Thomsett brinca que "um interessado é uma pessoa empunhando uma estaca grande e afiada (...) se você não prestar atenção nos seus interessados, você sabe onde a estaca vai acabar ficando".

* N.R.T. Essa brincadeira é porque o termo usado para "interessado" em inglês, *stakeholder*, tem o sentido literal de "porta-estaca".

2 A análise será discutida em profundidade nos Capítulos 7 e 8.

3 Deve-se notar que a "modelagem" deve ser interpretada de modo diferente quando a manutenção de um software existente é realizada. Em alguns casos, a modelagem de análise e do projeto de fato ocorre, mas em outras situações de manutenção, a modelagem pode ser usada para ajudar a entender o software legado bem como para representar adições ou modificações nele.

Document shared on www.docsity.com

marcos de projeto. O conjunto de tarefas escolhido é o que melhor acomoda as necessidades do projeto e as características da equipe. Isso implica que uma ação de engenharia de software (por exemplo, o projeto) pode ser adaptada às necessidades específicas do projeto de software e às características da equipe do projeto.

INFO

Conjunto de Tarefas

Um conjunto de tarefas define o trabalho real a ser feito para atingir os objetivos de uma ação de engenharia de software. Por exemplo, "levantamento de requisitos" é uma importante ação de engenharia de software que ocorre durante a atividade **comunicação**. O objetivo do levantamento de requisitos é entender o que vários interessados desejam do software que está sendo construído.

Para um projeto pequeno e relativamente simples, o conjunto de tarefas para o levantamento de requisitos pode ter o seguinte aspecto:

1. Faça uma lista de interessados no projeto.
2. Convide todos os interessados para uma reunião informal.
3. Peça a cada interessado para fazer uma lista das características e funções desejadas.
4. Discuta os requisitos e construa uma lista final.
5. Priorize os requisitos.
6. Observe áreas de incerteza.

Para um projeto de software maior e mais complexo, um conjunto de tarefas diferente seria necessário. Ele pode incluir as seguintes tarefas de trabalho:

1. Faça uma lista de interessados no projeto.
2. Entreviste cada interessado separadamente para determinar seus desejos e necessidades gerais.
3. Construa uma lista preliminar de funções e características com base nas informações fornecidas pelos interessados.
4. Programe uma série de reuniões facilitadas de levantamento de requisitos.
5. Realize as reuniões.
6. Produza cenários informais de usuário como parte de cada reunião.
7. Refine os cenários de usuário com base no feedback dos interessados.
8. Construa uma lista revisada de requisitos dos interessados.
9. Use técnicas de implantação de função de qualidade para priorizar os requisitos.
10. Empacote os requisitos de modo que eles possam ser entregues incrementalmente.
11. Anote as restrições e limitações que serão colocadas no sistema.
12. Discuta métodos para validação do sistema.

Ambos os conjuntos de tarefas realizam o levantamento de requisitos, mas eles são muito diferentes em profundidade e formalidade. A equipe de software escolhe o conjunto de tarefas que lhe permitir alcançar o objetivo de cada atividade de processo e a ação de engenharia de software e, ainda, manter a qualidade e a agilidade.

O arcabouço descrito na visão genérica de engenharia de software é complementado por várias atividades *guarda-chuva*. Atividades típicas dessa categoria incluem:

PONTO CHAVE

Atividades guarda-chuva ocorrem ao longo do processo de software e focalizam principalmente a gestão, o monitoramento e o controle do projeto.

Acompanhamento e controle de projeto de software — permite à equipe de software avaliar o progresso com base no plano de projeto e tomar a ação necessária para manter o cronograma.

Gestão de risco — avalia os riscos que podem afetar o resultado do projeto ou a qualidade do produto.

Garantia de qualidade de software — define e conduz as atividades necessárias para garantir a qualidade do software.

Revisões técnicas formais — avaliam os produtos de trabalho da engenharia de software, num esforço para descobrir e remover erros antes que eles sejam propagados para a próxima ação ou atividade.

Medição — define e reúne medidas de processo, projeto e produto que ajudam a equipe a entregar um software que satisfaça às necessidades do usuário; pode ser usada conjugada com todas as outras atividades de arcabouço e guarda-chuva.

Gestão de configuração de software — gerencia os efeitos das modificações ao longo de todo o processo de software.

Gestão de reusabilidade — define critérios para a reutilização dos produtos de trabalho (inclusive componentes de software) e estabelece mecanismos para obter componentes reusáveis.

Preparação e produção do produto do trabalho — abrange as atividades necessárias para criar produtos do trabalho como modelos, documentos, registros, formulários e listas.

As atividades guarda-chuva são aplicadas ao longo de todo o processo de software e serão discutidas em detalhes mais adiante neste livro.

Todos os modelos de processo podem ser caracterizados dentro do arcabouço de processo mostrado na Figura 2.2. A aplicação inteligente de qualquer modelo de processo de software deve reconhecer que a adaptação (ao problema, ao projeto, à equipe e à cultura organizacional) é essencial para o sucesso. Mas os modelos de processo diferem fundamentalmente:

PONTO CHAVE

A adaptação de processos de software é essencial para o sucesso do projeto.

? Como os modelos de processo diferem uns dos outros?

- No fluxo geral de atividades e tarefas e interdependências entre atividades e tarefas.
- No grau em que as tarefas de trabalho são definidas dentro de cada atividade de arcabouço.
- No grau em que os produtos do trabalho são identificados e solicitados.
- Na maneira como as atividades de garantia de qualidade são aplicadas.
- No modo como o monitoramento de projeto e as atividades de controle são aplicados.
- No grau geral de detalhes e rigor em que o processo é descrito.
- No grau em que clientes e outros interessados estão envolvidos no projeto.
- No nível de autonomia da equipe de projeto de software.
- No grau em que a organização da equipe e os papéis são prescritos.

> "Sinto que uma receita é apenas um tema que um cozinheiro inteligente pode usar cada vez com uma variante."
>
> **Madame Benoit**

Modelos de processo que enfatizam a definição, identificação e aplicação detalhada de atividades e tarefas de processo têm sido aplicados na comunidade de engenharia de software durante os últimos trinta anos. Quando esses *modelos prescritivos de processo* são aplicados, o objetivo é melhorar a qualidade do sistema para tornar os projetos mais gerenciáveis, as datas de entrega e os custos mais previsíveis e para guiar equipes de engenheiros de software à medida que eles realizam o trabalho necessário para construir um sistema. Infelizmente, tem havido épocas em que esses objetivos não têm sido alcançados. Se os modelos prescritivos forem aplicados dogmaticamente e sem adaptação, eles podem aumentar o nível de burocracia associado à construção de sistemas baseados em computador e, inadvertidamente, criar dificuldades para desenvolvedores e clientes.

? O que caracteriza um processo "ágil"?

Modelos de processo que enfatizam a agilidade do projeto e seguem uma série de princípios[4] que levam a uma abordagem mais informal (mas, segundo seus proponentes, não menos efetiva) do processo de software foram propostos no últimos anos. Esses *modelos de processo ágil* enfatizam a manobrabilidade e a adaptabilidade. Eles são adequados a muitos tipos de projeto e são particularmente úteis quando aplicações Web passam por engenharia.

Que filosofia de processo de software é melhor? Essa questão tem provocado um debate emocional entre engenheiros de software e será discutida no Capítulo 4. Por hora, é importante notar que essas duas filosofias de processo têm um objetivo comum — criar softwares de alta qualidade que satisfaçam às necessidades do cliente —, mas abordagens diferentes.

2.3 O CMMI (Capability Maturity Model Integration)

O SEI (Software Engineering Institute) desenvolveu um abrangente metamodelo de processo baseado em um conjunto de capacidades de engenharia de software que devem estar presentes à medida que as empresas alcançam diferentes níveis de capacidade e maturidade de processo. Para

4 Modelos ágeis e os princípios que os guiam serão discutidos no Capítulo 4.

Document shared on www.docsity.com

Veja na Web

Informação completa sobre CMMI pode ser obtida em http://www.sei.cmu.edu/cmmi/.

conseguir essas capacidades, o SEI alega que uma organização deve desenvolver um modelo de processo (Figura 2.2) que siga as diretrizes do CMMI [CMM02].

O CMMI representa um metamodelo de processo de dois modos diferentes: (1) como um modelo *contínuo* e (2) como um modelo *em estágios*. O metamodelo CMMI contínuo descreve um processo em duas dimensões como ilustrado na Figura 2.3. Cada área de processo (por exemplo, planejamento de projeto ou gestão de requisitos) é avaliada formalmente com base em metas e práticas específicas, e é classificada de acordo com os seguintes níveis de capacitação:

Nível 0: Incompleto. A área de processo (por exemplo, gestão de requisitos) não é realizada ou não atinge todas as metas e objetivos definidos pelo CMMI para o nível 1 de capacitação.

Nível 1: Realizado. Todas as metas específicas da área de processo (como definidas pelo CMMI) foram satisfeitas. As tarefas de trabalho necessárias para produzir os produtos de trabalho definidos estão sendo conduzidas.

Nível 2: Gerido. Todos os critérios do nível 1 foram satisfeitos. Além disso, todo trabalho associado à área de processo está de acordo com a política definida pela organização; todo o pessoal que está fazendo o trabalho tem acesso aos recursos adequados para fazer o serviço; os interessados estão ativamente envolvidos nas áreas de processo, conforme necessário; todas as tarefas e produtos de trabalho são "monitorados, controlados e revisados e são avaliados quanto à aderência à descrição do processo" [CMM02].

Nível 3: Definido. Todos os critérios do nível 2 foram alcançados. Além disso, o processo é "feito sob medida para o conjunto-padrão de processos da organização, de acordo com as suas diretrizes quanto a fazer coisas sob medida e contribui com produtos de trabalho, medições e outras informações de aperfeiçoamento de processo para o patrimônio de processos da organização" [CMM02].

Nível 4: Quantitativamente gerido. Todos os critérios do nível 3 foram alcançados. Além disso, a área de processo é controlada e aperfeiçoada usando medições e avaliação quantitativa. "Objetivos quantitativos para qualidade e desempenho de processo são estabelecidos e usados como critério na gestão do processo" [CMM02].

AVISO

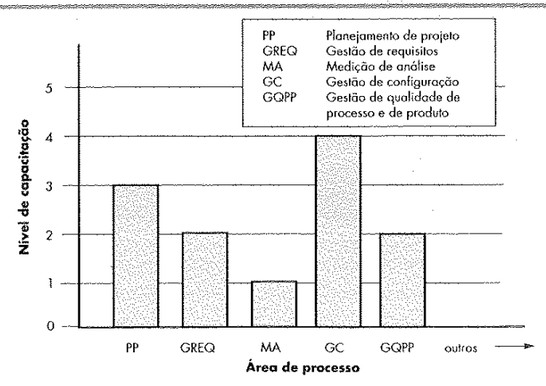
Toda organização deve tentar alcançar o objetivo do CMMI. Todavia, a implementação de todos os aspectos do modelo pode ser um exagero em algumas situações.

Nível 5: Otimizado. Todas as capacidades do nível 4 foram alcançadas. Além disso, a área de processo é adaptada e otimizada usando meios quantitativos (estatísticos) para satisfazer às alterações de necessidades do cliente e continuamente aperfeiçoar a eficácia da área de processo em consideração. [CMM02].

> "Muito da crise de software é auto-infligida, como quando um executivo-chefe de informática diz: 'Eu preferia ter isso errado do que atrasado. Podemos sempre corrigir isso depois'."
>
> **Mark Paulk**

FIGURA 2.3

Perfil da capacitação de área de processo CMMI (PH102)



O CMMI define cada área de processo em termos de "metas específicas" e das "práticas específicas" necessárias para atingir tais metas. As *metas específicas* estabelecem as características que devem existir se as atividades determinadas por uma área de processo tiverem de ser efetivas. As *práticas específicas* refinam uma meta num conjunto de atividades relacionadas ao processo.

Por exemplo, o **planejamento de projeto** é uma das oito áreas de projeto definidas pelo CMMI para a categoria "gestão de projeto".[5] As metas específicas (ME) e as práticas específicas (PE) associadas ao planejamento de projeto são [CMM02]:

Veja na Web

Informações completas bem como uma versão do CMMI para download podem ser obtidas em www.sei.cmu.edu/cmmi/.

ME 1 Estabelecer estimativas

PE 1.1-1 Estime o escopo do projeto
PE 1.2-1 Estabeleça estimativas de produto do trabalho e atributos de tarefas
PE 1.3-1 Defina o ciclo de vida do projeto
PE 1.4-1 Determine estimativas de esforço e de custo

ME 2 Desenvolver um plano de projeto

PE 2.1-1 Estabeleça o orçamento e o cronograma
PE 2.2-1 Identifique os riscos de projeto
PE 2.3-1 Planeje a gestão de dados
PE 2.4-1 Planeje os recursos de projeto
PE 2.5-1 Planeje as habilidades e conhecimentos necessários
PE 2.6-1 Planeje o envolvimento dos interessados
PE 2.7-1 Estabeleça o plano de projeto

ME 3 Obter comprometimento com o plano

PE 3.1-1 Revise planos que afetam o projeto
PE 3.2-1 Reconcilie os níveis de trabalho e os recursos
PE 3.3-1 Obtenha comprometimento com o plano

Além de metas e práticas específicas, o CMMI também define um conjunto de cinco metas genéricas e práticas relacionadas para cada área de processo. Cada uma das cinco metas genéricas corresponde a um dos cinco níveis de capacitação. Assim, para alcançar um determinado nível de capacitação, a meta genérica para aquele nível e as práticas genéricas que correspondem àquela meta precisam ser alcançadas. Para ilustrar, as metas genéricas (MG) e as práticas genéricas (PG) para a área de processo de planejamento de projeto são [CMM02]:

MG 1 Alcançar metas específicas

PG 1.1 Realize as práticas básicas

MG 2 Institucionalizar um processo gerido

PG 2.1 Estabeleça uma política organizacional
PG 2.2 Planeje o processo
PG 2.3 Forneça os recursos
PG 2.4 Atribua responsabilidades
PG 2.5 Treine o pessoal
PG 2.6 Gerencie configurações
PG2 .7 Identifique e envolva os interessados relevantes
PG 2.8 Monitore e controle o processo
PG 2.9 Avalie objetivamente a aderência
PG 2.10 Revise o estado com a gestão de nível superior

5 Outras áreas de processo definidas para "gestão de projeto" incluem: monitoração e controle de projeto, gestão de acordo com o fornecedor, gestão integrada de projeto para IPPD, gestão de risco, formação integrada de equipe, gestão integrada de fornecedor e gestão de projeto quantitativo.

Document shared on www.docsity.com

MG 3 Institucionalizar um processo definido

PG 3.1 Estabeleça um processo definido
PG 3.2 Reúna informações de aperfeiçoamento

MG 4 Institucionalizar um processo quantitativamente gerido

PG 4.1 Estabeleça objetivos quantitativos para o processo
PG 4.2 Estabilize o desempenho dos subprocessos

MG 5 Institucionalizar um processo otimizado

PG 5.1 Garanta o aperfeiçoamento contínuo do processo
PG 5.2 Corrija as causas básicas dos problemas

O modelo em estágios do CMMI define as mesmas áreas de processo, metas e práticas que o modelo contínuo. A principal diferença é que o modelo em estágios define cinco níveis de maturidade em vez de cinco níveis de capacitação. Para atingir um nível de maturidade, as metas e práticas específicas associadas a um conjunto de áreas de processo precisam ser alcançadas. O relacionamento entre os níveis de maturidade e as áreas de processo é mostrado na Figura 2.4.

2.4 Padrões de Processo

? O que é um padrão de processo?

O processo de software pode ser definido como uma coleção de padrões que definem um conjunto de atividades, ações, tarefas de trabalho, produtos de trabalho e/ou comportamentos relacionados [AMB98] necessários ao desenvolvimento de softwares de computador. Dito em termos mais gerais, um *padrão de processo* nos fornece um gabarito — um método consistente para descrever uma característica importante do processo de software. Pela combinação de padrões, uma equipe de software pode construir um processo que melhor satisfaça às necessidades de um projeto.

Figura 2.4

Áreas de processo necessárias para alcançar um nível de maturidade

Nível	Foco	Áreas de Processo
Otimizado	***Aperfeiçoamento contínuo de processo***	**Inovação e Implantação Organizacional** **Análise e Resolução Causal**
Quantitativamente gerido	***Gestão quantitativa***	**Desempenho de Processo Organizacional** **Gestão Quantitativa de Projeto**
Definido	***Padronização de processo***	**Desenvolvimento de Requisitos** **Solução Técnica** **Integração de Produtos** **Verificação** **Validação** **Foco do Processo Organizacional** **Definição do Processo Organizacional** **Treinamento Organizacional** **Gestão Integrada de Projeto** **Gestão Integrada de Fornecedor** **Gestão de Risco** **Análise e Resolução de Decisão** **Ambiente Organizacional para Integração** **Formação Integrada de Equipe**
Gerido	***Gestão básica de projeto***	**Gestão de Requisitos** **Planejamento de Projeto** **Monitoramento e Controle de Projeto** **Gestão de Acordo com Fornecedor** **Medição e Análise** **Garantia de Qualidade de Processo e de Produto**
Realizado		

INFO

O CMMI — Devemos ou Não Devemos?

O CMMI é um metamodelo de processo. Ele define (em mais de 700 páginas) as características de processo que devem existir quando uma organização deseja estabelecer um processo de software que seja completo. A questão que tem sido debatida por mais de uma década é: O CMMI exagera? Como a maioria das coisas na vida (e em softwares), a resposta não é um simples sim ou não.

O espírito do CMMI deve sempre ser adotado. Correndo o risco de supersimplificar, ele alega que o desenvolvimento de softwares deve ser levado a sério — deve ser totalmente planejado; controlado uniformemente; monitorado precisamente; e conduzido profissionalmente. Deve enfocar as necessidades dos interessados no projeto, as habilidades dos engenheiros de software e a qualidade do produto final. Esses pontos são inquestionáveis.

Os requisitos detalhados do CMMI devem ser seriamente considerados se uma organização construir sistemas grandes e complexos que envolvam dúzias ou centenas de pessoas durante muitos meses ou anos. Pode ser que o CMMI esteja totalmente certo em tais situações, isso se a cultura organizacional for obediente a modelos-padrão de processo e a gerência estiver comprometida em torná-lo um sucesso. No entanto, em outras situações, o CMMI pode ser simplesmente demais para uma organização assimilá-lo com sucesso. Isso significa que o CMMI é ruim, burocrático demais ou antiquado? Não, não significa. Significa simplesmente que o que é certo para uma cultura organizacional pode não ser para outra.

O CMMI é uma realização significativa em engenharia de software. Ele fornece uma discussão abrangente das atividades e ações que devem estar presentes quando uma organização constrói softwares de computador. Mesmo que uma organização opte por não adotá-lo em detalhes, cada equipe de software deveria abraçar seu espírito e obter inspiração a partir de sua discussão do processo e da prática de engenharia de software.

> "A repetição de padrões é uma coisa muito diferente da repetição de partes. Na verdade, as diferentes partes serão exclusivas porque os padrões são os mesmos."
>
> **Christopher Alexander**

Padrões podem ser definidos em qualquer nível de abstração.[6] Em alguns casos, um padrão pode ser usado para descrever um processo completo (por exemplo, prototipação). Em outras situações, eles podem ser usados para descrever uma importante atividade de arcabouço (por exemplo, planejamento) ou uma tarefa dentro de uma atividade de arcabouço (por exemplo, estimativa de projeto).

PONTO CHAVE

Um gabarito de padrão fornece meios consistentes para descrever um padrão.

Ambler [AMB98] propôs o seguinte gabarito para descrever um padrão de processo:

Nome do Padrão. Ao padrão é dado um nome significativo que descreva sua função dentro do processo de software (por exemplo, **comunicação-com-o-cliente**).

Intenção. O objetivo do padrão é descrito brevemente. Por exemplo, a intenção de **comunicação-com-o-cliente** é "estabelecer um relacionamento colaborativo com o cliente esforçando-se para definir o escopo do projeto, os requisitos do negócio e outras restrições de projeto". A intenção pode ser mais expandida com texto explanatório adicional e diagramas apropriados, se necessário.

Tipo. O tipo de padrão é especificado. Ambler [Amb98] sugere três tipos:

- Os *padrões de tarefa* definem uma ação de engenharia de software ou tarefa de trabalho que é parte do processo e relevante para a prática bem-sucedida da engenharia de software (por exemplo, **levantamento de requisitos** é um padrão de tarefa).

6 Os padrões são aplicáveis a muitas atividades de engenharia de software. A análise, o projeto e teste de padrões serão discutidos nos Capítulos 7, 9, 10, 12 e 14. Padrões e "antipadrões" para atividades de gestão de projeto serão discutidos na Parte 4 deste livro.

Document shared on www.docsity.com

- Os *padrões de estágio* definem uma atividade de arcabouço para o processo. Como uma atividade de arcabouço abrange várias tarefas de trabalho, um padrão de estágio incorpora vários padrões de tarefa que são relevantes para o estágio (atividade de arcabouço). Um exemplo de um padrão de estágio poderia ser **comunicação.** Esse padrão incorporaria o padrão de tarefa **levantamento de requisitos** e outros.
- Os *padrões de fase* definem a seqüência de atividades de arcabouço que ocorrem no processo, mesmo quando o fluxo global de atividades é de natureza iterativa. Um exemplo de padrão de fase poderia ser um **modelo espiral** ou **prototipação.**[7]

Contexto Inicial. As condições sob as quais o padrão se aplica são descritas. Antes da iniciação do padrão, pergunta-se (1) que atividades relacionadas à organização ou à equipe já ocorreram, (2) qual é o estado das entradas para o processo e (3) que informações de engenharia de software ou informações de projeto já existem.

Por exemplo, o padrão **planejamento** (um padrão de estágio) exige que (1) os clientes e os engenheiros de software tenham estabelecido uma comunicação colaborativa; (2) a conclusão bem-sucedida de um certo número de padrões de tarefa (especificados) para que o padrão **comunicação-com-o-cliente** tenha ocorrido; (3) o escopo do projeto, os requisitos básicos do negócio e as restrições de projeto sejam conhecidos.

Problema. O problema a ser resolvido pelo padrão é descrito. Por exemplo, o problema a ser resolvido por **comunicação-com-o-cliente** poderia ser descrito da seguinte maneira: *a comunicação entre o desenvolvedor e o cliente é freqüentemente inadequada porque um formato efetivo para fazer o levantamento de informações não é estabelecido, um mecanismo útil para registrá-las não é criado e uma revisão significativa não é realizada.*

Solução. A implementação do padrão é descrita. Esta seção discute como o estado inicial do processo (que existe antes do padrão ser implementado) é modificado em conseqüência da iniciação do padrão. Descreve também como a informação de engenharia de software ou a informação de projeto que está disponível antes da iniciação do padrão é transformada como conseqüência bem-sucedida do padrão.

Veja na Web

Recursos abrangentes sobre padrões de processo podem ser encontrados em www.ambysoft.com/processPatternsPage.html.

Contexto Resultante. As condições que resultarão quando o padrão tiver sido implementado com sucesso são descritas. Ao término do padrão pergunta-se: (1) que atividades relacionadas à organização ou à equipe devem ter ocorrido, (2) qual é o estado de saída do processo, (3) que informações de engenharia de software ou informações de projeto foram desenvolvidas.

Padrões Relacionados. Uma lista de todos os padrões de processo diretamente relacionados a este é fornecida — como uma hierarquia ou em alguma outra forma diagramática. Por exemplo, o padrão de estágio **comunicação** inclui os padrões de tarefa **montagem-da-equipe-de-projeto**, **definição-colaborativa-das-diretrizes**, **isolamento-do-escopo**, **levantamento-de-requisitos**, **descrição-de-restrições** e **criação-de-miniespec/modelos**.

Usos Conhecidos/Exemplos. São indicadas as instâncias específicas nas quais o padrão é aplicável. Por exemplo, a **comunicação** é obrigatória no início de cada projeto de software; recomendada ao longo do projeto de software; e obrigatória quando a atividade **implantação** está em curso.

Os padrões de projeto fornecem um mecanismo efetivo para descrever qualquer processo de software. Eles permitem a uma organização de engenharia de software desenvolver uma descrição hierárquica do processo começando em um alto nível de abstração (um padrão de fase). A descrição é refinada em um conjunto de padrões de estágio que descrevem atividades de arcabouço e ainda mais refinada, de modo hierárquico, em padrões de tarefa mais detalhados para cada padrão de estágio. Uma vez que os padrões de processo tenham sido desenvolvidos, eles podem ser reutilizados para a definição de variantes de processo — isto é, um modelo de processo personalizado pode ser definido por uma equipe de software usando padrões como blocos construtivos para o modelo de processo.

7 Esses padrões de fase serão discutidos no Capítulo 3.

INFO

Um Exemplo de Padrão de Processo

O seguinte padrão de processo abreviado descreve uma abordagem que pode ser aplicável quando os interessados têm uma idéia geral do que precisa ser feito, mas estão inseguros quanto aos requisitos de software específicos.

Nome do padrão. Prototipação.

Intenção. O objetivo do padrão é construir um modelo (protótipo) que possa ser avaliado iterativamente pelos interessados num esforço para identificar ou solidificar requisitos de software.

Tipo. Padrão de fase.

Contexto inicial. As seguintes condições precisam ser satisfeitas antes da iniciação deste padrão: (1) os interessados foram identificados; (2) um modo de comunicação entre os interessados e a equipe de software foi estabelecido; (3) o problema a ser resolvido foi identificado pelos interessados; (4) um entendimento inicial do escopo do projeto, dos requisitos de negócio básicos e das restrições de projeto foi desenvolvido.

Problema. Os requisitos são imprecisos ou inexistentes, no entanto há claro reconhecimento de que existe um problema, e o problema precisa ser encarado com uma solução de software. Os interessados estão inseguros do que desejam; ou seja, eles não são capazes de descrever os requisitos de software em qualquer detalhe.

Solução. Uma descrição do processo de prototipação é aqui apresentada. Veja o Capítulo 3 para detalhes.

Contexto resultante. Um protótipo de software que identifica os requisitos básicos (por exemplo, modos de interação, características computacionais, funções de processamento) é aprovado pelos interessados. A seguir, (1) o protótipo pode evoluir por meio de uma série de incrementos para tornar-se o software de produção, ou (2) o protótipo pode ser descartado e o software de produção é construído usando algum outro padrão de processo.

Padrões relacionados. Os seguintes padrões são relacionados a este padrão: **comunicação-com-o-cliente**; **projeto-iterativo**; **desenvolvimento-iterativo**; **avaliação-pelo-cliente**; **levantamento-de-requisitos**.

Usos conhecidos/exemplos. A prototipação é recomendada quando os requisitos são incertos.

2.5 Avaliação de Processo

PONTO CHAVE

A avaliação tenta entender o estado atual do processo do software com a intenção de aperfeiçoá-lo.

? Quais técnicas formais estão disponíveis para avaliar o processo de software?

A existência de um processo de software não é garantia de que o software será entregue no prazo, de que ele satisfaça às necessidades do cliente, ou de que ele exiba as características técnicas que levarão a características de qualidade no longo prazo (Capítulo 26). Padrões de processo precisam ser acoplados à sólida prática da engenharia de software (Parte 2 deste livro). Além disso, o processo em si deve ser avaliado para garantir que ele satisfaça a um conjunto de critérios básicos de processo que demonstraram ser essenciais para uma engenharia de software bem-sucedida.[8] O relacionamento entre o processo de software e os métodos aplicados para avaliação e melhoria é mostrado na Figura 2.5. Várias e diferentes abordagens de *avaliação de processo de software* têm sido propostas durante as últimas décadas.

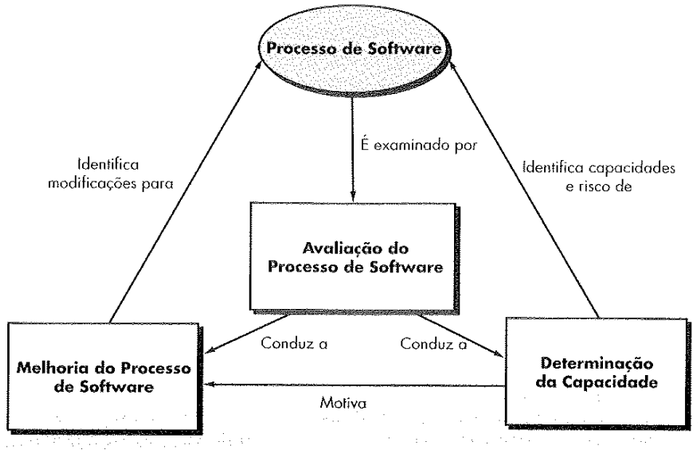
O **SCAMPI (Standard CMMI Assessment Method for Process Improvement, Método de Avaliação Normalizado do CMMI para Aperfeiçoamento de Processo)** fornece um modelo de processo de avaliação com cinco passos que incorpora a iniciação, o diagnóstico, o estabelecimento, a ação e o aprendizado. O método SCAMPI usa o CMMI da SEI (Seção 2.3) como base para a avaliação [SEI00].

A **CBA IPI (CMM-Based Appraisal for Internal Process Improvement, Avaliação do Processo de Aperfeiçoamento Interno Baseado no CMM)** fornece uma técnica de diagnóstico para avaliar a maturidade relativa de uma organização de software usando o CMM da SEI (Precursor do CMMI, discutido na Seção 2.3) como base para a avaliação [DUN01].

8 O CMMI da SEI [CMM02] descreve as características de um processo de software e os critérios para um processo bem-sucedido em volumosos detalhes.

Document shared on www.docsity.com

FIGURA 2.5



A norma SPICE (ISO/IEC 15504) define um conjunto de requisitos para avaliação de processo de software. A intenção da norma é assistir às organizações no desenvolvimento de uma avaliação objetiva da eficácia de qualquer processo definido de software [SPI99].

A **ISO 9001:2000 para Software** é uma norma genérica que se aplica a qualquer organização que deseje aperfeiçoar a qualidade global dos produtos, sistemas ou serviços que ela fornece. Assim, a norma é diretamente aplicável a organizações e empresas de software.

Como a ISO 9001:2000 é amplamente usada em escala internacional, vamos comentá-la brevemente no parágrafo que se segue.

> "Organizações de software têm mostrado deficiências significativas na sua habilidade de capitalizar sobre a experiência ganha com projetos concluídos."
>
> **NASA**

A ISO (International Organization for Standardization) desenvolveu a norma ISO 9001:2000 [ISO00] para definir os requisitos de um sistema de gestão de qualidade (Capítulo 26) que servirá para produzir produtos de alta qualidade e, conseqüentemente, aumentar a satisfação do cliente.[9]

A estratégia subjacente sugerida pela ISO 9001:2000 é descrita da seguinte maneira [ISO01]:

> A ISO 9001:2000 enfatiza a importância de uma organização identificar, implementar, gerir e aperfeiçoar continuamente a efetividade dos processos necessários ao sistema de gestão de qualidade e gerir as interações desses processos a fim de alcançar os objetivos da organização . . .

Veja na Web

Um excelente resumo da ISO 9001:2000 pode ser encontrado em http://praxion.com/iso9001.htm.

A ISO 9001:2000 adotou um ciclo "planejar-fazer-verificar-agir" que é aplicado aos elementos de gestão de qualidade de um projeto de software. Dentro do contexto de software, "planejar" estabelece os objetivos do processo, suas atividades e as tarefas necessárias para a obtenção de softwares de alta qualidade e a resultante satisfação do cliente. "Fazer" implementa o processo de software (incluindo as atividades de arcabouço e guarda-chuva). "Verificar" monitora e mensura o processo a fim de garantir que todos os requisitos estabelecidos para a gestão de qualidade tenham sido atingidos. "Agir" inicia as atividades de aperfeiçoamento do processo de software que trabalham continuamente para aperfeiçoar o processo.

Para uma discussão detalhada, os leitores interessados devem consultar as próprias normas ISO ou [CIA01], [KET01], [MON01] para informações abrangentes.

9 A garantia de qualidade de software (SQA, Software Quality Assurance), um importante elemento da gestão de qualidade, foi definida como uma atividade guarda-chuva que é aplicada ao longo de todo o arcabouço do processo. Ela será discutida em detalhes no Capítulo 26.

2.6 Modelos de Processo Pessoal e de Equipe

O melhor processo de software é aquele que se aproxima do pessoal que fará o serviço. Se um modelo de processo de software tiver sido desenvolvido no nível da empresa ou organização, ele pode ser efetivo apenas se for suscetível a adaptações significativas de modo a satisfazer às necessidades da equipe de projeto que está realmente fazendo o trabalho de engenharia de software. Numa instalação ideal, cada engenheiro de software criaria um processo que melhor satisfizesse às suas necessidades e, ao mesmo tempo, às necessidades mais globais da equipe e da organização. Alternativamente, a própria equipe poderia criar o seu próprio processo e, ao mesmo tempo, satisfazer às necessidades mais restritas dos indivíduos e às necessidades mais amplas da organização. Watts Humphrey ([HUM97] e [HUM00]) argumenta que é possível criar um "processo pessoal de software" e/ou um "processo de equipe de software". Ambos requerem trabalho árduo, treinamento e coordenação, mas ambos são passíveis de obtenção.[10]

> "Uma pessoa que é bem-sucedida simplesmente criou o hábito de fazer coisas que as pessoas malsucedidas não vão fazer."
>
> **Dexter Yager**

Veja na Web

Uma grande variedade de recursos para PSP pode ser encontrada em www.ipd.uka.de/PSP/.

2.6.1. PSP (Personal Software Process — Processo Pessoal de Software)

Todo desenvolvedor usa algum processo para construir softwares de computador. O processo pode ser aleatório ou *ad hoc*, pode modificar-se diariamente, pode não ser eficiente, efetivo ou, mesmo, bem-sucedido, mas existe um processo. Watts Humphrey [HUM97] sugere que para modificar um processo pessoal não efetivo, o indivíduo precisa passar por quatro fases, cada uma exigindo treinamento e cuidadosa instrumentação. O PSP (*processo pessoal de software*) enfatiza a medição pessoal tanto do produto do trabalho que é produzido quanto da qualidade resultante do produto do trabalho. Além disso, o PSP torna o profissional responsável pelo planejamento do projeto (por exemplo, fazer orçamentos e cronogramas) e dá poder ao profissional para controlar a qualidade de todos os produtos do trabalho de software que são desenvolvidos.

O modelo de processo PSP define cinco atividades de arcabouço: planejamento, projeto de alto nível, revisão de projeto de alto nível, desenvolvimento e pós-conclusão.

? Quais atividades de arcabouço são usadas durante o PSP?

Planejamento. Essa atividade isola os requisitos e, com base neles, desenvolve estimativas tanto de tamanho quanto de recursos. Além disso, é feita uma estimativa de defeitos (o número de defeitos projetado para o trabalho). Toda a métrica é registrada em planilhas ou gabaritos. Finalmente, as tarefas de desenvolvimento são identificadas e um cronograma de projeto é criado.

Projeto de alto nível. São desenvolvidas especificações externas para cada componente a ser construído e é criado um projeto dos componentes. Protótipos são construídos quando existe incerteza. Todos os itens são registrados e monitorados.

Revisão do projeto de alto nível. Métodos de verificação formal (Capítulo 26) são aplicados para descobrir erros no projeto. A métrica é mantida para todas as tarefas e resultados importantes do trabalho.

Desenvolvimento. O projeto em nível de componente é refinado e revisado. O código é gerado, revisado, compilado e testado. A métrica é mantida para todas as tarefas e resultados importantes do trabalho.

PONTO CHAVE

PSP enfatiza a necessidade de registrar e analisar os tipos de erros que você faz, de modo que você possa desenvolver estratégias para eliminá-los.

Pós-conclusão. Usando as medidas e a métrica coletadas (uma quantidade substancial de dados que devem ser analisados estatisticamente), a efetividade do processo é determinada. As medidas e a métrica devem fornecer diretrizes para a modificação do processo de modo a aperfeiçoar sua efetividade.

O PSP enfatiza a necessidade de cada engenheiro de software identificar logo os erros e, igualmente importante, entender os tipos de erro que ele tende a fazer. Isso é conseguido por meio de uma atividade de avaliação rigorosa desenvolvida em todos os produtos de trabalho produzidos pelo engenheiro de software.

10 É importante observar que os proponentes do desenvolvimento ágil de software (Capítulo 4) também argumentam que o processo deve ficar perto da equipe. Eles propõem um método alternativo para conseguir isso.

Document shared on www.docsity.com

O PSP representa uma abordagem disciplinada, baseada na métrica, da engenharia de software, que pode levar a um choque de culturas em muitos profissionais. No entanto, quando o PSP é adequadamente apresentado aos engenheiros de software [HUM96], o aperfeiçoamento resultante na produtividade da engenharia de software e na qualidade do software é significativo [FER97]. No entanto, o PSP não tem sido amplamente adotado pela indústria. As razões, infelizmente, têm mais a ver com a natureza humana e a inércia organizacional do que com os pontos fortes e fracos da abordagem PSP. O PSP é intelectualmente desafiador e exige um nível de comprometimento (dos profissionais e de seus gerentes) que nem sempre é possível de ser obtido. O treinamento é relativamente demorado e os custos de treinamento são altos. O nível de medição exigido é culturalmente difícil para muitos entre o pessoal de software.

O PSP pode ser usado como processo efetivo de software em nível pessoal? A resposta é um inequívoco sim. Mas, mesmo que o PSP não seja adotado na totalidade, vale a pena aprender muitos dos conceitos de aperfeiçoamento do processo pessoal que ele introduz.

2.6.2 TSP (Team Process Software — Processo de Equipe de Software)

Veja na Web

Informações sobre a construção de equipes de alto desempenho usando TSP e PSP podem ser obtidas em www.sei.cmu.edu/tsp/.

Como muitos projetos de software de nível industrial são desenvolvidos por uma equipe de profissionais, Watts Humphrey estendeu as lições aprendidas com a introdução do PSP e propôs um TSP (*processo de equipe de software*). O objetivo do TSP é construir uma equipe de projeto "autodirigida" que se organize para produzir softwares de alta qualidade. Humphrey [HUM98] define os seguintes objetivos para o TSP:

- Construir equipes autodirigidas que planejem e monitorem seu trabalho, estabeleçam metas e possuam seus próprios processos e planos. Podem ser equipes essencialmente de software ou equipes de produto integrado (IPT — Integrated Product Team) de três a aproximadamente 20 engenheiros.
- Mostrar aos gerentes como acompanhar e motivar suas equipes, e como ajudá-las a manter desempenho de pico.
- Acelerar o aperfeiçoamento do processo de software tornando o comportamento de nível 5 do CMM normal e esperado.
- Fornecer diretrizes de aperfeiçoamento para organizações de alta maturidade.
- Facilitar o ensino universitário das habilidades de equipe de nível industrial.

AVISO

Para formar uma equipe autodirigida, você deve colaborar bem internamente e comunicar-se bem externamente.

Uma equipe autodirigida tem um entendimento consistente de suas metas e objetivos gerais. Ela define papéis e responsabilidades para cada membro da equipe; monitora dados de projeto quantitativos (sobre a produtividade e a qualidade); identifica um processo de equipe apropriado para o projeto e uma estratégia para a implementação do processo; define normas locais aplicáveis ao trabalho de engenharia de software da equipe; avalia continuamente o risco e reage a isso; além de monitorar, gerenciar e relatar o estado do projeto.

> "Encontrar bons jogadores é fácil. Conseguir que eles joguem como uma equipe é outra história."
>
> **Casey Stengel**

PONTO CHAVE

A documentação do TSP define elementos do processo de equipe e atividades que ocorrem dentro do processo.

O TSP define as seguintes atividades de arcabouço: lançamento, projeto de alto nível, implementação, integração e teste, e pós-conclusão. Como suas contrapartidas no PSP (note que a terminologia é um tanto diferente), essas atividades permitem que a equipe planeje, projete e construa softwares de modo disciplinado e, ao mesmo tempo, meça quantitativamente o processo e o produto. A pós-conclusão prepara o cenário para aperfeiçoamentos do processo.

O TSP usa uma grande variedade de documentos, formulários e normas que servem para guiar os membros da equipe no trabalho deles. Os *documentos* definem as atividades específicas do processo (isto é, lançamento, projeto, implementação, integração e teste, e pós-conclusão do projeto) e outras funções de trabalho mais detalhadas (por exemplo, planejamento do desenvolvimento, desenvolvimento de requisitos, gestão de configuração de software e testes unitários) que fazem parte do processo de equipe. Para ilustrar, considere a atividade inicial de processo — *lançamento do projeto*.

Veja na Web

Informações sobre o quadro de avisos do processo de software — uma ferramenta de apoio ao PSP e ao TSP — podem ser encontradas em processdash.sourceforge.net.

Cada projeto é "lançado" usando-se uma seqüência de tarefas (definidas como um documento) que permite à equipe estabelecer uma base sólida para iniciar o projeto. Recomenda-se o seguinte *documento de lançamento* (apenas o esboço) [HUM00]:

- Revise os objetivos do projeto com a gerência, entre em acordo e documente os objetivos da equipe.
- Estabeleça os papéis da equipe.
- Defina o processo de desenvolvimento da equipe.
- Faça um planejamento de qualidade e estabeleça metas de qualidade.
- Planeje as instalações de apoio necessárias.
- Produza uma estratégia global de desenvolvimento.
- Faça um plano de desenvolvimento para todo o projeto.
- Faça planos detalhados para cada engenheiro na fase seguinte.
- Integre os planos individuais num plano de equipe.
- Refaça o balanceamento da carga de trabalho da equipe para obter um cronograma mínimo geral.
- Avalie os riscos de projeto e atribua responsabilidades de monitoramento para cada risco-chave.

Deve-se notar que a atividade de lançamento pode ser aplicada antes de cada atividade de arcabouço TSP mencionada anteriormente. Isso acomoda a natureza iterativa de muitos projetos e permite à equipe adaptar-se às necessidades mutantes do cliente e às lições aprendidas nas atividades anteriores.

O TSP reconhece que as melhores equipes de software são autodirigidas. Os membros da equipe estabelecem os objetivos do projeto, adaptam o processo para satisfazer às suas necessidades, têm controle sobre o cronograma e, por meio da medição e análise da métrica coletada, trabalham continuamente para melhorar a abordagem da equipe do ponto de vista da engenharia de software.

Como o PSP, o TSP é uma abordagem rigorosa da engenharia de software que fornece benefícios distintos e quantificáveis em produtividade e qualidade. A equipe deve estabelecer um total comprometimento com o processo e deve passar por treinamento para garantir que a abordagem seja propriamente aplicada.

2.7 Tecnologia de Processo

Os modelos de processo genéricos discutidos nas seções precedentes devem ser adaptados para o uso de uma equipe de projeto de software. Para conseguir isso, foram desenvolvidas *ferramentas de tecnologia de processo* a fim de ajudar as organizações a analisar seu processo atual, organizar tarefas de trabalho, controlar e monitorar o progresso e gerenciar a qualidade técnica [NEG99].

As ferramentas de tecnologia de processo permitem a uma organização de software construir um modelo automatizado do arcabouço comum de processo, dos conjuntos de tarefas e das atividades guarda-chuva discutidas na Seção 2.2. O modelo, normalmente representado por uma rede, pode então ser analisado para determinar o fluxo típico de trabalho e examinar estruturas de processo alternativas que possam reduzir o tempo de desenvolvimento ou o custo.

Uma vez criado um processo aceitável, outras ferramentas de tecnologia de processo podem ser usadas para atribuir, monitorar e até mesmo controlar todas as tarefas de engenharia de software definidas como parte do modelo de processo. Cada membro de uma equipe de software pode usar

Document shared on www.docsity.com

tais ferramentas para desenvolver um *checklist* das tarefas de trabalho a ser desenvolvidas, dos produtos de trabalho a ser produzidos e das atividades de garantia de qualidade a ser conduzidas. A ferramenta de tecnologia de processo também pode ser usada para coordenar o uso de outras ferramentas de engenharia de software apoiadas por computador que sejam adequadas a uma tarefa de trabalho específica.

FERRAMENTAS DE SOFTWARE

Ferramentas de Modelagem de Processo

Objetivo: Se uma organização trabalha para aperfeiçoar um processo de negócio (ou software), ela deve primeiramente entendê-lo. As ferramentas de modelagem de processo (também chamadas de *ferramentas de tecnologia de processo ou gestão de processo*) são usadas para representar os elementos-chave de um processo de modo que ele possa ser melhor entendido. Tais ferramentas podem também fornecer links para descrições do processo que ajudem os que estão envolvidos no processo a entender as ações e tarefas de trabalho necessárias para realizá-lo. Ferramentas de modelagem de processo fornecem links para outras ferramentas que fornecem apoio a atividades definidas do processo.

Mecânica: As ferramentas dessa categoria permitem a uma equipe definir os elementos de um modelo de processo específico (ações, tarefas, produtos de trabalho, pontos de garantia de qualidade), fornecer diretrizes detalhadas sobre o conteúdo ou a descrição de cada elemento do processo e, depois, gerir o projeto à medida que ele é conduzido. Em alguns casos, as ferramentas de tecnologia de processo incorporam tarefas-padrão de gestão de projeto tais como orçar, montar cronogramas, monitorar e controlar.

Ferramentas Representativas: [11]

As ferramentas *Igrafx Process Tools*, distribuídas pela Corel Corporation (www.igrafx.com/products/process), são um conjunto de ferramentas que permitem a uma equipe mapear, medir e modelar o processo de software.

O *Objexis Team Portal*, desenvolvido pela Objexis Corporation (www.objexis.com), fornece definição e controle para todo o fluxo de trabalho do processo.

2.8 Produto e Processo

Se o processo for fraco, o produto final sofrerá inevitavelmente, mas uma ênfase exagerada no processo também é perigosa. Em um ensaio resumido, Margaret Davis [DAV95] comenta a dualidade do produto e do processo:

> A cada dez anos, avançando ou recuando cinco, a comunidade de software redefine "o problema" mudando o foco de aspectos do produto para aspectos do processo. Assim, adotamos linguagens de programação estruturada (produto), seguidas de métodos de análise estruturada (processo) e, depois, do encapsulamento dos dados (produto), da ênfase atual no Modelo de Maturidade da Capacidade de Desenvolvimento de Software, do SEI (Software Engineering Institute) (processo) [seguido por métodos orientados a objetos e desenvolvimento ágil de softwares].

Enquanto um pêndulo tende a ficar naturalmente em repouso no ponto intermediário entre dois extremos, o foco da comunidade de software desloca-se constantemente, pois uma nova força é aplicada quando a última oscilação falha. Essas oscilações são prejudiciais em si e por si, porque confundem o profissional de software médio modificando radicalmente o significado de realizar o trabalho, ou de realizá-lo bem. As oscilações também não resolvem "o problema", pois estão fadadas a falhar, enquanto produto e processo forem tratados como uma dicotomia em vez de uma dualidade.

Há precedência na comunidade científica para sugerir noções de dualidade quando contradições nas observações não podem ser totalmente explicadas por uma ou outra teoria competitiva. A natureza dual da luz, que parece ser simultaneamente partícula e onda, tem sido aceita desde 1920,

11 As ferramentas indicadas aqui não representam uma recomendação, mas sim uma exemplificação de ferramentas nessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas de seus respectivos desenvolvedores.

quando Louis de Broglie a propôs. Acredito que as observações que podemos fazer das ferramentas de software e do seu desenvolvimento demonstram uma dualidade fundamental entre produto e processo. Não se pode deduzir ou entender toda a ferramenta, seu contexto, uso, significado e valor se a virmos somente como um processo ou somente como um produto...

Toda atividade humana pode ser um processo, mas cada um de nós obtém um senso de auto-valor daquelas atividades que resultam numa representação ou instância, que pode ser usada ou apreciada por mais de uma pessoa, utilizada repetidamente ou empregada em algum outro contexto não considerado. Isto é, obtemos um sentimento de satisfação pelo reuso de nossos produtos, por nós mesmos ou pelos outros.

Assim, enquanto a rápida assimilação de metas de reutilização no desenvolvimento de software aumenta potencialmente a satisfação dos profissionais de software com seu trabalho, ela aumenta também a urgência da aceitação da dualidade entre produto e processo. Pensando em um artefato reutilizável somente como um produto ou somente como um processo, ou obscurece o contexto e os modos de usá-lo, ou obscurece o fato de que cada uso resulta num produto que, por sua vez, será usado como entrada para alguma outra atividade de desenvolvimento de software. Adotar uma visão ao invés da outra reduz substancialmente as oportunidades de reutilização e, conseqüentemente, as oportunidades de satisfação crescente com o trabalho.

> "Sem dúvida, o sistema ideal, se fosse possível de obtenção, seria um código ao mesmo tempo tão flexível e tão pequeno, que seria capaz de fornecer previamente, para cada situação concebível, uma regra própria e adequada. Mas a vida é complexa demais para trazer o alcance dessa idéia para o âmbito do poder humano."
>
> **Benjamin Cardozo**

As pessoas obtêm tanta (ou mais) satisfação de um processo criativo quanto de um produto final. Um artista aprecia tanto as pinceladas quanto o resultado emoldurado. Um escritor aprecia tanto a busca de uma metáfora adequada quanto o livro pronto. Um profissional de software criativo também deveria obter tanta satisfação do processo quanto do produto final.

O trabalho do pessoal de software vai se modificar nos próximos anos. A dualidade entre produto e processo é um elemento importante para manter pessoas criativas engajadas enquanto a transição da programação para a engenharia de software é concluída.

2.9 Resumo

A engenharia de software é uma disciplina que integra processo, métodos e ferramentas para o desenvolvimento de softwares de computador. Foram propostos vários e diferentes modelos de processo para a engenharía de software, mas todos definem um conjunto de atividades de arcabouço, uma coleção de tarefas que são conduzidas para realizar cada atividade, produtos de trabalho produzidos como conseqüência das tarefas, e um conjunto de atividades guarda-chuva que se espalham por todo o processo. Padrões de processo podem ser usados para definir as características de um processo.

O CMMI (Capability Maturity Model Integration) é um metamodelo de processo abrangente que descreve as metas, práticas e capacidades específicas que devem estar presentes num processo de software maduro. A SPICE e outras normas definem os requisitos para conduzir uma avaliação do processo de software, e a norma ISO 9001:2000 examina a gestão de qualidade dentro de um processo.

Modelos pessoais e de equipe para o processo de software têm sido propostos. Ambos enfatizam a medição, planejamento e autodireção como ingredientes-chave para um processo de software bem-sucedido.

Os princípios, conceitos e métodos que nos permitem executar o processo que chamamos de *engenharia de software* são considerados ao longo deste livro.

Document shared on www.docsity.com

Referências Bibliográficas

[AMB98] Ambler, S., *Process Patterns: Building Large-Scale Systems Using Object Technology*, Cambridge, University Press/SIGS Books, 1998.

[BAE98] Baetjer Jr., H., *Software as Capital*, IEEE Computer Society Press, 1998, p. 85.

[CIA01] Cianfrani, C., *et al.*, *ISO 9001:2000 Explained*, American Society of Quality, 2001.

[CMM02] *Capability Maturity Model Integration (CMMI)*, versão 1.1, Software Engineering Institute, mar. 2002, http://www.sei.cmu.edu/cmmi/.

[DAV95] Davis, M.J., "Process and Product: Dichotomy or Duality", *Software Engineering Notes*, ACM Press, v. 20, n. 2, abr. 1995, p. 17-18.

[DUN01] Dunaway, D., e Masters, S., *CMM-Based Appraisal for Internal Process Improvement (CBA IPI) Version 1.2 Method Description*, Software Engineering Institute, 2001, http://www.sei.cmu.edu/publications/documents/01.reports/01tr033.html.

[ELE98] El Eman, K., Drouin, J., e Melo, W. (eds.), *SPICE: The Theory and Practices of Software Process Improvement and Capability Determination*, IEEE Computer Society Press, 1998.

[FER97] Ferguson, P., *et al.*, "Results of applying the personal software process". *IEEE* Computer, v. 30, n. 5, maio 1997, p. 24-31.

[HUM96] Humphrey, W., "Using a Defined and Measured Personal Software Process", *IEEE Software*, v. 13, n. 3, maio/jun. 1996, p. 77-88.

[HUM97] Humphrey, W., *Introduction to the Personal Software Process*, Addison-Wesley, 1997.

[HUM98] _____, "The Three Dimensions of Process Improvement, Part III: The Team Process", *Crosstalk*, abr. 1998, http://www.stsc.hill.af.mil/crosstalk/1998/apr/dimensions.asp.

[HUM00] _____, *Introduction to the Team Software Process*, Addison-Wesley, 2000.

[IEE93] *IEEE Standards Collection: Software Engineering*, IEEE Standard 610.12-1990, IEEE, 1993.

[ISO00] *ISO 9001:2000 Document Set*, International Organization for Standards, 2000, http://www.iso.ch/iso/en/iso9000-14000/iso9000/iso9000index.html.

[ISO01] "Guidance on the Process Approach to Quality Management Systems", Document ISO/TC 176/SC2/N544R, International Organization for Standards, maio 2001.

[KET01] Ketola, J., e Roberts, K., *ISO 9001:2000 in a Nutshell*, 2ª ed., Paton Press, 2001.

[MON01] Monnich, H., Jr., Monnich, H., ISO 9001:2000 for Small-and-Medium-Sized Businesses, American Society of Quality, 2001.

[NAU69] Naur, P. e Randall, B., (eds.), *Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee*, NATO, 1969.

[NEG99] Negele, H., "Modeling of Integrated Product Development Processes", *Proc. 9th Annual Syposium of INCOSE*, Reino Unido, 1999.

[PAU93] Paulk, M. *et al.*, *Capability Maturity Model for Software*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1993.

[PHI02] Phillips, M. "CMMI V1.1 Tutorial", abr. 2002, http://www.sei.cmu.edu/cmmi/.

[SEI00] *SCAMPI, V1.0 Standard CMMI® Assessment Method for Process Improvement: Method Description*, Software Engineering Institute, Technical Report CMU/SEI-2000-TR-009, http://www.sei.cmu.edu/publications/documents/00.reports/00tr009.html.

[SPI99] "SPICE: Software Process Assessment, Part1: Concepts and Introduction", versão 1.0, ISO/IEC JTC1, 1999.

Problemas e Pontos a Considerar

2.1. Na introdução deste capítulo, Baetjer observa: "O processo fornece interação entre usuários e projetistas, entre usuários e ferramentas em desenvolvimento e entre projetistas e ferramentas em desenvolvimento [tecnologia]". Relacione cinco questões que (a) projetistas deveriam fazer a usuários; (b) usuários deveriam fazer a projetistas; (c) usuários deveriam fazer a si mesmos sobre o produto de software que deve ser construído e (d) projetistas deveriam fazer a si mesmos sobre o produto de software que deve ser construído e sobre o processo que será usado para construí-lo.

2.2. A Figura 2.1 coloca as três camadas de engenharia de software sobre uma camada denominada "foco na qualidade". Isso implica um programa de qualidade para toda a organização tal como de Gestão de Qualidade Total. Pesquise e faça um esboço dos pontos-chave de um programa de Gestão de Qualidade Total.

2.3. Existe algum caso em que as atividades genéricas do processo de engenharia de software não se aplicam? Em caso afirmativo, descreva-o.

2.4. Atividades guarda-chuva ocorrem ao longo do processo de software. Você acha que elas são aplicadas uniformemente em todo o processo ou algumas são concentradas em uma ou mais atividades de arcabouço?

2.5. Descreva um arcabouço de processo com suas próprias palavras. Quando se diz que as atividades de arcabouço são aplicadas a todos os projetos, isso quer dizer que as mesmas tarefas de trabalho são aplicadas a todos os projetos, independentemente do tamanho e da complexidade? Explique.

2.6. Tente desenvolver um conjunto de tarefas para a atividade de *comunicação*.

2.7. Pesquise o CMMI um pouco mais detalhadamente e discuta os prós e contras dos modelos CMMI contínuo e por estágios.

2.8. Faça o *download* da documentação do CMMI do site da SEI na Web e selecione uma área de processo diferente do planejamento de projeto. Faça uma lista de metas específicas (ME) e das práticas específicas (PE) associadas definidas para a área que você escolheu.

2.9. Considere a atividade de arcabouço *comunicação*. Desenvolva um padrão de processo completo (que seria um padrão de estágio) usando o gabarito apresentado na Seção 2.4.

2.10. Qual a finalidade da avaliação de processo? Por que a SPICE foi desenvolvida como norma de avaliação de processo?

2.11. Faça alguma pesquisa sobre PSP e entregue uma apresentação que indique os benefícios quantitativos do processo.

2.12. O uso de "documentos" (mecanismo necessário no TSP) não é universalmente aceito dentro da comunidade de software. Faça uma lista dos prós e dos contras relativos aos "documentos" e sugira ao menos duas situações em que eles seriam úteis, e outras duas situações em que eles poderiam fornecer menos benefícios.

Leituras e Fontes de Informação Adicionais

O estado da arte atual em engenharia de software e o processo de engenharia de software podem ser melhor acompanhados por publicações mensais como *IEEE Software*, *Computer* e *IEEE Transactions on Software Engineering*. Periódicos da indústria como o *Application Development Trends* e o *Cutter IT Journal*, freqüentemente contêm artigos sobre tópicos de engenharia de software. A disciplina é "resumida" todo ano nos anais da *Proceedings of the International Conference on Software Engineering*, patrocinada pelo IEEE e pelo ACM, e é discutida em profundidade em revistas como *ACM Transactions on Software Engineering and Methodology*; *ACM Software Engineering Notes*, e *Annals of Software Engineering*. Milhares de páginas da Web são dedicadas à engenharia e ao processo de software.

Muitos livros dedicados ao processo de software e à engenharia de software foram publicados nos últimos anos. Alguns apresentam um panorama de todo o processo, enquanto outros se concentram em alguns tópicos importantes, em detrimento de outros. Entre as ofertas mais populares (além deste livro!) estão:

Abran, A., e Moore, J., *SWEBOK: Guide to the Software Engineering Body of Knowlwdge*, IEEE, 2002.
Ahern, D. *et al.*, *CMMI Distilled*, Addison-Wesley, 2001.
Chrisis, B. *et al.*, *CMMI: Guidelines for Process Integration and Product Improvement*, Addison-Wesley, 2003.
Christensen, M., e Thayer, R., *A Project Manager´s Guide to Software Engineering Best Practices*, IEEE-CS Press (Wiley), 2002.
Glass, R., *Fact and Fallacies of Software Engineering, Addison-Wesley*, 2002.
Hunter, R., e Thayer, R., (eds.), *Software Process Improvement*, IEEE-CS Press (Wiley), 2001.
Persse, J., *Implementing the Capability Maturity Model*, Wiley, 2001.
Pflegeer, S., *Software Engineering: Theory and Practice*, 2ª ed., Prentice-Hall, 2001.
Potter, N. e Sakry, M., *Making Process Improvement Work*, Addison-Wesley, 2002.
Sommerville, I., *Software Engineering*, 6ª ed., Addison-Wesley, 2000.

No lado mais leve, um livro de Robert Glass (*Software Conflict*, Yourdon Press, 1991) apresenta ensaios divertidos e controversos sobre softwares e o processo de engenharia de software. Yourdon (*Death March Projects*, Prentice-Hall, 1997) discute o que vai mal quando a maioria dos projetos de software falha e como evitar esses erros.

Garmus (*Measuring the Software Process*, Prentice-Hall 1995) e Florac e Carlton (*Measuring the Software Process*, Addison-Wesley, 1999) discutem o uso de medidas como meio de avaliar estatisticamente a eficácia de qualquer processo de software.

Document shared on www.docsity.com

Uma grande variedade de normas e procedimentos sobre engenharia de software foi publicada na última década. O *IEEE Software Engineering Standards* contém muitas normas diferentes que cobrem quase todos os aspectos importantes da tecnologia. O conjunto de documentos ISO 9001:2000 fornece diretrizes para as organizações de software que desejam melhorar suas atividades de gestão de qualidade. Outras normas de engenharia de software podem ser obtidas com o Departamento de Defesa, a FAA e outras agências norte-americanas governamentais e não lucrativas. Fairclough (*Software Engineering Guides*, Prentice-Hall, 1996) fornece referências detalhadas das normas de engenharia de software produzidas pela ESA (European Space Agency).

Uma grande variedade de fontes de informação sobre engenharia de software e sobre o processo de software está disponível na Internet. Uma lista atual de referências da World Wide Web relevantes para o processo de software pode ser encontrada na página deste livro: **http://www.mhhe.com/pressman.**

Modelos Prescritivos de Processo

CAPÍTULO 3

Conceitos-Chave

desenvolvimento concorrente 46
métodos formais 49
modelo de desenvolvimento baseado em componentes .. 48
modelo DSOA 51
modelo em cascata 38
modelo espiral 44
modelo RAD 41
modelos prescritivos 38
processo evolucionário 42
processo incremental 39
Processo Unificado 51
prototipagem 42

Os modelos prescritivos de processo foram originalmente propostos para colocar ordem no caos do desenvolvimento de software. A história tem indicado que esses modelos convencionais têm trazido uma certa dose de estrutura útil para o trabalho de engenharia de software e têm fornecido um roteiro razoavelmente efetivo para as equipes de software. No entanto, o trabalho de engenharia de software e o produto que ele produz permanecem no "limite do caos" [NOG00].

Em um trabalho interessante sobre o estranho relacionamento entre ordem e caos no mundo do software, Nogueira e seus colegas [NOG00] afirmam:

> O limite do caos é definido como "um estado natural entre ordem e caos, um amplo compromisso entre estrutura e surpresa" [KAU95]. O limite do caos pode ser visualizado como um estado instável, parcialmente estruturado ... É instável porque é constantemente atraído para o caos ou para a ordem absoluta.
>
> Temos a tendência de pensar que ordem é o estado ideal da natureza. Isso pode ser um erro. A pesquisa... apóia a teoria de que operação fora do equilíbrio gera criatividade, processos auto-organizados e crescentes retornos [ROO96]. Ordem absoluta significa a ausência de variabilidade, o que poderia ser uma vantagem em ambientes imprevisíveis. A modificação ocorre quando existe alguma estrutura tal que a modificação pode ser organizada, mas não tão rígida que não possa ocorrer. Caos em excesso, por outro lado, pode tornar impossível a coordenação e coerência. A falta de estrutura nem sempre significa desordem.

Panorama

O que é? Modelos prescritivos de processo definem um conjunto distinto de atividades, ações, tarefas, marcos e produtos de trabalho que são necessários para fazer engenharia de software com alta qualidade. Esses modelos de processo não são perfeitos, mas efetivamente fornecem um roteiro útil para o trabalho de engenharia de software.

Quem faz? Engenheiros de software e seus gerentes adaptam um modelo de processo prescritivo a suas necessidades e depois o seguem. Além disso, o pessoal que solicitou o software tem um papel a desenvolver à medida que o modelo de processo é seguido.

Por que é importante? Porque fornece estabilidade, controle e organização a uma atividade que pode, se deixada sem controle, tornar-se bastante caótica. Alguns têm-se referido a modelos prescritivos de processo como "modelos de processo rigorosos", porque eles freqüentemente incluem as capacitações sugeridas pelo CMMI (Capítulo 2). No entanto, cada modelo de processo deve ser adaptado para que seja usado efetivamente em um projeto de software específico.

Quais são os passos? O processo dirige uma equipe de software por meio de um arcabouço de atividades guarda-chuva que são organizadas em um fluxo de processo que pode ser linear, incremental ou evolutivo. A terminologia e os detalhes de cada modelo de processo diferem, mas as atividades genéricas de arcabouço permanecem razoavelmente consistentes.

Qual é o produto do trabalho? Do ponto de vista de um engenheiro de software, os produtos do trabalho são os programas, documentos e dados que são produzidos em conseqüência das atividades e tarefas definidas pelo processo.

Como tenho certeza de que fiz corretamente? Existem diversos mecanismos de avaliação de processo de software que permitem às organizações determinar a "maturidade" do seu processo de software. Todavia, a qualidade, pontualidade e viabilidade no longo prazo do produto que você constrói são os melhores indicadores da eficácia do processo usado.

Document shared on www.docsity.com

As implicações filosóficas desse argumento são significativas para a engenharia de software. Se os modelos prescritivos de processo[1] buscam estrutura e ordem, serão apropriados ao mundo do software que busca modificações? No entanto, se rejeitarmos modelos de processo convencionais (e a ordem que eles implicam) e os substituirmos por algo menos estruturado, tornamos impossível obter coordenação e coerência no mundo do software?

Não há respostas fáceis para essas questões, mas existem alternativas disponíveis para os engenheiros de software. Neste capítulo examinamos a abordagem de processo prescritivo na qual ordem e consistência do projeto são tópicos dominantes. No Capítulo 4 examinaremos a abordagem de processo ágil, na qual auto-organização, colaboração, comunicação e adaptabilidade dominam a filosofia do processo.

3.1 Modelos Prescritivos

Ponto chave

Um modelo de processo prescritivo preenche o arcabouço de processo com conjuntos explícitos de tarefas para as ações de engenharia de software.

Toda organização de engenharia de software deveria descrever um conjunto específico de atividades de arcabouço (Capítulo 2) para o(s) processo(s) de software que adota. Deveria preencher cada atividade de arcabouço com um conjunto de ações de engenharia de software e definir cada ação em termos de um conjunto de tarefas que identifique o trabalho (e produtos de trabalho) a ser realizado para satisfazer às metas de desenvolvimento. Deveria, então, adaptar o modelo de processo resultante para acomodar a natureza específica de cada projeto, o pessoal que vai fazer o trabalho e o ambiente no qual o trabalho vai ser conduzido. Independentemente do modelo de processo que seja selecionado, os engenheiros de software têm tradicionalmente escolhido um arcabouço de processo genérico que inclui as seguintes atividades de arcabouço: comunicação, planejamento, modelagem, construção e implantação.

> "Existem muitos modos de ir para a frente, mas apenas um modo de ficar parado."
>
> Franklin D. Roosevelt

Aviso

Mesmo que um processo seja prescritivo, não considere que ele seja estático. Modelos prescritivos devem ser adaptados ao pessoal, ao problema e ao projeto.

Nas próximas seções examinaremos vários modelos prescritivos de processo de software; nós os chamamos de "prescritivos" porque eles prescrevem um conjunto de elementos de processo — atividades de arcabouço, ações de engenharia de software, tarefas, produtos de trabalho, mecanismos de garantia de qualidade e de controle de modificações para cada projeto. Cada modelo de processo também prescreve um *fluxo de trabalho* — isto é, a maneira como os elementos do processo se inter-relacionam uns com os outros.

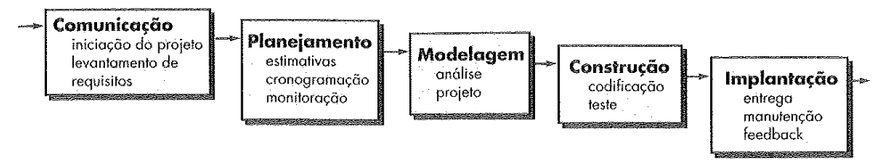
Todos os modelos de processo de software podem acomodar as atividades genéricas de arcabouço que foram descritas no Capítulo 2, mas cada um aplica uma ênfase diferente a essas atividades e define um fluxo de trabalho que invoca cada atividade de arcabouço (bem como ações de engenharia de software e tarefas) de maneira diferente.

3.2 O Modelo em Cascata

Há ocasiões em que os requisitos de um problema são razoavelmente bem compreendidos — quando o trabalho flui da comunicação até a implantação de um modo razoavelmente linear. Essa situação é algumas vezes encontrada quando adaptações bem definidas ou aperfeiçoamentos de um sistema existente precisam ser feitos (por exemplo, uma adaptação de um software de contabilidade que foi determinada devido a modificações na regulamentação governamental). Pode ocorrer também em um número limitado de novos esforços de desenvolvimento, mas apenas quando os requisitos são bem definidos e razoavelmente estáveis.

1 Modelos prescritivos de processo são freqüentemente chamados de modelos de processo "convencionais".

Figura 3.1 O modelo em cascata



O *modelo em cascata*, algumas vezes chamado de *ciclo de vida clássico*, sugere uma abordagem sistemática e seqüencial[2] para o desenvolvimento de softwares que começa com a especificação dos requisitos pelo cliente e progride ao longo do planejamento, modelagem, construção e implantação, culminando na manutenção progressiva do software acabado.

O modelo em cascata é o paradigma mais antigo da engenharia de software. No entanto, nas duas últimas décadas, a crítica a esse modelo de processo tem provocado, mesmo em seus mais ardentes adeptos, questionamentos sobre sua eficácia [HAN95]. Entre os problemas que são algumas vezes encontrados quando o modelo em cascata é aplicado estão:

? Por que o modelo em cascata algumas vezes falha?

1. Projetos reais raramente seguem o fluxo seqüencial que o modelo propõe. Apesar de o modelo linear poder acomodar a iteração, ele o faz indiretamente. Como resultado, as modificações podem causar confusão à medida que a equipe de projeto prossegue.
2. Em geral, é difícil para o cliente estabelecer todos os requisitos explicitamente. O modelo em cascata exige isso e tem dificuldade de acomodar a incerteza natural que existe no começo de muitos projetos.
3. O cliente precisa ter paciência. Uma versão executável do(s) programa(s) não vai ficar disponível até o período final do intervalo de tempo do projeto. Um erro grosseiro pode ser desastroso se não for detectado até que o programa executável seja revisto.

Numa análise interessante de projetos reais, Bradac [BRA94] descobriu que a natureza linear do modelo em cascata leva a "estados de bloqueio" nos quais alguns membros da equipe de projeto precisam esperar que outros membros completem as tarefas dependentes. Na realidade, o tempo gasto em espera pode exceder o tempo gasto no trabalho produtivo! O estado de bloqueio tende a ocorrer mais no início e no fim de um processo seqüencial linear.

Hoje em dia, o trabalho de software é em ritmo rápido e sujeito a uma torrente sem fim de modificações (de características, funções e conteúdo da informação). O modelo em cascata é freqüentemente inadequado para esse tipo de trabalho. No entanto, pode servir como um modelo de processo útil em situações nas quais os requisitos são fixos e o trabalho deve prosseguir até o fim de modo linear.

3.3 Modelos Incrementais de Processo

Há muitas situações em que os requisitos iniciais do software são razoavelmente bem definidos, mas o escopo global do esforço de desenvolvimento elimina um processo puramente linear. Além disso, pode haver uma necessidade compulsiva de fornecer rapidamente um conjunto limitado de funcionalidades do software aos usuários e depois refinar e expandir aquela funcionalidade em versões subseqüentes do software. Em tais casos, um modelo de processo que é destinado a produzir o software em incrementos é escolhido.

2 Apesar do modelo original em cascata proposto por Winston Royce [ROY 70] prever "ciclos de realimentação", a grande maioria das organizações que aplica esse modelo de processo o trata como se ele fosse estritamente linear.

Document shared on www.docsity.com

> "Freqüentemente, o trabalho de software segue a primeira lei do ciclismo: independentemente de onde você está indo é subida e contra o vento."
>
> Autor desconhecido

PONTO CHAVE

O modelo incremental entrega uma série de versões chamadas de *incrementos*, que fornecem progressivamente mais funcionalidade para os clientes à medida que cada incremento é entregue.

AVISO

Se o seu cliente exige entrega em uma data que é impossível de satisfazer, sugira a entrega de um ou mais incrementos naquela data e o restante do software (incrementos adicionais) posteriormente.

3.3.1 O Modelo Incremental

O *modelo incremental* combina elementos do modelo em cascata aplicado de maneira iterativa. Como mostra a Figura 3.2, o modelo incremental aplica seqüências lineares de uma forma racional à medida que o tempo passa. Cada seqüência linear produz "incrementos" do software passíveis de serem entregues [MCD93]. Por exemplo, softwares de processamento de texto desenvolvidos segundo o paradigma incremental poderiam entregar a gestão básica de arquivos, edição e produção de documentos no primeiro incremento; capacidades de edição e de produção de documentos mais sofisticados no segundo incremento; verificação ortográfica e gramatical no terceiro incremento; e capacidade avançada de disposição de página no quarto incremento. Deve-se notar que o fluxo de processo para qualquer incremento pode incorporar o paradigma de prototipagem, discutido na Seção 3.4.1.

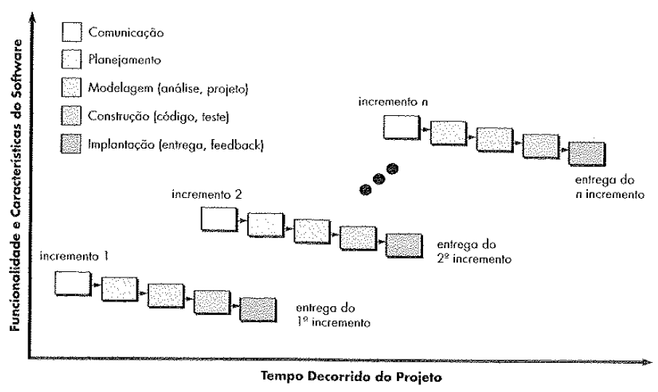
Quando um modelo incremental é usado, o primeiro incremento é freqüentemente chamado de *núcleo do produto*. Isto é, os requisitos básicos são satisfeitos, mas muitas características suplementares (algumas conhecidas, outras desconhecidas) deixam de ser elaboradas. O núcleo do produto é usado pelo cliente (ou passa por revisão detalhada). Um plano é desenvolvido para o próximo incremento como resultado do uso e/ou avaliação. O plano visa à modificação do núcleo do produto para melhor satisfazer às necessidades do cliente e à elaboração de características e funcionalidades adicionais. Esse processo é repetido após a realização de cada incremento, até que o produto completo seja produzido.

O modelo de processo incremental, como a prototipagem e outras abordagens evolucionárias, é iterativo por natureza. Mas, diferentemente da prototipagem, o modelo incremental tem o objetivo de apresentar um produto operacional a cada incremento. Os primeiros incrementos são versões simplificadas do produto final, mas oferecem capacidades que servem ao usuário, além de uma plataforma para a avaliação do usuário.[3]

O desenvolvimento incremental é particularmente útil quando não há mão-de-obra disponível para uma implementação completa, dentro do prazo comercial de entrega estabelecido para o projeto. Os primeiros incrementos podem ser implementados com menos pessoal. Se o núcleo do

FIGURA 3.2

O modelo incremental



3 É importante notar que uma filosofia incremental também é usada por todos os modelos de processo "ágeis" discutidos no Capítulo 4.

produto for bem recebido, então pessoal extra (se necessário) pode ser adicionado para implementar o próximo incremento. Além disso, os incrementos podem ser planejados para gerir os riscos técnicos. Por exemplo, um sistema importante pode exigir a disponibilidade de um hardware novo, que está em desenvolvimento, e cuja data de entrega é incerta. Pode ser possível planejar os primeiros incrementos de modo que seja evitado o uso desse hardware, permitindo, assim, que uma parte da funcionalidade seja entregue aos usuários finais sem demora excessiva.

3.3.2 O Modelo RAD

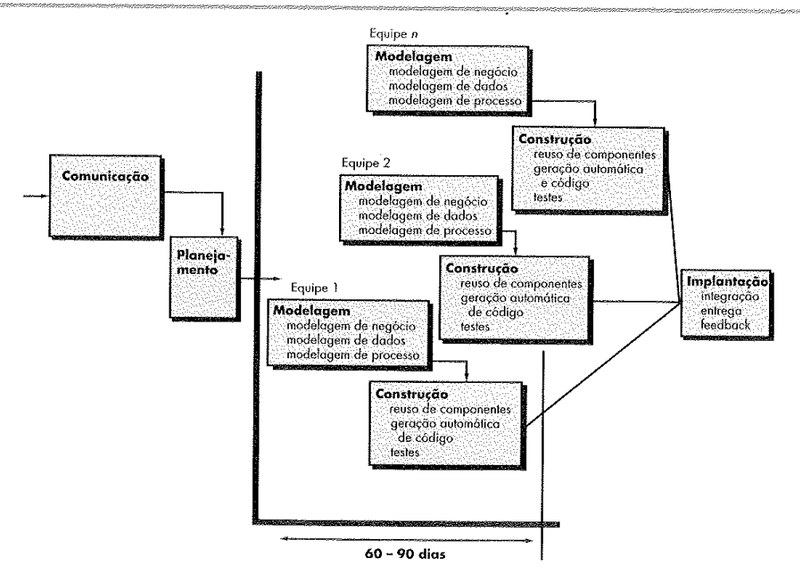
O *RAD* (*Rapid Application Development*, desenvolvimento rápido de aplicação) é um modelo de processo de software incremental que enfatiza um ciclo de desenvolvimento curto. O modelo RAD é uma adaptação "de alta velocidade" do modelo em cascata, no qual o desenvolvimento rápido é conseguido com o uso de uma abordagem de construção baseada em componentes. Se os requisitos forem bem compreendidos e o objetivo do projeto for restrito,[4] o processo RAD permite a uma equipe de desenvolvimento criar um "sistema plenamente funcional", dentro de um período de tempo muito curto (por exemplo, 60 a 90 dias) [MAR91].

Como outros modelos de processo, a abordagem RAD se enquadra nas atividades genéricas de arcabouço apresentadas anteriormente. A *comunicação* trabalha para entender os problemas do negócio e as características de informação que o software precisa acomodar. O *planejamento* é essencial, porque várias equipes de software trabalham em paralelo em diferentes funções do sistema. A *modelagem* abrange três das principais fases — modelagem do negócio, modelagem dos dados e modelagem dos processos — e estabelece representações de projeto que servem como base para a atividade de construção do RAD. A *construção* enfatiza o uso de componentes de software preexistentes e a aplicação da geração automática de código. Finalmente, a *implantação* estabelece a base para iterações subseqüentes, se necessárias [KER94].

O modelo do processo RAD é ilustrado na Figura 3.3. Obviamente, as restrições de tempo impostas em um projeto RAD exigem "âmbito passível de aumento" [KER94]. Se uma aplicação comercial pode ser modularizada de modo a permitir que cada função principal possa ser completada em

FIGURA 3.3

O modelo RAD



4 Essas condições nunca são garantidas. De fato, muitos projetos de software têm requisitos mal definidos no começo. Em tais casos, as abordagens de prototipagem ou evolucionária (Seção 3.4) são opções de processo muito melhores. Veja [REI95].

Document shared on www.docsity.com

menos de três meses (usando a abordagem descrita anteriormente), é uma candidata ao RAD. Cada função principal pode ser tratada por uma equipe RAD distinta e, depois, integrada para formar um todo.

? Quais são as desvantagens do modelo RAD?

Como todos os modelos de processos, a abordagem RAD tem desvantagens [BUT94]: (1) para projetos grandes, mas passíveis de sofrer aumento, o RAD exige recursos humanos suficientes para criar um número adequado de equipes RAD; (2) se desenvolvedores e clientes não estiverem comprometidos com as atividades continuamente rápidas, necessárias para completar o sistema em um curtíssimo espaço de tempo, os projetos RAD falharão; (3) se o sistema não puder ser adequadamente modularizado, a construção dos componentes necessários ao RAD será problemática; (4) se for necessário um alto desempenho e esse desempenho tiver de ser conseguido ajustando as interfaces dos componentes do sistema, a abordagem RAD pode não funcionar; e (5) o RAD pode não ser adequado quando os riscos técnicos são altos (por exemplo, quando uma nova aplicação faz bastante uso de nova tecnologia).

3.4 Modelos Evolucionários de Processo de Software

PONTO CHAVE

Modelos evolucionários de processo produzem uma versão cada vez mais completa do software a cada iteração.

O software, como todo sistema complexo, evolui com o passar do tempo [GIL88]. Os requisitos do negócio e do produto mudam freqüentemente à medida que o desenvolvimento prossegue, dificultando um caminho direto para um produto final; prazos reduzidos de mercado tornam impossível completar um produto de software abrangente, mas uma versão reduzida pode ser elaborada para fazer face à competitividade ou às pressões do negócio; um conjunto de requisitos básicos de um produto ou sistema é bem entendido, mas os detalhes das extensões do produto ou sistema ainda precisam ser definidos. Nesse caso, e em situações semelhantes, os engenheiros de software precisam de um modelo de processo que tenha sido explicitamente projetado para acomodar um produto que evolui com o tempo.

Os *modelos evolucionários* são iterativos. Eles são caracterizados de forma a permitir aos engenheiros de software desenvolver versões cada vez mais completas do software.

3.4.1 Prototipagem

AVISO

Quando seu cliente tiver uma necessidade razoável mas não tiver noção dos detalhes, desenvolva um protótipo como primeiro passo.

O cliente, freqüentemente, define um conjunto de objetivos gerais para o software, mas não identifica detalhadamente requisitos de entrada, processamento ou saída. Em outros casos, o desenvolvedor pode estar inseguro da eficiência de um algoritmo, da adaptabilidade de um sistema operacional ou da forma que a *interação homem/máquina* deve assumir. Nessas, e em muitas outras situações, um *paradigma de prototipagem* pode oferecer a melhor abordagem.

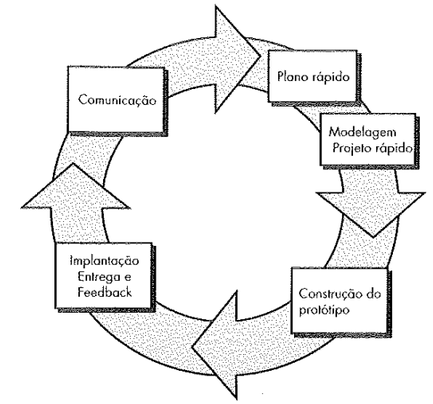
Apesar de a prototipagem poder ser usada como um modelo de processo independente, ela é mais comumente usada como uma técnica que pode ser implementada dentro do contexto de qualquer um dos modelos de processo mencionados neste capítulo. Independentemente da maneira como é aplicado, o paradigma de prototipagem auxilia o engenheiro de software e o cliente a entenderem melhor o que deve ser construído quando os requisitos estão *confusos*.

O paradigma de prototipagem (Figura 3.4) começa com a comunicação. O engenheiro de software e o cliente encontram-se e definem os objetivos gerais do software, identificam as necessidades conhecidas e delineiam áreas que necessitam de mais definições. Uma iteração de prototipagem é planejada rapidamente e a modelagem (na forma de um "projeto rápido") ocorre. O projeto rápido concentra-se na representação daqueles aspectos do software que estarão visíveis para o cliente/usuário (por exemplo, leiaute da interface humana ou formatos de saída de tela). O projeto rápido leva à construção de um protótipo, que é implantado e depois avaliado pelo cliente/usuário. O feedback é usado para refinar os requisitos do software. A iteração ocorre à medida que o protótipo é ajustado para satisfazer às necessidades do cliente, e, ao mesmo tempo, permite ao desenvolvedor entender melhor o que precisa ser feito.

Idealmente, o protótipo serve como um mecanismo para identificação dos requisitos do software. Se um protótipo executável é elaborado, o desenvolvedor tenta usar partes de programas existentes ou aplicar ferramentas (por exemplo, geradores de relatórios, gestores de janelas etc.) que possibilitem programas executáveis serem gerados rapidamente.

FIGURA 3.4

O modelo de prototipagem



Mas o que fazer com o protótipo quando ele tiver cumprido a finalidade descrita? Brooks [BRO75] fornece a resposta:

> Na maioria dos projetos, o primeiro sistema construído consegue ser apenas utilizável. Pode ser muito lento, muito grande, muito complicado de usar, ou tudo isso ao mesmo tempo. Não há outra alternativa senão começar de novo e construir uma versão reprojetada, na qual esses problemas são resolvidos... Quando um novo conceito de sistema ou uma nova tecnologia é usada deve-se construir um sistema para ser descartado, porque nem mesmo o melhor planejamento é tão onisciente para fazer certo na primeira vez. A questão de gerência, entretanto, não é se o sistema piloto elaborado deve ser descartado. Ele será. A única questão é planejar antecipadamente a construção de um descartável, ou prometer entregar o software descartável aos clientes.

O protótipo pode servir como "o primeiro sistema", aquele que Brooks recomenda que se descarte. Mas essa pode ser uma visão idealizada. É verdade que tanto clientes quanto desenvolvedores gostam do paradigma de prototipagem. Os usuários têm o sabor de um sistema real e os desenvolvedores conseguem construir algo imediatamente. Ainda assim, a prototipagem pode ser problemática pelas seguintes razões:

AVISO

Resista à pressão de aperfeiçoar um protótipo malfeito para torná-lo um produto de produção. A qualidade quase sempre sofre com o resultado.

1. O cliente vê o que parece ser uma versão executável do software, ignorando que o protótipo apenas consiga funcionar precariamente ("ele é mantido em pé com goma de mascar e arame"), sem saber que, na pressa de fazê-lo rodar, ninguém considerou a sua qualidade global ou manutenibilidade no longo prazo. Quando informado de que o produto deve ser refeito para que altos níveis de qualidade possam ser atingidos, o cliente reclama e exige "alguns consertos", para transformar o protótipo num produto executável. Em geral, a gerência de desenvolvimento de software concorda.
2. O desenvolvedor freqüentemente faz concessões na implementação a fim de conseguir rapidamente um protótipo executável. Um sistema operacional, ou uma linguagem de programação inapropriada, pode ser usado simplesmente por estar disponível e ser conhecido; um algoritmo ineficiente pode ser implementado simplesmente para demonstrar capacidade. Passado um certo tempo, o desenvolvedor pode ficar familiarizado com essas escolhas e esquecer todas as razões por que elas eram inadequadas. A escolha muito aquém da ideal tornou-se agora parte integral do sistema.

Apesar de problemas poderem ocorrer, a prototipagem pode ser um paradigma efetivo para a engenharia de software. O importante é definir as regras do jogo no início; isto é, cliente e desenvolvedor devem estar de acordo que o protótipo é construído para servir como um mecanismo de definição dos requisitos. Depois ele será descartado (pelo menos em parte), e o software real será submetido à engenharia com um olho na qualidade.

Document shared on www.docsity.com

CASASEGURA

Seleção de um Modelo de Processo, Parte I

A cena: Sala de reuniões do grupo de engenharia de software na empresa CPI, uma empresa (fictícia) que fabrica produtos de consumo para uso residencial e comercial.

Os personagens: Lee Warren, gerente de engenharia; Doug Miller, gerente de engenharia de software, Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software e Ed Robbins, membro da equipe de software.

A conversa:

Lee: Então, vamos recapitular. Eu gastei algum tempo discutindo a linha de produtos *CasaSegura* como a vemos no momento. Sem dúvida, temos muito trabalho a fazer para simplesmente definir a coisa, mas eu gostaria que vocês começassem a pensar como vão abordar a parte do software desse projeto.

Doug: Parece que temos sido bastante desorganizados em nossa abordagem de software no passado.

Ed: Eu não sei, Doug. Sempre conseguimos despachar o produto.

Doug: Isso é verdade, mas não sem muito sofrimento, e esse projeto parece ser maior e mais complexo do que qualquer coisa que tenhamos feito no passado.

Jamie: Não parece tão difícil, mas eu concordo... Nossa abordagem *ad hoc* para os projetos anteriores não vai funcionar aqui, particularmente se tivermos um prazo muito apertado.

Doug (sorrindo): Eu quero ser um pouco mais profissional na nossa abordagem. Fui a um curso rápido na última semana e aprendi bastante sobre engenharia de software... boa coisa. Precisamos de um processo aqui.

Jamie (franzindo a testa): Meu trabalho é construir programas de computador, não ficar manipulando papel.

Doug: Dê uma chance antes de ficar brava comigo. Eis o que eu quero dizer. [Doug prossegue descrevendo o arcabouço de processo descrito no Capítulo 2 e os modelos prescritivos de processo apresentados até agora.]

Doug: Assim, de qualquer modo, parece que um modelo linear não é adequado para nós... ele considera que tenhamos todos os requisitos antecipadamente e, conhecendo esta empresa, isso não é provável.

Vinod: Sim, e aquele modelo RAD parece orientado demais à tecnologia da informação... provavelmente bom para construir um sistema de controle de estoque ou alguma outra coisa, mas não é adequado para o *CasaSegura*.

Doug: Eu concordo.

Ed: Essa abordagem de prototipagem parece boa. Muito parecida com o que nós fazemos aqui.

Vinod: Esse é que é o problema. Estou preocupado que ela não vai nos dar estrutura suficiente.

Doug: Não se preocupe, temos muitas outras opções e eu quero que vocês escolham o que é melhor para a equipe e para o projeto.

3.4.2 O Modelo Espiral

O *modelo espiral*, originalmente proposto por Boehm [BOE88], é um modelo evolucionário de processo de software que combina a natureza iterativa da *prototipagem* com os aspectos controlados e sistemáticos do modelo em cascata. Ele fornece o potencial para o desenvolvimento rápido de versões de software cada vez mais completas. Boehm [BOE01] descreve o modelo da seguinte maneira:

> O modelo espiral de desenvolvimento é um gerador de *modelo de processo guiado por risco* usado para guiar a engenharia de sistemas intensivos em software com vários interessados concorrentes. Ele tem duas principais características distintas. A primeira é uma abordagem *cíclica*, para aumentar incrementalmente o grau de definição e implementação de um sistema enquanto diminui seu grau de risco. A outra é um conjunto de *marcos de ancoragem*, para garantir o comprometimento dos interessados com soluções exeqüíveis e mutuamente satisfatórias para o sistema.

PONTO CHAVE

O modelo espiral pode ser adaptado para ser aplicado ao longo de todo o ciclo de vida de uma aplicação, desde o desenvolvimento conceitual até a manutenção.

Usando o modelo espiral, o software é desenvolvido numa série de versões evolucionárias. Durante as primeiras iterações, as versões podem ser um modelo de papel ou protótipo. Durante as últimas iterações, são produzidas versões cada vez mais completas do sistema submetido à engenharia.

Um modelo espiral é dividido em um conjunto de atividades de arcabouço definidas pela equipe de engenharia de software. Para fins ilustrativos, usamos as atividades genéricas de arcabouço discutidas anteriormente.[5] Cada uma das atividades de arcabouço representa um segmento do caminho espiral ilustrado na Figura 3.5. À medida que esse processo evolucionário começa, a equipe de software realiza atividades que são indicadas por um circuito em volta da espiral em sentido horário, começando pelo centro. O risco (Capítulo 25) é considerado à medida que cada evolução é feita. Os *marcos de ancoragem* — uma combinação de produtos de trabalho e condições que são obtidas ao longo do caminho da espiral — são indicados para cada passagem evolucionária.

5 O modelo espiral discutido nesta seção é uma variante do modelo proposto por Boehm. Para mais informações sobre o modelo espiral original, veja [BOE88]. Discussões mais recentes sobre o modelo espiral de *Boehm* podem ser encontradas em [BOE98].

O primeiro circuito em torno da espiral poderia resultar no desenvolvimento da especificação de um produto; passagens subseqüentes em torno da espiral poderiam ser usadas para desenvolver um protótipo e depois, progressivamente, versões mais sofisticadas do software. Cada passagem pela região de planejamento resulta em ajustes do plano do projeto. O custo e o cronograma são ajustados com base no feedback derivado do cliente após a entrega. Além disso, o gerente do projeto ajusta o número planejado de iterações necessárias para completar o software.

Diferentemente de outros modelos de processo, que terminam quando o software é entregue, o modelo espiral pode ser adaptado para aplicação ao longo da vida do software de computador. Assim, o primeiro circuito em volta da espiral poderia representar um "projeto de desenvolvimento de conceitos" que começa no centro da espiral e continua por várias iterações[6] até que o desenvolvimento do conceito seja completado. Se o conceito for desenvolvido em um produto real, o processo prossegue para fora na espiral e um "projeto de desenvolvimento de novo produto" começa. O novo produto vai evoluir por meio de um certo número de iterações, em torno da espiral. Depois, um circuito em volta da espiral poderia ser usado para representar um "projeto de aperfeiçoamento de produto". Em resumo, a espiral, quando caracterizada desse modo, permanece operacional até que o software seja retirado de serviço. Há momentos em que o processo fica adormecido, mas sempre que uma modificação é iniciada, o processo começa no ponto de entrada adequado (por exemplo, aperfeiçoamento de produto).

Veja na Web

Informações úteis sobre o modelo espiral podem ser obtidas em www.sei.cmu.edu/cbs/spiral2000/.

AVISO

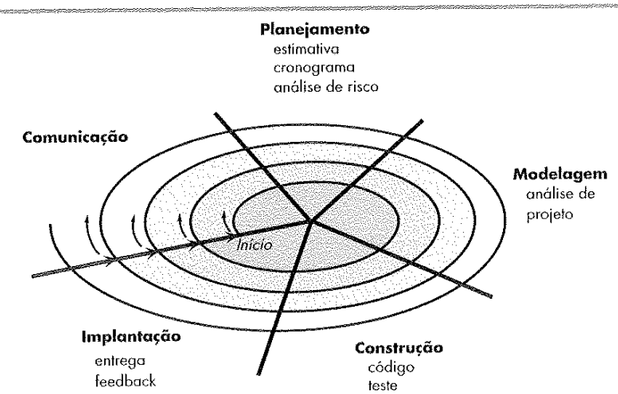
Se sua gerência exige desenvolvimento com orçamento fixo (geralmente uma má idéia), a espiral pode ser um problema: à medida que cada circuito é completado o custo do projeto é revisitado e revisado.

O modelo espiral é uma abordagem realista do desenvolvimento de sistemas e softwares de grande porte. Como o software evolui à medida que o processo avança, o desenvolvedor e o cliente entendem melhor e reagem aos riscos de cada nível evolucionário. O modelo espiral usa a prototipagem como um mecanismo de redução de risco, porém, mais importante, permite ao desenvolvedor aplicar a abordagem de prototipagem em qualquer estágio da evolução do produto. Ele mantém a abordagem sistemática passo-a-passo, sugerida pelo ciclo de vida clássico, mas o incorpora a um arcabouço iterativo que reflete mais realisticamente o mundo real. O modelo espiral exige a consideração direta dos riscos técnicos em todos os estágios do projeto e, se aplicado adequadamente, deve reduzir os riscos antes que eles se tornem problemáticos.

No entanto, como outros paradigmas, o modelo espiral não é uma panacéia. Pode ser difícil convencer os clientes (particularmente em situações de contrato) que a abordagem evolucionária

FIGURA 3.5

Um modelo espiral típico



6 As setas que apontam para dentro ao longo do eixo que separa a região de implantação da região de comunicação indicam um potencial de iteração local ao longo do mesmo caminho espiral.

Document shared on www.docsity.com

é controlável. Ela exige competência considerável na avaliação de riscos e depende dessa competência para ter sucesso. Se um risco importante não for descoberto e gerenciado, fatalmente ocorrerão problemas.

3.4.3 O Modelo de Desenvolvimento Concorrente

O *modelo de desenvolvimento concorrente*, algumas vezes chamado de *engenharia concorrente*, pode ser representado esquematicamente com uma série de atividades de arcabouço, ações e tarefas de engenharia de software e seus estados associados. Por exemplo, a atividade *modelagem* definida para o modelo espiral é realizada pela invocação das seguintes ações: prototipagem e/ou modelagem de análises e especificação e projeto.[7]

AVISO

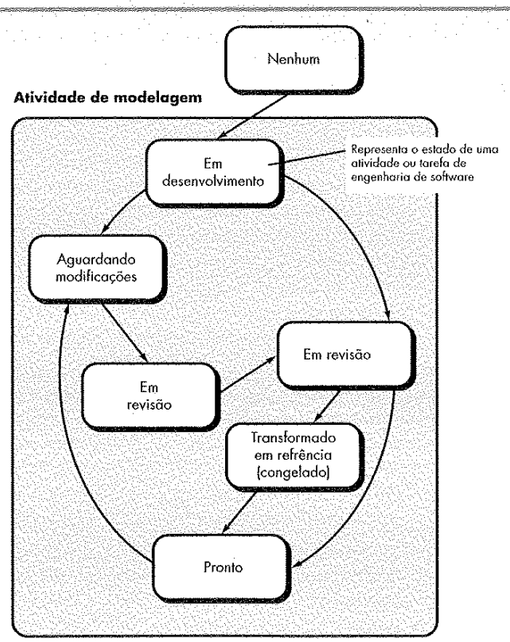
O modelo concorrente é freqüentemente mais adequado para projetos de engenharia de sistemas em que diferentes equipes de engenharia (Capítulo 6) estão envolvidas.

A Figura 3.6 mostra uma representação esquemática de uma tarefa de engenharia de software dentro da atividade de modelagem para o modelo concorrente de processo. A atividade — *modelagem* — pode estar em qualquer um dos estados[8] notados a qualquer tempo. Analogamente, outras atividades ou tarefas (por exemplo, comunicação ou construção) podem ser representadas de forma semelhante. Todas as atividades existem concomitantemente, mas estão em diferentes estados. Por exemplo, no começo de um projeto a atividade de *comunicação* (que não é mostrada na figura) completou sua primeira iteração e está no estado **aguardando modificações**. A atividade de *modelagem* que estava no estado **nenhum**, enquanto a comunicação inicial era completada, sofre agora a transição para o estado **em desenvolvimento**. Se, todavia, o cliente indicar que devem ser feitas modificações nos requisitos, a atividade de *modelagem* mudará do estado **em desenvolvimento** para o estado de **aguardando modificações**.

O modelo de processo concorrente define uma série de eventos que vão disparar transições de estado para estado, para cada uma das atividades, ações ou tarefas de engenharia de software. Por exemplo, durante os primeiros estágios do projeto (uma ação de engenharia de software que ocorre durante a atividade de modelagem), uma inconsistência é descoberta no modelo de análise. Isso gera o evento *correção do modelo de análise*, que vai disparar a passagem da atividade de *análise* do estado **pronto** para o estado **aguardando modificações**.

FIGURA 3.6

Um elemento do modelo concorrente de processo



7 Deve-se ressaltar que análise e projeto são ações complexas, que exigem discussão substancial. A Parte 2 deste livro considera esses tópicos em detalhe.

8 Um *estado* é um modo de comportamento observável externamente.

O modelo concorrente de processo é aplicável a todos os tipos de desenvolvimento de software e fornece uma imagem precisa do estado atual de um projeto. Em vez de confinar as atividades, ações e tarefas de engenharia de software a uma seqüência de eventos, ele define uma rede de atividades. Cada atividade, ação ou tarefa da rede existe simultaneamente com outras atividades, ações ou tarefas. Eventos gerados em um ponto da rede do processo disparam transições entre estados.

3.4.4 Um Comentário Final sobre Processos Evolucionários

Já mencionamos que o software de computador moderno é caracterizado por modificações contínuas, prazos muito curtos e por uma enfática necessidade de satisfação do cliente/usuário. Em muitos casos, o período até a colocação no mercado é o requisito gerencial mais importante. Se um nicho de mercado é perdido, o projeto de software em si pode ser insignificante.[9]

> "Eu só vim até aqui e apenas o amanhã guia o meu caminho."
>
> Dave Mathews Band

Os modelos evolucionários de processo foram concebidos para resolver esses pontos e, no entanto, como uma classe geral de modelos de processo, eles também têm pontos fracos. Esses pontos foram resumidos por Nogueira e seus colegas [NOG00]:

> Apesar dos benefícios inquestionáveis dos processos evolucionários do software, temos algumas preocupações. A primeira preocupação é que prototipagem [e outros processos evolucionários mais sofisticados] traz problemas para o planejamento do projeto por causa do número incerto de ciclos necessários para construir o produto. A maioria das técnicas de gestão e estimativa de projeto é baseada na disposição linear das atividades de modo que elas não se encaixem completamente.
>
> Segunda, os processos evolucionários de software não estabelecem a velocidade máxima da evolução. Se a evolução ocorrer muito depressa, sem um período de descanso, é certo que o processo vai ficar em caos. Por outro lado, se a sua velocidade for muito lenta, então a produtividade pode ser afetada...
>
> Terceira, os processos de software devem ser focados na flexibilidade e extensibilidade, em vez de na alta qualidade. Essa afirmativa parece assustadora. No entanto, devemos priorizar a velocidade de desenvolvimento sobre zero-defeitos. Estender o desenvolvimento a fim de conseguir alta qualidade pode resultar em entrega atrasada do produto, quando o nicho de oportunidade tiver desaparecido. Esse deslocamento de paradigma é imposto pela concorrência à beira do caos.

De fato, um processo de software que enfoca a flexibilidade, a extensibilidade e a velocidade do desenvolvimento em detrimento da alta qualidade parece assustador. No entanto, essa idéia tem sido proposta por vários especialistas muito respeitados em engenharia de software (por exemplo, [YOU95], [BAC97]).

A intenção dos modelos evolucionários é desenvolver softwares de alta qualidade[10] de maneira iterativa ou incremental. No entanto, é possível usar um processo evolucionário para enfatizar flexibilidade, extensibilidade e velocidade de desenvolvimento. O desafio para as equipes de software e seus gerente é estabelecer um equilíbrio adequado entre esses parâmetros críticos de projeto e de produto e a satisfação do cliente (o maior árbitro da qualidade do software).

9 É importante notar, no entanto, que ser o primeiro a ser colocado no mercado não é garantia de sucesso. De fato, muitos produtos de software muito bem-sucedidos têm sido o segundo ou até o terceiro a ser colocado no mercado (aprendendo com os erros dos seus predecessores).

10 Nesse contexto, a qualidade de software é definida mais amplamente para incluir não apenas a satisfação do cliente, mas também uma variedade de critérios técnicos que serão discutidos no Capítulo 26.

Document shared on www.docsity.com

CASASEGURA

Seleção de um Modelo de Processo, Parte 2

A cena: Sala de reuniões do grupo de engenharia de software na empresa CPI, uma empresa que fabrica produtos de consumo para uso residencial e comercial.

Os personagens: Lee Warren, gerente de engenharia; Doug Miller, gerente de engenharia de software, Ed e Vinod, membros da equipe de software.

A conversa:

[Doug descreve as opções de processo evolucionário.]

Ed: Agora estou vendo alguma coisa da qual gosto. Uma abordagem incremental faz sentido e eu realmente gosto do fluxo daquela coisa de modelo espiral. Isso é manter a coisa real.

Vinod: Eu concordo. Entregamos um incremento, aprendemos por meio do feedback do cliente, replanejamos e depois entregamos outro incremento. Ela também se encaixa na natureza do produto. Podemos pôr alguma coisa no mercado rapidamente e depois adicionar funcionalidades a cada versão, isto é, incremento.

Lee: Espere um pouco, você disse que reformamos o plano em cada volta em torno da espiral, Doug? Isso não é tão bom, precisamos de um plano, de um cronograma e precisamos nos ater a ele.

Doug: Essa é uma escola de pensamento antiga, Lee. Como o Ed disse, temos de manter a coisa real. Eu proponho que é melhor adaptar o plano à medida que aprendemos mais e à medida que as modificações são solicitadas. É muito mais realista. Qual o sentido de um plano se ele não reflete a realidade?

Lee (franzindo a testa): Eu penso que sim, mas a gerência superior não vai gostar disso ... eles querem um plano fixo.

Doug (sorrindo): Então você vai ter que reeducá-los, meu amigo.

3.5 MODELOS ESPECIALIZADOS DE PROCESSO

Os modelos especializados de processo têm muitas das características de um ou mais dos modelos convencionais apresentados nas seções anteriores. Entretanto, os modelos especializados tendem a ser aplicados quando uma abordagem de engenharia de software estreitamente definida é escolhida.[11]

3.5.1 Desenvolvimento Baseado em Componentes

Os componentes de software comercial de prateleira (COTS — *Commercial-off-the-shelf*), desenvolvidos por vendedores que os oferecem como produtos, podem ser usados quando o software precisar ser construído. Esses componentes fornecem funcionalidades-alvo com interfaces bem definidas que permitem ao componente ser integrado no software.

O *modelo de desenvolvimento* baseado em componentes (Capítulo 30) incorpora muitas das características do modelo espiral. Evolucionário por natureza [NIE92], demanda uma abordagem iterativa para a criação de softwares. Todavia, o modelo compõe aplicações a partir de componentes de software previamente preparados. As atividades de modelagem e construção começam com a identificação de componentes candidatos. Esses componentes podem ser projetados como módulos de software convencional ou como classes ou pacotes de classes orientados a objetos[12]. Independentemente da tecnologia usada para criar os componentes, o modelo de desenvolvimento baseado em componentes incorpora os seguintes passos (implementados usando uma abordagem evolucionária):

Veja na Web

Informações úteis sobre desenvolvimento baseado em componentes podem ser obtidas em www.cbd-hq.com.

- Produtos baseados em componentes disponíveis são pesquisados e avaliados para o domínio da aplicação em questão.

11 Em alguns casos, esses modelos especializados de processo poderiam ser melhor caracterizados como uma coleção de técnicas ou uma metodologia para atingir uma meta específica de desenvolvimento de software. No entanto, eles envolvem um processo.

12 A tecnologia orientada a objetos será discutida na Parte 2 deste livro. Nesse contexto, uma classe encapsula um conjunto de dados e procedimentos que processam os dados. Um pacote de classes é uma coleção de classes relacionadas que trabalham juntas para chegar a um resultado final.

- Tópicos de integração de componentes são considerados.
- Uma arquitetura de software (Capítulo 10) é projetada para acomodar os componentes.
- Componentes (Capítulo 11) são integrados à arquitetura.
- Testes abrangentes (Capítulos 13 e 14) são realizados para garantir a funcionalidade adequada.

O modelo de desenvolvimento baseado em componentes leva ao reuso de software, e a reusabilidade fornece aos engenheiros vários benefícios mensuráveis. Com base em estudos de reusabilidade, a QSM Associates, Inc. relata que o desenvolvimento baseado em componentes leva à redução de 70% do prazo do ciclo de desenvolvimento; uma redução de 84% no custo do projeto; e um índice de produtividade de 26,2, comparado com o padrão de 16,9 para a indústria [YOU94]. Apesar desses resultados serem uma função da robustez da biblioteca de componentes, existe pouca dúvida de que o modelo de desenvolvimento baseado em componentes fornece vantagens significativas para os engenheiros de software.

3.5.2 O Modelo de Métodos Formais

O modelo de *métodos formais* (Capítulo 28) abrange um conjunto de atividades que levam à especificação matemática formal do software de computador. Os métodos formais permitem ao engenheiro de software especificar, desenvolver e verificar um sistema baseado em computador pela aplicação de uma rigorosa notação matemática. Uma variante dessa abordagem, chamada *engenharia de software sala limpa* [MIL87, DYE92], é aplicada atualmente por algumas organizações de desenvolvimento de software e será discutida no Capítulo 29.

> "É mais fácil escrever um programa incorreto do que entender um correto."
>
> **Alan Perlis**

Quando métodos formais são usados durante o desenvolvimento, eles fornecem um mecanismo para eliminação de muitos dos problemas que são difíceis de resolver usando outros paradigmas de engenharia de software. Ambigüidade, inconclusão e inconsistência podem ser descobertas e corrigidas mais facilmente — não por meio de revisões *ad hoc*, mas por meio da aplicação de análise matemática. Quando são usados durante o projeto, métodos formais servem de base para a verificação do programa e, assim, permitem que o engenheiro de software descubra e corrija erros que poderiam passar desapercebidos.

Apesar de não vir a ser uma abordagem de uso geral, o modelo de métodos formais oferece a promessa de softwares livres de defeitos. Todavia, foram formuladas as seguintes preocupações sobre sua aplicabilidade num ambiente comercial:

? Se os métodos formais podem demonstrar a correção do software, por que eles não são amplamente usados?

- O desenvolvimento de modelos formais é atualmente muito lento e dispendioso.
- Como poucos desenvolvedores de software têm o preparo necessário para aplicar métodos formais, torna-se necessário um treinamento extensivo.
- É difícil usar os modelos como um mecanismo de comunicação, com clientes despreparados tecnicamente.

Apesar dessas preocupações, a abordagem de métodos formais tem ganho adeptos entre desenvolvedores de software que precisam construir softwares críticos em termos de segurança (por exemplo, desenvolvedores de aviônica de aeronaves e de dispositivos médicos) e entre desenvolvedores de software que sofreriam pesadas sanções econômicas se ocorressem erros no software.

3.5.3 Desenvolvimento de Software Orientado a Aspectos

Independentemente do processo de software escolhido, os construtores de softwares complexos invariavelmente implementam um conjunto de características, funções e conteúdo de informa-

Document shared on www.docsity.com

Veja na Web

Uma grande variedade de recursos e informação sobre a POA (programação orientada a aspectos) pode ser encontrada em **aosd.net**.

ções localizadas. Essas características localizadas de software são modeladas como componentes (por exemplo, classes orientadas a objetos) e depois construídas dentro do contexto de uma arquitetura de sistemas. À medida que sistemas modernos baseados em computador tornam-se mais sofisticados (e complexos), certas "preocupações" — propriedades solicitadas pelo cliente ou áreas de preocupação técnica — cobrem toda a arquitetura. Algumas preocupações são propriedades de alto nível de um sistema (por exemplo, segurança, tolerância a falhas). Outras preocupações afetam funções (por exemplo, aplicação de regras de negócio), enquanto outras são sistêmicas (por exemplo, sincronização de tarefas ou gestão de memória).

Quando as preocupações entrecortam várias funções, características e informações do sistema, elas são freqüentemente referidas como *preocupações transversais*. *Requisitos referentes a aspectos* definem essas preocupações transversais, que têm impacto em toda a arquitetura do software. O DSOA — *Desenvolvimento de Software Orientado a Aspectos* (ASOD — *Aspect Oriented Software Development*), freqüentemente referido como POA (*Programação Orientada a Aspectos*), é um paradigma relativamente novo de engenharia de software que fornece um processo e abordagem metodológica para definir, especificar, projetar e construir *aspectos* — "mecanismos que transcendem subrotinas e herança para localizar a expressão de uma preocupação transversal" [ELR01].

PONTO CHAVE

O DSOA define "aspectos" que expressam preocupações do cliente que permeiam múltiplas funções, características e informações do sistema.

Grundy [GRU02] fornece maior discussão de aspectos no contexto do que ele chama de ECOA — *engenharia de componentes orientada a aspectos* (em inglês, *AOCE — Aspect-Oriented Component Engineering*).

> A ECOA usa um conceito de fatias horizontais por meio de componentes de software decompostos verticalmente chamados de "aspectos", para caracterizar propriedades de componentes transversais funcionais e não-funcionais. Aspectos comuns e sistêmicos incluem interfaces com o usuário, trabalho colaborativo, distribuição, persistência, gestão de memória, processamento de transações, segurança, integridade etc. Componentes podem fornecer ou exigir um ou mais "detalhes do aspecto" relativos a um determinado aspecto, por exemplo, um mecanismo de visão, disponibilidade extensível e espécie de interface (aspectos da interface com o usuário); geração, transporte e recebimento de eventos (aspectos de distribuição); armazenamento, recuperação e indexação de dados (aspectos de persistência); autenticação, codificação e direitos de acesso (aspectos de segurança); atomicidade de transação, controle de concorrência e estratégia de registro (aspectos de transação); e assim por diante. Cada detalhe do aspecto tem um certo número de propriedades, relativas às características funcionais e/ou não funcionais do detalhe de aspecto.

FERRAMENTAS DE SOFTWARE

Gestão de Processo

Objetivo: Assistir a definição, execução e gestão de modelos prescritivos de processo.

Mecânica: Ferramentas de gestão de processo permitem que uma organização ou equipe de software defina um modelo completo de processo de software (atividades de arcabouço, ações, tarefas, pontos de verificação de garantia de qualidade, marcos de referência e produtos de trabalho). Além disso, as ferramentas fornecem um roteiro à medida que os engenheiros de software fazem o trabalho técnico e um gabarito para gestores que precisam monitorar e controlar o processo de software.

Ferramentas Representativas: [13]

- O *GDPA*, um conjunto de ferramentas para pesquisa de definição de processo, desenvolvido na Universidade de Bremen, na Alemanha (*www.informatik.uni-bremen.de/uniform/gdpa/home.htm*), fornece uma ampla variedade de funções de modelagem e gestão de processos.
- O *SpeeDev*, desenvolvido pela SpeeDev Corporation (*www.speedv.com*) inclui um conjunto de ferramentas para definição de processos, gestão de requisitos, resolução de tópicos, planejamento de projeto e monitoração.
- O *Step Gate Process*, desenvolvido pela Objexis (*www.objexis.com*) inclui muitas ferramentas que assistem a automação do fluxo de trabalho.

Uma discussão válida dos métodos e notação que podem ser usados para definir e descrever um modelo de processo completo pode ser encontrada em http://205.252.62.38/English/D-ProcessNotation.htm.

13 As ferramentas indicadas aqui não representam uma recomendação, mas apenas uma exemplificação de ferramentas dessa categoria. Na maior parte dos casos, os nomes são marcas registradas de seus respectivos desenvolvedores.

Um processo orientado a aspectos distinto ainda não foi totalmente desenvolvido. No entanto, é provável que tal processo adote características tanto do modelo de processo espiral quanto do concorrente (Seções 3.4.2 e 3.4.3). A natureza evolucionária da espiral é apropriada à medida que os aspectos são identificados e depois construídos. A natureza paralela do desenvolvimento concorrente é essencial, porque aspectos passam por engenharia independentemente de componentes de software localizados e, no entanto, os aspectos têm um impacto direto nesses componentes. Assim, é essencial instanciar a comunicação assíncrona entre atividades do processo de software aplicáveis à engenharia e a construção de aspectos e de componentes.

É melhor deixar uma discussão detalhada do desenvolvimento de software orientado a aspectos para livros dedicados ao assunto. O leitor interessado deveria ver [GRA03], [KIS02] ou [ELR01].

3.6 O Processo Unificado

Em seu livro pioneiro sobre o *Processo Unificado*, Ivar Jacobson, Grady Booch e James Rumbaugh [JAC99] discutem a necessidade de um processo de software "guiado por casos de uso, centrado na arquitetura, iterativo e incremental" quando afirmam:

> Hoje, a tendência em software é em direção a sistemas maiores, mais complexos. Isso se deve, em parte, ao fato de que os computadores têm se tornado mais potentes a cada ano, levando os usuários a esperar mais deles. Essa tendência tem também sido influenciada pelo uso da Internet, que está se expandindo, para trocar toda espécie de informação... Nosso apetite por softwares cada vez mais sofisticados cresce à medida que aprendemos de uma versão de um produto para a seguinte como o produto poderia ser aperfeiçoado. Desejamos softwares que sejam melhor adaptados às nossas necessidades, mas que, por sua vez, não torne o software somente mais complexo. Em resumo, desejamos mais.

De certo modo, o PU (Processo Unificado) é uma tentativa de apoiar-se nos melhores recursos e características dos modelos convencionais de processo de software, mas caracterizá-los de um modo que implemente muitos dos melhores princípios de desenvolvimento ágil de softwares (Capítulo 4). O Processo Unificado reconhece a importância da comunicação com o cliente e dos métodos diretos para descrever a visão do cliente de um sistema (isto é, o caso de uso.)[14] Ele enfatiza o importante papel da arquitetura de software e "ajuda o arquiteto a se concentrar nas metas corretas, tais como compreensibilidade, abertura a modificações futuras e reuso" [JAC99]. Ele sugere um fluxo de processo que é iterativo e incremental, dando a sensação evolucionária que é essencial no desenvolvimento moderno do software.

Nesta seção apresentamos um panorama dos elementos-chave do processo unificado. Na Parte 2 deste livro discutiremos os métodos que compõem o processo e as técnicas e notação complementares da modelagem UML[15] que são necessárias à medida que o Processo Unificado é aplicado no trabalho real de engenharia de software.

3.6.1 Um Breve Histórico

Durante a década de 1980 e no início da de 1990, métodos e linguagens de programação orientados a objetos (OO)[16] ganharam uma audiência espalhada por toda a comunidade de engenharia de software. Uma grande variedade de métodos de análise orientada a objetos (AOO) e projeto orientado a objetos (POO) foram propostos durante o mesmo período de tempo, e um modelo de processo

14 Um *caso de uso* (Capítulos 7 e 8) é um texto narrativo ou gabarito que descreve uma função ou característica do sistema do ponto de vista do usuário. O caso de uso é escrito pelo usuário e serve como base para a criação de um modelo de análise mais abrangente.

15 UML (Linguagem Unificada de Modelagem, em inglês — *Unified Modeling Language*) tem se tornado a notação mais amplamente usada para modelagem de análise e projeto. Ela representa um casamento de três importantes notações orientadas a objetos.

16 Se você não está familiarizado com métodos orientados a objetos, um breve panorama é apresentado nos Capítulos 8 e 9. Para uma apresentação mais detalhada veja [REE02], [STI01] ou [FOW99].

Document shared on www.docsity.com

orientado a objetos de propósito geral (semelhante aos modelos evolucionários apresentados neste capítulo) foi introduzido. Como a maioria dos "novos" paradigmas de engenharia de software, os adeptos de cada um dos métodos de AOO e POO argumentaram sobre qual era o melhor, mas nenhum método ou linguagem individual dominou a paisagem de engenharia de software.

No início da década de 1990, James Rumbaugh [RUM91], Grady Booch [BOO94] e Ivar Jacobson [JAC92] começaram a trabalhar em um "método unificado" que combinaria as melhores características de cada um dos seus métodos individuais e adotaria características adicionais propostas por outros especialistas (por exemplo, [WIR90]) no ramo de OO. O resultado foi a UML — uma *linguagem unificada de modelagem* que contém uma notação robusta para a modelagem e desenvolvimento de sistemas OO. Por volta de 1997, a UML tornou-se uma norma industrial para o desenvolvimento de softwares orientados a objetos. Ao mesmo tempo, a Rational Corporation e outros vendedores desenvolveram ferramentas automatizadas para apoiar métodos UML.

A UML fornece a tecnologia necessária para apoiar a prática de engenharia de software orientada a objetos, mas não fornece o arcabouço de processo para guiar as equipes de projeto na aplicação da tecnologia. Ao longo dos cinco anos seguintes, Jacobson, Rumbaugh e Booch desenvolveram o *Processo Unificado*, um arcabouço para a engenharia de software orientada a objetos usando a UML. Hoje em dia, o Processo Unificado e a UML são amplamente usados em projetos OO de todas as naturezas. O modelo iterativo e incremental proposto pelo PU pode, e deve, ser adaptado para satisfazer às necessidades específicas de projeto.

Uma variedade de produtos de trabalho (por exemplo, modelos e documentos) pode ser produzida como conseqüência da aplicação da UML. No entanto, eles são freqüentemente mutilados por engenheiros de software para tornar o desenvolvimento mais ágil e mais suscetível a modificações.

Veja na Web

Documentos importantes sobre o PU podem ser encontrados em www.rational.com/products/rup/writepapers.jsp.

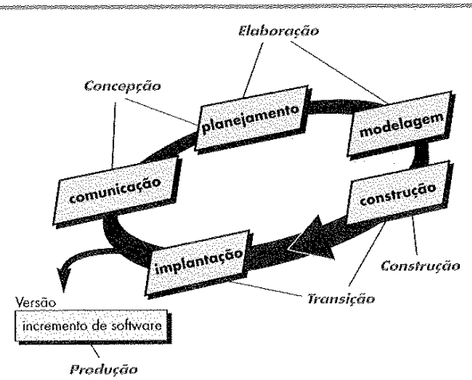
3.6.2 Fases do Processo Unificado[17]

Discutimos cinco atividades genéricas de arcabouço e argumentamos que elas podem ser usadas para descrever qualquer modelo de processo de software. O Processo Unificado não é exceção. A Figura 3.7 mostra as "fases" do PU (Processo Unificado) e as relaciona com as atividades genéricas que foram discutidas no Capítulo 2.

A fase de *concepção* do PU abrange atividades de comunicação com o cliente e de planejamento. Em colaboração com o cliente e com os usuários finais, os requisitos de negócio para o software são identificados, um rascunho da arquitetura do sistema é proposto e um plano para a natureza iterativa e incremental do projeto que vai ser seguido é desenvolvido. Requisitos de negócio fundamentais são descritos por meio de casos de uso preliminares que descrevem quais características e

FIGURA 3.7

O Processo Unificado



17 O Processo Unificado é algumas vezes chamado de *Processo Unificado Racional* (*RUP – Rational Unified Process*) por causa da Rational Corporation, uma contribuinte pioneira para o desenvolvimento e refinamento do processo e uma construtora de ambientes completos (ferramentas e tecnologias) que apóiam o processo.

funções são desejáveis para cada classe importante de usuários. Em geral, um caso de uso descreve uma seqüência de ações que são realizadas por um *ator* (por exemplo, uma pessoa, uma máquina, outro sistema) à medida que o ator interage com o software. Casos de uso ajudam a identificar o escopo do projeto e fornecem base para o planejamento do projeto.

PONTO CHAVE

As *fases* do PU são semelhantes em intenção às atividades genéricas de arcabouço definidas neste livro.

A arquitetura nesse ponto não é nada mais que um esboço tentativo dos principais subsistemas e das funções e características que os compõem. Posteriormente, a arquitetura será refinada e expandida em um conjunto de modelos que representarão diferentes visões do sistema. O planejamento identifica recursos, avalia os principais riscos, define um cronograma e estabelece uma base para as fases que devem ser aplicadas à medida que o incremento de software é desenvolvido.

A fase de *elaboração* inclui a comunicação com o cliente e atividades de modelagem do modelo genérico de processo (Figura 3.7). A elaboração refina e expande os casos de uso preliminares que foram desenvolvidos como parte da fase de concepção e expande a representação arquitetural para incluir cinco visões diferentes do software — o modelo de casos de uso, o modelo de análise, o modelo de projeto, o modelo de implementação e o modelo de implantação. Em alguns casos, a elaboração cria uma "referência arquitetural executável" [ARL02] que representa uma primeira versão do sistema executável.[18] A referência arquitetural demonstra a viabilidade da arquitetura, mas não fornece todas as características e funções requeridas para uso do sistema. Além disso, o plano é cuidadosamente revisto no ápice da fase de elaboração para garantir que o escopo, os riscos e as datas de entrega permaneçam razoáveis. Modificações no plano podem ser feitas nesse momento.

A fase de *construção* do PU é idêntica à atividade de construção definida para o processo genérico de software. Usando o modelo arquitetural como entrada, a fase de construção desenvolve ou adquire os componentes de software que vão tornar cada caso de uso operacional para os usuários finais. Para conseguir isso, os modelos de análise e projeto que foram iniciados durante a fase de elaboração são completados de modo a refletir a versão final do incremento de software. Todas as características e funções necessárias e requeridas do incremento de software (isto é, a versão) serão implementadas no código-fonte. À medida que os componentes são implementados, testes unitários são projetados e executados para cada um deles. Além disso, as atividades de integração (montagem de componentes e testes de integração) são conduzidas. Casos de uso são usados para derivar uma seqüência de testes de aceitação que serão executados antes do início da fase seguinte do PU.

Veja na Web

Discussões esclarecedoras e comentários sobre o PU podem ser encontrados em www.unifiedprocess.org.

A fase de *transição* do PU abrange os últimos estágios da atividade genérica de construção e a primeira parte da atividade genérica de implantação. O software é dado aos usuários finais para teste beta[19] e relatórios de *feedback* do usuário sobre defeitos e modificações necessárias. Além disso, a equipe de software cria as informações de apoio necessárias (por exemplo, manuais de usuário, guias de solução de problemas e procedimentos de instalação) que precisam ser entregues. Na conclusão da fase de transição, o incremento de software torna-se uma versão utilizável do software.

A fase de *produção* do PU coincide com a atividade de implantação do processo genérico. Durante essa fase, o uso do software é monitorado, é fornecido suporte para o ambiente de operação (infraestrutura) e os relatórios de defeito e solicitações de modificações são submetidos e avaliados.

É provável que, ao mesmo tempo em que as fases de construção, transição e produção estejam sendo conduzidas, o trabalho já tenha sido iniciado no incremento de software seguinte. Isso significa que as cinco fases do PU não ocorrem em seqüência, mas em titubeante concorrência.

Um fluxo de trabalho de engenharia de software é distribuído ao longo de todas as fases do PU. No contexto do PU, um *fluxo de trabalho* é análogo a um conjunto de tarefas (definido no Capítulo 2), isto é, um fluxo de trabalho identifica as tarefas exigidas para realizar uma ação importante de engenharia de software e os produtos de trabalho que são produzidos em conseqüência da conclusão bem-sucedida dessas tarefas. Deve-se notar que nem toda tarefa identificada para um fluxo de trabalho do PU é conduzida para qualquer projeto de software. A equipe adapta o processo (ações, tarefas, subtarefas e produtos de trabalho) para satisfazer às suas necessidades.

18 É importante notar que a referência arquitetural não é um protótipo (Seção 3.4.1) no sentido de que ela não é descartada. Pelo contrário, a referência é incorporada durante a fase seguinte do PU.

19 Teste beta é uma ação controlada de teste (Capítulo 13) na qual o software é usado por usuários finais reais com a intenção de descobrir defeitos e deficiências. Um esquema formal de relato de defeitos/deficiências é estabelecido e a equipe de software avalia o *feedback*.

Document shared on www.docsity.com

3.6.3 Produtos de Trabalho do Processo Unificado

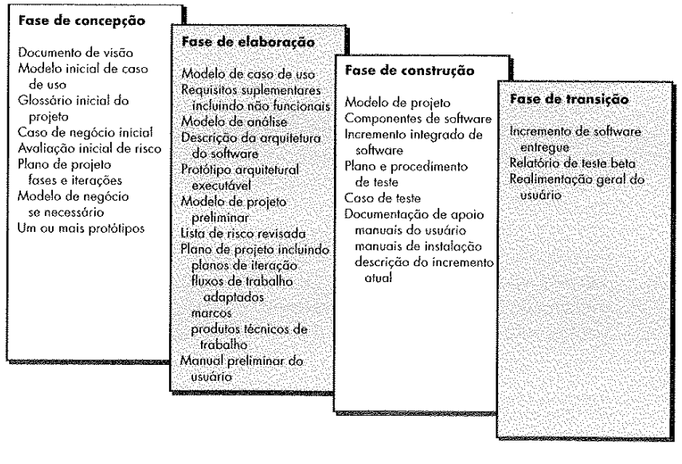
A Figura 3.8 ilustra os *produtos de trabalho mais importantes* produzidos em conseqüência das quatro fases técnicas do PU. Durante a fase de concepção, o intuito é estabelecer uma "visão" global do projeto, identificar um conjunto de requisitos do negócio, preparar um caso de negócio para o software e definir riscos de negócio e de projeto que possam representar uma ameaça para o sucesso. Do ponto de vista dos engenheiros de software, o produto de trabalho mais importante produzido durante a concepção é o *modelo de casos de uso* — uma coleção de casos de uso que descreve como atores externos ("usuários" humanos e não humanos do software) interagem com o sistema e obtêm valor dele. Em essência, o modelo de casos de uso é uma coleção de cenários de uso descritos com gabaritos padronizados que implicam características e funções de software para descrever um conjunto de pré-condições, um fluxo de eventos ou cenário e um conjunto de pós-condições para a interação que é ilustrada. Inicialmente, os casos de uso descrevem requisitos no nível do domínio do negócio (isto é, o nível de abstração é alto). Entretanto, o modelo de casos de uso é refinado e elaborado à medida que cada fase do PU é conduzida, e serve como importante entrada para a criação dos produtos de trabalho subseqüentes. Durante a fase de concepção, apenas 10% a 20% do modelo de casos de uso é completado. Depois da elaboração, entre 80% e 90% do modelo já foi criado.

A fase de elaboração produz um conjunto de produtos de trabalho que elaboram requisitos (inclusive requisitos não-funcionais)[20] e produzem uma descrição arquitetural e um projeto preliminar. Quando o engenheiro de software inicia a análise orientada a objetos, o principal objetivo é definir um conjunto de classes de análise que descreva adequadamente o comportamento do sistema. O *modelo de análise* do PU é o produto de trabalho desenvolvido em conseqüência dessa atividade. As classes e pacotes de análise (coleções de classes) definidos como parte do modelo de análise são ainda mais refinados em um *modelo de projeto* que identifica classes de projeto, subsistemas e as interfaces entre os subsistemas. Tanto os modelos de análise quanto os de projeto expandem e refinam uma representação evolutiva da arquitetura do software. Além disso, a fase de elaboração revisita os riscos e o plano de projeto para garantir que ambos permaneçam válidos.

A fase de construção produz um modelo de *implementação* que traduz classes de projeto em componentes de software que serão construídos para concretizar o sistema, e um modelo de *implantação* mapeia os componentes para o ambiente físico de computação. Finalmente, um modelo de *teste* descreve os testes usados para garantir que os casos de uso sejam adequadamente refletidos no software que está sendo construído.

FIGURA 3.8

Principais produtos de trabalho produzidos em cada fase do PU



20 Requisitos que não podem ser depreendidos do modelo de casos de uso.

A fase de transição entrega o incremento de software e avalia os produtos de trabalho produzidos à medida que os usuários finais trabalham com o software. O *feedback* a partir do teste beta e solicitações de modificações qualitativas são produzidos nessa hora.

3.7 RESUMO

Modelos prescritivos de processo de software têm sido aplicados durante muitos anos em um esforço de trazer ordem e estrutura para o desenvolvimento de softwares. Cada um desses modelos convencionais sugere um fluxo de processo um tanto diferente, mas todos realizam o mesmo conjunto de atividades genéricas de arcabouço: comunicação, planejamento, modelagem, construção e implantação.

O modelo em cascata sugere uma progressão linear das atividades de arcabouço que é freqüentemente inconsistente com as realidades modernas (por exemplo, modificações contínuas, sistemas evolutivos, prazos apertados) do mundo de software. No entanto, ele tem aplicabilidade em situações em que os requisitos são bem definidos e estáveis.

Modelos incrementais de processo de software produzem o software como uma série de versões de incrementos. O modelo RAD é projetado para projetos maiores que devem ser entregues em prazos apertados.

Modelos evolucionários de processo reconhecem a natureza iterativa da maioria dos projetos de engenharia de software e são projetados para acomodar modificações. Modelos evolucionários, como os modelos espirais e de prototipagem, produzem produtos de trabalho incrementais (ou versões do software que funcionam) rapidamente. Esses modelos podem ser adotados para serem aplicados ao longo de todas as atividades de engenharia de software — desde o desenvolvimento de conceitos até a manutenção do sistema no longo prazo.

O modelo baseado em componentes enfatiza a reutilização e a montagem de componentes. O modelo de métodos formais encoraja uma abordagem com base na matemática para o desenvolvimento e a verificação de software. O modelo orientado a aspectos acomoda preocupações transversais que cobrem toda a arquitetura do sistema.

O Processo Unificado é um processo de software "orientado por casos de uso, centrado na arquitetura, iterativo e incremental", projetado como um arcabouço para métodos e ferramentas UML. O Processo Unificado é um modelo incremental no qual cinco fases são definidas: (1) uma fase de *concepção* que engloba tanto a comunicação com o cliente quanto atividades de planejamento, enfatiza o desenvolvimento e refinamento de casos de uso como o modelo principal; (2) uma fase de *elaboração* que engloba atividades de comunicação com o cliente e modelagem com foco na criação de modelos de análise e projeto com ênfase nas definições de classes e representações arquiteturais; (3) uma fase de *construção* que refina e então traduz o modelo de projeto para componentes de software implementados; (4) uma fase de *transição* que transfere o software do desenvolvedor para o usuário final para testes beta e aceitação; e (5) uma fase de *produção* em que contínuo monitoramento e suporte são conduzidos.

REFERÊNCIAS BIBLIOGRÁFICAS

[AMB02] Ambler, S. e Constantine, L., *The Unified Process Inception Phase*, CMP Books, 2002.
[ARL02] Arlow, J. e Neustadt, I., *UML and the Unified Process*, Addison-Wesley, 2002.
[BAC97] Bach, J., "Good Enough Quality: Beyond the Buzzword", *IEEE Computer*, v. 30, n. 8, ago. 1997, p. 96-98.
[BOE88] Boehm, B., "A Spiral Model for Software Development and Enhancement", *Computer*, v. 21, n. 5, maio 1988, p. 61-72.
[BOE98] _____, "Using the WINWIN Spiral Model: A Case Study", *Computer*, v. 31, n. 7, jul. 1998, p. 33-44.
[BOE01] _____, "The Spiral Model as a Tool for Evolutionary Software Acquisition", *CrossTalk*, maio 2001, disponível em htpp://www.stsc.hill.af.mil/crosstalk/2001/05/boehm.html.
[BOO94] Booch, G., *Object-Oriented Analysis and Design*, 2ª ed., Benjamin Cummings, 1994.
[BRA94] Bradac, M., Perry, D. e Votta, L., "Prototyping a Process Monitoring Experiment", *IEEE Trans. Software Engineering*, v. 20, n. 10, out. 1994, p.774-784.
[BRO75] Brooks, F., *The Mythical Man-Month*, Addison-Wesley, 1975.

Document shared on www.docsity.com

[BUT94] Butler, J., "Rapid Application Development in Action", *Managing System Development*, Applied Computer Research, v. 14, n. 5, maio 1994, p. 6-8.
[DYE92] Dyer, M., *The Cleanroom Approach to Quality Software Development*, Wiley, 1992.
[ELR 01] Elrad, T., Filman, R. e Bader, A., (Eds.), "Aspect-Oriented Programming", *Comm. ACM*, v. 44, n. 10, out. 2001, edição especial.
[FOW99] Fowler, M. e Scott, K., *UML Distilled*, 2ª ed., Addison-Wesley, 1999.
[GIL88] Gilb, T., *Principles of Software Engineering Management*, Addison-Wesley, 1988.
[GRA03] Gradecki, J. e Lesiecki, N., *Mastering AspectJ: Aspect-Oriented Programming in Java*, Wiley, 2003.
[GRU02] Grundy, J., "Aspect-Oriented Component Engineering", 2002, http://www.cs.auckland.ac.nz/~john-g/aspects.html.
[HAN95] Hanna, M., "Farewell to Waterfalls", *Software Magazine*, maio 1995, p. 38-46.
[HES96] Hesse, W., "Theory and Practice of the Software Process — A Field Study and its Implications for Project Management", *Software Process Technology*, 5º Workshop Europeu, EWSPT 96, LNCS 1149, 1996, p. 241-256.
[HES01] _____, "Dinosaur Meets Archaeopteryx? Seven Theses on Rational's Unified Process (RUP)", *Proc. 8th Intl. Workshop on Evaluation of Modeling Methods in System Analysis and Design*, Cap. VII, Interlaken, 2001.
[JAC92] Jacobson, I., *Object-Oriented Software Engineering*, Addison-Wesley, 1992.
[JAC99] _____, Booch, G., e Rumbaugh, J., *The Unified Software Development Process*, Addison-Wesley, 1999.
[KAU95] Kauffman, S., *At Home in the Universe*, Oxford, 1995.
[KER94] Kerr, J. e Hunter, R., *Inside RAD*, McGraw-Hill, 1994.
[KIS02] Kiselev, I., *Aspect-Oriented Programming with AspectJ*, Sams Publishers, 2002.
[MAR91] Martin, J., *Rapid Application Development*, Prentice-Hall, 1991.
[McDE93] McDermid, J. e Rook, P., "Software Development Process Models", *Software Engineer's Reference Book*, CRC Press, 1993, p. 15/26-15/28.
[MIL87] Mills, H. D., Dyer, M., e Linger, R., "Cleanroom Software Engineering", *IEEE Software*, set.1987, p. 19-25.
[NIE92] Nierstrasz, O., Gibbs, S. e Tsichritzis, D., "Component-Oriented Software Development", *CACM*, v. 35, n. 9, set. 1992, p. 160-165.
[NOG00] Nogueira, J., Jones, C. e Luqi, "Surfing the Edge of Chaos: Applications to Software Engineering", Command and Control Research and Technology Symposium, Naval Post Graduate School, Monterey, CA, jun. 2000, download de http://www.dodccrp.org/2000CCRTS/cd/html/pdf_papers/Track_4/075.pdf.
[REE02] Reed, P., *Developing Applications with Java and UML*, Addison-Wesley, 2002.
[REI95] Reilly, J. P., "Does RAD Live Up to the Hype", *IEEE Software*, set. 1995, p. 24-26.
[ROO96] Roos, J., "The Poised Organization: Navigating Effectively on Knowledge Landscapes", 1996, http://www.imd.ch/fac/roos/paper_po.html.
[ROY70] Royce, W. W., "Managing the Development of Large Software Systems: Concepts and Techniques", *Proc. WESCON*, ago. 1970.
[RUM91] Rumbaugh, J., *et al.*, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.
[STI01] Stiller, E. e LeBlanc, C., *Project-Based Software Engineering: An Object-Oriented Approach*, Addison-Wesley, 2001.
[WIR90] Wirfs-Brock, R., Wilkerson, B. e Weiner, L., *Designing Object-Oriented Software*, Prentice-Hall, 1990.
[YOU94] Yourdon, E., "Software Reuse", *Application Development Strategies*, v. 6, n. 12, dez.1994, p. 1-16.
[YOU95] _____, "When Good Enough Is Best", *IEEE Software*, v. 12, n. 3, maio 1995, p. 79-81.

Problemas e Pontos a Considerar

3.1. Leia [NOG00] e escreva um trabalho de duas ou três páginas que discuta o impacto do "caos" na engenharia de software.

3.2. Dê três exemplos de projetos de software que seriam apropriados para o modelo em cascata. Seja específico.

3.3. Dê três exemplos de projetos de software que seriam apropriados para o modelo de prototipagem. Seja específico.

3.4. Quais adaptações de processo são necessárias se o protótipo evoluir para um produto ou sistema a ser entregue?

3.5. Para alcançar o desenvolvimento rápido, o modelo RAD supõe a existência de uma coisa. O que é e por que a suposição não é sempre verdadeira?

3.6. Dê três exemplos de projetos de software que seriam apropriados para o modelo incremental. Seja específico.

3.7. À medida que você se move para fora ao longo do fluxo de processo espiral, o que pode dizer sobre o software que está sendo desenvolvido ou mantido?

3.8. É possível combinar modelos de processo? Se for, dê um exemplo.

3.9. O modelo concorrente de processo define um conjunto de "estados". Descreva o que esses estados representam com suas próprias palavras e, depois, indique como eles atuam no modelo concorrente de processo.

3.10. Quais são as vantagens e desvantagens em desenvolver softwares em que a qualidade é "suficientemente boa"? Isto é, o que acontece quando enfatizamos a velocidade de desenvolvimento sobre a qualidade de produto?

3.11. Dê três exemplos de projetos de software que seriam apropriados para o modelo baseado em componentes. Seja específico.

3.12. É possível provar que um componente de software, e mesmo um programa todo, está correto? Então, por que nem todo mundo faz isso?

3.13. Discuta o significado de "preocupações transversais" com suas palavras. A literatura sobre o POA está se expandindo rapidamente. Pesquise e escreva um breve trabalho sobre o estado-da-arte atual.

3.14. Processo Unificado e UML são a mesma coisa? Explique sua resposta.

3.15. Qual é a diferença entre uma fase do PU e um fluxo de trabalho do PU?

Leituras e Fontes de Informação Adicionais

A maioria dos livros-texto de engenharia de software considera os modelos prescritivos de processo em algum detalhe. Os livros de Sommerville (*Software Engineering*, 6ª edição, Addison-Wesley, 2000), Pflegeer (*Software Engineering: Theory and Practice*, Prentice-Hall, 2001), e Schach (*Object-Oriented and Classical Software Engineering*, McGraw-Hill, 2001) consideram os paradigmas convencionais e discutem seus pontos fortes e fracos. Apesar de não ser especificamente dedicado ao processo, Brooks (*The Mythical Man-Month*, 2ª edição, Addison-Wesley,1995) apresenta a antiga sabedoria de projeto que tem tudo a ver com processo. Firesmith e Henderson-Sellers (*The OPEN Process Framework: An Introduction*, Addison-Wesley, 2001) apresentam um gabarito geral para a criação de "*processos de software flexíveis e disciplinados*" e discutem os objetivos e atributos do processo.

Sharpe e McDermott (*Workflow Modeling: Tools for Process Improvement and Applications Development*, Artech House, 2001) apresentam ferramentas para modelagem de processos de software e de negócios. Jacobson, Griss e Jonsson (*Software Reuse*, Addison-Wesley, 1997) e McClure (*Software Reuse Techniques*, Prentice-Hall, 1997) apresentam informações bastante úteis sobre o desenvolvimento baseado em componentes. Heineman e Council (*Component-Based Software Engineering*, Addison-Wesley, 2001) descrevem o processo requerido para implementar sistemas baseados em componentes. Kenett e Baker (*Software Process Quality: Management and Control*, Marcel Dekker, 1999) consideram como a gestão de qualidade e o projeto de processo estão intimamente ligados um ao outro.

Ambriola (*Software Process Technology*, Springer-Verlag, 2001), Derniame e seus colegas (*Software Process: Principles, Methodology and Technology*, Springer-Verlag, 1999) e Gruhn e Hartmanis (*Software Process Technology*, Springer-Verlag, 1999) apresentam anais editados de conferências que abordam muitas questões de pesquisa e teóricas que são relevantes para o processo de software.

Jacobson, Booch e Rumbaugh escreveram um livro pioneiro sobre o Processo Unificado [JAC99]. No entanto, os livros de Arlow e Neustadt [ARL02] e uma série de três volumes de Ambler e Constantine [AMB02] fornecem excelente informação complementar. Krutchen (*The Rational Unified Process*, 2ª edição, Addison-Wesley, 2000) escreveu uma boa introdução sobre o PU. A gestão de projeto dentro do contexto do PU é descrita em detalhes por Royce (*Software Project Management: A Unified Framework*, Addison-Wesley,1998). A descrição definitiva do PU tem sido desenvolvida pela Rational Corporation e está disponível on-line em *www.rational.com*.

Uma grande variedade de fontes de informação sobre engenharia de software e processos de software está disponível na Internet. Uma lista atualizada de referências da World Wide Web que são relevantes para o processo de software pode ser encontrada na página Web do SEPA: **http://www.mhhe.com/presman**.

Document shared on www.docsity.com

CAPÍTULO 4

DESENVOLVIMENTO ÁGIL

CONCEITOS-CHAVE

agilidade 59
características da equipe ... 62
Crystal 71
DAS (ASD) 66
DSDM 68
Extreme Programming 63
FDD 71
manifesto ágil 58
modelagem ágil 72
modelos ágeis de processo .. 63
política 61
princípios de agilidade 60
programação aos pares 65
refabricação (*refactoring*) .. 65
Scrum 69

Em 2001, Kent Beck e 16 outros notáveis desenvolvedores, produtores e consultores de software [BEC01a] (conhecidos como a "Aliança Ágil") assinaram o "Manifesto para o Desenvolvimento Ágil de Software". Eles declararam:

> Estamos descobrindo melhores modos de desenvolvimento de software fazendo-o e ajudando outros a fazê-lo. Por meio desse trabalho passamos a valorizar:
>
> *Indivíduos e interações* em vez de processos e ferramentas.
>
> *Softwares funcionando* em vez de documentação abrangente.
>
> *Colaboração do cliente* em vez de negociação de contratos.
>
> *Resposta a modificações* em vez de seguir um plano.
>
> Isto é, ainda que haja valor nos itens à direita, valorizamos mais os itens à esquerda.

Um manifesto é normalmente associado com um movimento político emergente — um movimento que ataque a velha guarda e sugira mudanças revolucionárias (espera-se que seja para melhor). De algum modo, isso é exatamente o que o desenvolvimento ágil é.

Embora as idéias subjacentes que guiam o desenvolvimento ágil tenham estado conosco há muitos anos, somente durante a década de 1990 é que essas idéias foram cristalizadas em um "movimento". Em essência, os métodos ágeis[1] foram desenvolvidos em um esforço para vencer as fraquezas percebidas e reais da engenharia de software convencional. O desenvolvimento ágil pode fornecer importantes benefícios, mas ele não é aplicável a todos os projetos, produtos, pessoas e situações. Ele também *não é* contrário à sólida prática de engenharia de software e pode ser aplicado como uma filosofia prevalecente a todo o trabalho de software.

PANORAMA

O que é? A engenharia de software ágil combina uma filosofia e um conjunto de diretrizes de desenvolvimento. A filosofia encoraja a satisfação do cliente e a entrega incremental do software logo de início; equipes de projeto pequenas, altamente motivadas; métodos informais; produtos de trabalho de engenharia de soft-ware mínimos e simplicidade global do desenvolvimento. As diretrizes de desenvolvimento enfatizam a entrega em contraposição à análise e ao projeto (apesar dessas atividades não serem desencorajadas) e a comunicação ativa e contínua entre desenvolvedores e clientes.

Quem faz? Engenheiros de software e outros interessados no projeto (gerentes, clientes e usuários finais) trabalham juntos em uma equipe ágil — uma equipe que é auto-organizada e controla seu próprio destino. Uma equipe ágil enfatiza a comunicação e a colaboração entre todos os que a compõem.

Por que é importante? O ambiente moderno de negócios que cria sistemas baseados em computador e produtos de software é apressado e sempre mutável. A engenharia ágil de software representa uma alternativa razoável para a engenharia de software convencional para certas categorias de software e certos tipos de projeto de software. Tem sido demonstrado que ela entrega rapidamente sistemas bem-sucedidos.

Quais são os passos? O desenvolvimento ágil poderia ser melhor denominado "pequena engenharia de software". As atividades básicas de arcabouço — comunicação com o cliente, o planejamento, a modelagem, construção, entrega e avaliação — permanecem. Mas elas são reduzidas a um conjunto mínimo de tarefas que leva a equipe de projeto à construção e entrega (alguns alegariam que isso é feito à custa da análise do problema e projeto da solução).

Qual é o produto do trabalho? Clientes e engenheiros de software que têm adotado a filosofia ágil têm a mesma impressão — o único produto de trabalho realmente importante é um "incremento de software" operacional que é entregue ao cliente na data de entrega combinada.

Como tenho certeza de que fiz corretamente? Se a equipe ágil concordar que o processo funciona e produzir incrementos de software em condições de serem entregues e que satisfaçam ao cliente, você fez corretamente.

1 Os métodos ágeis são algumas vezes chamados de métodos *leves* ou *magros*.

Na economia moderna, é freqüentemente difícil ou impossível prever como um sistema baseado em computador (por exemplo, uma aplicação com base na Web) evoluirá com o passar do tempo. Condições de mercado mudam rapidamente, necessidades dos usuários finais evoluem e novas ameaças de competição emergem sem alerta. Em muitas situações, não podemos mais definir completamente os requisitos antes do início do projeto. Os engenheiros de software devem ser ágeis o suficiente para responder a um ambiente de negócios mutante.

Isso significa que um reconhecimento dessas causas realísticas modernas nos obrigam a descartar princípios, conceitos, métodos e ferramentas valiosos de engenharia de software? Certamente não! Como todas as disciplinas de engenharia, a engenharia de software continua a evoluir. Ela pode ser adaptada facilmente para encarar os desafios colocados pela demanda por agilidade.

> "Agilidade: 1, tudo o mais: 0."
>
> Tom DeMarco

Em um livro intrigante sobre o desenvolvimento ágil de softwares, Alistair Cockburn [COC02a] argumenta que os modelos prescritivos de processo introduzidos no Capítulo 3 têm uma deficiência importante: *eles esquecem as fragilidade das pessoas que constroem softwares de computador.* Engenheiros de software não são robôs. Eles exibem grande variedade de estilos de trabalho e diferenças significativas em nível de habilidade, criatividade, regularidade, consistência e espontaneidade. Alguns se comunicam bem na forma escrita, outros não. Cockburn argumenta que os modelos de processo podem "tratar as fraquezas comuns das pessoas com [ou] disciplina ou tolerância" [COCO2a], e que a maioria dos modelos prescritivos de processo escolhe a disciplina. Ele afirma: "Como a consistência em ação é uma fraqueza humana, metodologias de alta disciplina são frágeis" [COC02a].

Se os modelos de processo têm de funcionar, eles precisam fornecer um mecanismo realístico para encorajar a disciplina necessária, ou precisam ser caracterizados de um modo que mostre "tolerância" com as pessoas que fazem o trabalho de engenharia de software. Invariavelmente, práticas tolerantes são mais fáceis de serem adotadas e sustentadas pelo pessoal de software, mas (como Cockburn admite) elas podem ser menos produtivas. Como a maioria das coisas na vida, negociações devem ser consideradas.

4.1 O QUE É AGILIDADE?

O que é exatamente agilidade no contexto do trabalho de engenharia de software? Ivar Jacobson [JAC02] fornece uma discussão útil:

> *Agilidade* tornou-se atualmente uma palavra mágica quando se descreve um processo moderno de software. Tudo é ágil. Uma equipe ágil é uma equipe esperta, capaz de responder adequadamente a modificações. Modificação é aquilo para o qual o desenvolvimento de software está principalmente focado. Modificações no software que está sendo construído, modificações nos membros da equipe, modificações por causa de novas tecnologias, modificações de todas as espécies que podem ter impacto no produto que eles constroem ou no projeto que cria o produto. O apoio para modificações deveria ser incorporado em tudo que fazemos em software, algo que se adota porque está no coração e na alma do software. Uma equipe ágil reconhece que o software é desenvolvido por indivíduos trabalhando em equipes e que as especialidades dessas pessoas e sua capacidade de colaborar estão no âmago do sucesso do projeto.

Na visão de Jacobson, o acolhimento de modificações é o principal guia para a agilidade. Os engenheiros de software devem reagir rapidamente se tiverem de acomodar as rápidas modificações que Jacobson descreve.

Document shared on www.docsity.com

> "A agilidade é dinâmica, específica em conteúdo, agressiva no acolhimento de modificações e orientada ao crescimento."
>
> **Steven Goldman *et al.***

AVISO

Não cometa o erro de considerar que a agilidade lhe dá licença de improvisar soluções. Um processo é necessário e disciplina é essencial.

Mas a agilidade é mais do que uma resposta efetiva à modificação. Ela também engloba a filosofia apresentada no manifesto visto no início deste capítulo. Encoraja estruturas e atitudes de equipe que tornam a comunicação mais fácil (entre membros da equipe, entre pessoal de tecnologia e de negócios, entre engenheiros de software e seus gerentes). Ela enfatiza a rápida entrega de software operacional e dá menos importância para produtos de trabalho intermediários (nem sempre uma boa coisa); adota os clientes como parte da equipe de desenvolvimento e trabalha para eliminar a atitude "nós e eles" que continua a permear muitos projetos de software; ela reconhece que o planejamento em um mundo incerto tem seus limites e que um plano de projeto deve ser flexível.

A Aliança Ágil [AGI03] define 12 princípios para aqueles que querem alcançar agilidade:

1. Nossa maior prioridade é satisfazer ao cliente desde o início por meio de entrega contínua de software valioso.
2. Modificações de requisitos são bem-vindas, mesmo que tardias no desenvolvimento. Os processos ágeis aproveitam as modificações como vantagens para a competitividade do cliente.
3. Entrega de softwares funcionando freqüentemente, a cada duas semanas até dois meses, de preferência no menor espaço de tempo.
4. O pessoal de negócio e os desenvolvedores devem trabalhar juntos diariamente durante todo o projeto.
5. Construção de projetos em torno de indivíduos motivados. Forneça-lhes o ambiente e apoio que precisam e confie que eles farão o trabalho.
6. O método mais eficiente e efetivo de levar informação para e dentro de uma equipe de desenvolvimento é a conversa face a face.
7. Software funcionando é a principal medida de progresso.
8. Processos ágeis promovem desenvolvimento sustentável. Os patrocinadores, desenvolvedores e usuários devem ser capazes de manter um ritmo constante, indefinidamente.
9. Atenção contínua à excelência técnica e ao bom projeto facilitam a agilidade.
10. Simplicidade — a arte de maximizar a quantidade de trabalho não efetuado — é essencial.
11. As melhores arquiteturas, requisitos e projetos surgem de equipes auto-organizadas.
12. Em intervalos regulares, a equipe reflete sobre como se tornar mais efetiva, então sintoniza e ajusta adequadamente seu comportamento.

A agilidade pode ser aplicada a qualquer processo de software. Entretanto, para conseguir isso, é essencial que o processo seja projetado de modo que permita à equipe de projeto adaptar tarefas e aperfeiçoá-las, conduzir o planejamento para que se entenda a fluidez de uma abordagem de desenvolvimento ágil, eliminar tudo menos os produtos de trabalho mais essenciais e mantê-los simples, e enfatizar uma estratégia de entrega incremental que forneça o software funcionando ao cliente o mais rápido possível para o tipo de produto e ambiente operacional.

4.2 O que É um Processo Ágil?

Qualquer *processo ágil de software* é caracterizado de modo que atenda a três suposições-chave [FOW02] sobre a maioria dos projetos de software:

1. É difícil prever antecipadamente quais requisitos de software vão persistir e quais serão modificados. É igualmente difícil prever como as prioridades do cliente serão modificadas à medida que o projeto prossegue.
2. Para muitos tipos de software, o projeto e a construção são intercalados, isto é, as duas atividades devem ser realizadas juntas de modo que os modelos de projeto sejam comprovados à medida que são criados. É difícil prever o quanto de projeto é necessário antes que a construção seja usada para comprovar o projeto.

3. Análise, projeto, construção e testes não são tão previsíveis (do ponto de vista do planejamento) como gostaríamos.

Veja na Web

Uma coleção abrangente de artigos sobre o processo ágil pode ser encontrada em www.aanpo.org/articles/index.

Dadas essas três suposições, surge uma questão importante: como criar um processo que possa gerenciar a imprevisibilidade? A resposta, como já observamos, está na adaptabilidade do processo (as modificações rápidas do projeto e das condições técnicas). Um processo ágil, portanto, deve ser *adaptável*.

Mas a adaptação contínua sem progresso realiza pouco. Assim, um processo ágil de software deve ser adaptado *incrementalmente*. Para realizar a adaptação incremental, uma equipe ágil requer o feedback do cliente (de modo que adaptações apropriadas possam ser feitas). Um catalisador efetivo para o feedback do cliente é um protótipo operacional ou uma porção do sistema operacional. Assim, uma *estratégia de desenvolvimento incremental* deve ser instituída. *Incrementos de software* (protótipos executáveis ou partes um sistema operacional) devem ser entregues em curtos períodos de tempo de modo que a adaptação acerte o passo com as modificações (imprevisibilidade). Essa abordagem iterativa habilita o cliente a avaliar o incremento de software regularmente, fornecer o feedback necessário à equipe de software e influenciar as adaptações do processo feitas para acomodar o feedback.

> "Não há substituto para o feedback rápido, tanto do processo de desenvolvimento quanto no produto em si."
>
> **Martin Fowler**

4.2.1 A Política de Desenvolvimento Ágil

Há um considerável debate (algumas vezes estridente) sobre os benefícios e a aplicabilidade do desenvolvimento ágil de software em contraposição aos processos mais convencionais de engenharia de software. Jim Highsmith [HIG02a] descreve (jocosamente) os extremos ao caracterizar o sentimento do campo pró-agilidade ("agilistas"): "Os metodologistas tradicionais são um punhado de bitolados que preferem produzir documentação perfeita a um sistema funcionando que satisfaça às necessidades do negócio". Em contrapartida, ele descreve (novamente de maneira jocosa) a posição do campo de engenharia de software tradicional: "Os metodologistas levianos, quer dizer, ´ágeis´ são um punhado de gloriosos *hackers* que terão uma grande surpresa quando tiverem de ampliar seus brinquedos para chegar a um software que abranja toda a empresa".

Como toda argumentação sobre tecnologia de software, esse debate metodológico arrisca-se a degenerar em uma guerra religiosa. Se a guerra tiver início, o pensamento racional desaparecerá e crenças, em vez de fatos, orientarão a tomada de decisão.

Ninguém é contra a agilidade. A verdadeira questão é: qual é o melhor modo de alcançá-la? Outra questão tão importante quanto essa é: como construir softwares que satisfaçam às necessidades do cliente atual e exiba características de qualidade que lhe permitam ser estendido e ampliado para satisfazer às necessidades do cliente no longo prazo?

AVISO

Você não tem que escolher entre agilidade e engenharia de software. Em vez disso, defina uma abordagem de engenharia de software que seja ágil.

Não existem respostas absolutas para qualquer uma dessas questões. Mesmo dentro da escola ágil, há muitos modelos de processo propostos (Seção 4.3), cada qual com uma abordagem sutilmente diferente para o problema da agilidade. Em cada modelo há um conjunto de "idéias" (agilistas são contrários a chamá-las de "tarefas de trabalho") que representam um afastamento significativo da engenharia de software convencional. No entanto, muitos conceitos ágeis são simples adaptações de bons conceitos de engenharia de software. Conclusão: há muito a ser ganho considerando o melhor de ambas as escolas, e quase nada a ser ganho denegrindo qualquer uma dessas abordagens.

O leitor interessado deve ver em [HIG01], [HIG02a] e [DEM02] um resumo interessante sobre importantes tópicos políticos e técnicos.

4.2.2 Fatores Humanos

Os proponentes do desenvolvimento ágil de software sofrem muito para enfatizar a importância dos "fatores pessoais" no desenvolvimento ágil bem-sucedido. Como Cockburn e Highsmith

Document shared on www.docsity.com

[COC01] declaram, "o desenvolvimento ágil enfoca os talentos e habilidades dos indivíduos moldando o processo a pessoas e equipes específicas". O ponto-chave dessa declaração é que o processo se molda às necessidades das pessoas e da equipe, e não o contrário.[2]

> "O que é considerado meramente suficiente por uma equipe é ou suficiente demais ou insuficiente para outra."
>
> Alistair Cockburn

Se os membros de uma equipe de software tiverem de estabelecer as características do processo que é aplicado para construir softwares, uma certa quantidade de características-chave deve existir entre as pessoas de uma equipe ágil e na equipe em si.

? Quais são as características-chave do pessoal de uma equipe de software?

Competência. Em um contexto de desenvolvimento ágil (bem como em um contexto de engenharia de software convencional), "competência" inclui talento inato, habilidades específicas relacionadas a software e conhecimento global do processo que a equipe decidiu aplicar. Habilidade e conhecimento do processo podem e devem ser ensinados a todas as pessoas que servem como membros de equipes ágeis.

Foco comum. Embora os membros da equipe ágil possam realizar diferentes tarefas e trazer diferentes habilidades ao projeto, todos deveriam estar focados em uma meta — entregar um incremento de software em funcionamento ao cliente dentro do prazo prometido. Para atingir essa meta, a equipe também vai concentrar-se em adaptações contínuas (pequenas e grandes) que farão o processo satisfazer às necessidades da equipe.

Colaboração. Engenharia de software (independentemente do processo) diz respeito a avaliar, analisar e usar as informações que são comunicadas à equipe de software; criar informações que ajudarão o cliente e outros a entender o trabalho da equipe; e construir informações (software de computador e banco de dados relevantes) que forneçam valor de negócio para o cliente. Para realizar essas tarefas os membros da equipe precisam colaborar — uns com os outros, com o cliente e com os gerentes do negócio.

Capacidade de tomada de decisão. Qualquer boa equipe de software (inclusive equipes ágeis) deve ter a liberdade de controlar o seu próprio destino. Isso implica que seja dada autonomia à equipe — autoridade para tomada de decisão sobre tópicos técnicos e de projeto.

Habilidade de resolver problemas vagos. Os gerentes de software devem reconhecer que uma equipe ágil terá de lidar continuamente com ambigüidades e será continuamente confrontada por modificações. Em alguns casos a equipe precisa aceitar o fato de que o problema que está sendo resolvido hoje pode não ser o problema que precisará ser resolvido amanhã. No entanto, as lições aprendidas de qualquer atividade de solução de problema (inclusive aquelas que resolvem o problema errado) podem ser benéficas para a equipe mais adiante no projeto.

Respeito e confiança mútua. A equipe ágil deve tornar-se o que DeMarco e Lister [DEM98] chamam de equipe "consolidada" (veja o Capítulo 21). Uma equipe consolidada exibe a confiança e o respeito necessários para torná-la "tão fortemente aglutinada que o todo é maior que a soma das partes" [DEM98].

PONTO CHAVE

Uma equipe auto-organizada está no controle do trabalho que realiza. A equipe estabelece os seus próprios compromissos e define os planos para cumpri-los.

Auto-organização. No contexto de desenvolvimento ágil, a *auto-organização* implica três coisas: (1) a equipe ágil organiza-se para o trabalho a ser feito; (2) a equipe organiza o processo para melhor acomodar seu ambiente local; (3) a equipe organiza o cronograma de trabalho para conseguir melhor entrega do incremento de software. A auto-organização tem um certo número de benefícios técnicos, porém, mais importante que isso, ela serve para aperfeiçoar a colaboração e aumentar o moral da equipe. Em essência, a equipe serve como sua própria gerência. Ken Schwaber [SCH02] trata desses tópicos quando escreve: "A equipe seleciona quanto trabalho acredita que pode realizar dentro da iteração e a equipe se compromete com o trabalho. Nada desmotiva tanto uma equipe quanto alguém de fora assumir compromissos por ela. Nada motiva tanto uma equipe quanto a aceitação da responsabilidade de cumprir os compromissos que ela própria estabeleceu".

2 As organizações de engenharia de software mais bem-sucedidas reconhecem essa realidade independentemente do modelo de processo que escolhem.

4.3 Modelos Ágeis de Processo

A história da engenharia de software está congestionada com dúzias de descrições e metodologias obsoletas de processo, métodos de modelagem e notações, ferramentas e tecnologias. Cada uma delas ganhou notoriedade e foi depois eclipsada por algo mais novo e (pretensamente) melhor. Com a introdução de uma ampla gama de modelos ágeis de processo — cada qual lutando por aceitação na comunidade de desenvolvimento de software — o movimento ágil está seguindo o mesmo caminho histórico.[3]

> "A nossa profissão lida com metodologias como um adolescente de 14 anos lida com roupas."
>
> Stephen Hawrysh e Jim Ruprecht

Nas seções seguintes, apresentamos um panorama de alguns diferentes *modelos ágeis de processo*. Há muitas semelhanças (em filosofia e prática) entre essas abordagens. Nossa intenção é enfatizar as características de cada método que o tornam singular. É importante notar que *todos* os modelos ágeis satisfazem (em maior ou menor grau) ao *Manifesto para o Desenvolvimento Ágil de Software* e aos princípios mencionados na Seção 4.1.

4.3.1 Extreme Programming (XP)

Apesar do trabalho inicial sobre as idéias e métodos associados ao *Extreme Programming (XP)* ter ocorrido durante o final da década de 1980, o trabalho pioneiro sobre o assunto, escrito por Kent Beck [BEC99], foi publicado em 1999. Livros subseqüentes por Jeffries *et al.* [JEF01] sobre os detalhes técnicos do XP, e trabalho adicional por Beck e Fowler [BEC01b] sobre planejamento XP incorporam os detalhes ao método.

Veja na Web

Um excelente panorama de "regras" para XP pode ser encontrado em www.extremeprogramming.org/rules.html.

O XP usa uma abordagem orientada a objetos (Parte 2 deste livro) como seu paradigma de desenvolvimento predileto. O XP inclui um conjunto de regras e práticas que ocorrem no contexto de quatro atividades de arcabouço: planejamento, projeto, codificação e teste. A Figura 4.1 ilustra o processo XP e mostra algumas das idéias-chave e tarefas que estão associadas a cada atividade de arcabouço. As atividades-chave do XP são resumidas nos parágrafos seguintes.

Planejamento. A atividade de planejamento começa com a criação de um conjunto de *histórias* (também chamado de *histórias de usuário*) que descrevem as características e funcionalidades requeridas para o software a ser construído. Cada história (semelhante aos casos de uso descritos nos Capítulos 7 e 8) é escrita pelo cliente e é colocada em um cartão de indexação. O cliente atribui um *valor* (isto é, uma prioridade) para a história, com base no valor de negócio global da característica ou da função.[4] Membros da equipe XP avaliam então cada história e lhe atribuem um *custo* — medido em semanas de desenvolvimento. Se a história precisar mais do que três semanas de desenvolvimento, pede-se ao cliente para dividir a história em histórias menores e a atribuição de valor e custo ocorre novamente. É importante notar que novas histórias podem ser escritas a qualquer momento.

? O que é uma "história" XP?

Os clientes e a equipe XP trabalham juntos para decidir como agrupar histórias na versão seguinte (o próximo incremento de software) a ser desenvolvida pela equipe XP. Uma vez feito um *compromisso* básico para a versão (um acordo quanto às histórias a ser incluídas, data de entrega e outros assuntos de projeto), a equipe XP determina as histórias que serão desenvolvidas em um dos três modos a seguir: (1) todas as histórias serão implementadas imediatamente (dentro de poucas semanas); (2) as histórias com valor mais alto serão antecipadas no cronograma e implementadas primeiro; ou (3) as histórias de maior risco serão antecipadas no cronograma e implementadas primeiro.

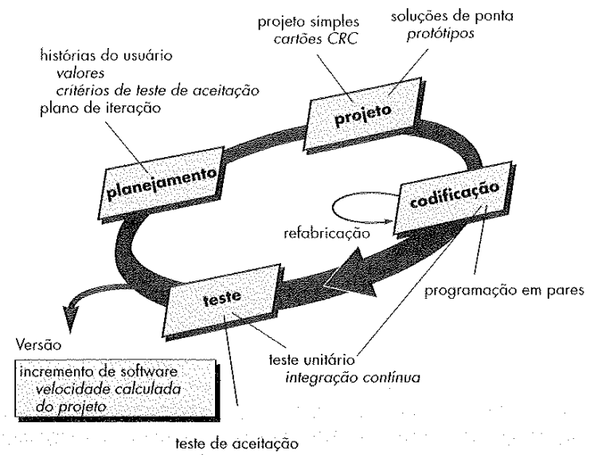
3 Isso não é uma coisa ruim. Antes de um ou mais modelos ou métodos serem aceitos como uma norma de fato, todos precisam lutar pelos corações e mentes dos engenheiros de software. Os "vencedores" evoluem para a melhor prática, enquanto os "perdedores" desaparecem ou se combinam com os modelos vencedores.

4 O valor de uma história pode também depender da presença de outra história.

Document shared on www.docsity.com

FIGURA 4.1

O Processo Extreme Programming



Veja na Web
Um "jogo de planejamento" XP de valor pode ser encontrado em c2.com/cgi/wiki?planningGame.

Depois que a primeira versão do projeto (também chamada de incremento de software) tiver sido entregue, a equipe XP calcula a velocidade do projeto. Dito simplesmente, *velocidade de projeto* é a quantidade de histórias do cliente implementadas durante a primeira versão. A velocidade do projeto pode ser usada para (1) ajudar a estimar as datas de entrega e o cronograma para as versões subseqüentes e (2) determinar se um comprometimento excessivo foi feito para todas as histórias ao longo de todo o projeto de desenvolvimento. Se um comprometimento excessivo ocorrer, o conteúdo das versões será modificado ou as datas de entrega finais são alteradas.

À medida que o trabalho de desenvolvimento prossegue, o cliente pode adicionar histórias, mudar o valor de uma história existente, subdividir histórias e eliminá-las. A equipe XP então reconsidera todas as versões remanescentes e modifica os seus planos adequadamente.

> "Extreme Programming é uma disciplina de desenvolvimento de software baseada em valores de simplicidade, comunicação, feedback e coragem."
>
> **Ron Jeffries**

Projeto. O projeto XP segue rigorosamente o princípio KIS (*keep it simple* — mantenha a simplicidade). Um projeto simples é sempre preferível em relação a uma representação mais complexa. Além disso, o projeto fornece diretrizes de implementação para uma história como ela está escrita — nada mais, nada menos. O projeto de funcionalidade extra (porque o desenvolvedor considera que ela será exigida depois) é desencorajado.[5]

O XP encoraja o uso de cartões CRC (Capítulo 8) como um mecanismo efetivo para raciocinar sobre o software no contexto orientado a objetos. Os cartões CRC (Class-Responsibility-Colaborator) identificam e organizam as classes orientadas a objetos[6] que são relevantes para o incremento de software atual. A equipe XP conduz o exercício do projeto usando um processo semelhante ao que está descrito no Capítulo 8, Seção 8.7.4. Os cartões CRC são o único produto de trabalho do projeto que é realizado como parte do processo XP.

Se um problema de projeto difícil é encontrado como parte do projeto de uma história, o XP recomenda a criação imediata de um protótipo operacional daquela parte do projeto. Denominado *solução de ponta*, o protótipo de projeto é implementado e avaliado. A intenção é diminuir o risco

5 Essas diretrizes de projeto devem ser seguidas em todo método de engenharia de software, ainda que haja ocasiões em que notação e terminologia sofisticadas de projeto podem ficar no caminho da simplicidade.

6 As classes orientadas a objetos serão discutidas em detalhes no Capítulo 8 e ao longo da Parte 2 deste livro.

Veja na Web
Técnicas e ferramentas de refabricação podem ser encontradas em www.refactoring.com.

quando a implementação verdadeira começa e validar as estimativas originais correspondentes à história que contém o problema de projeto.

O XP encoraja a *refabricação* — uma técnica de construção que é também uma técnica de projeto. Fowler [FOW00] define refabricação do seguinte modo:

> Refabricação é o processo de modificar um sistema de software de tal modo que ele não altere o comportamento externo do código, mas aperfeiçoe a estrutura interna. É um modo disciplinado de limpar o código [e modificar/simplificar o projeto interno] que minimiza as chances de introdução de defeitos. Em essência, quando você refabrica está aperfeiçoando o projeto do código depois que ele foi escrito.

Como o projeto XP praticamente não usa nenhuma notação e produz poucos, ou nenhum, produto de trabalho que não sejam os cartões CRC e as soluções de ponta, o projeto é visto como um artefato provisório que pode e deve ser continuamente modificado à medida que a construção prossegue. A intenção da refabricação é controlar essas modificações sugerindo pequenas alterações de projeto que "podem aperfeiçoar radicalmente o projeto" [FOW00]. Deve-se notar, no entanto, que o esforço necessário à refabricação pode crescer sensivelmente à medida que o tamanho de uma aplicação cresce.

Veja na Web
Informações úteis sobre o XP podem ser encontradas em www.xprogramming.com.

Uma noção central no XP é de que o projeto ocorre tanto antes quanto *depois* que a codificação começa. Refabricação significa que o projeto ocorre continuamente à medida que o sistema é construído. De fato, a atividade de construção em si vai fornecer à equipe XP diretrizes sobre como aperfeiçoar o projeto.

Codificação. O XP recomenda que depois que as histórias forem desenvolvidas e o trabalho preliminar de projeto for feito, a equipe não avance para o código mas, em vez disso, desenvolva uma série de testes unitários que exercitarão cada uma das histórias que devem ser incluídas na versão atual (incremento de software).[7] Uma vez criados os testes unitários, o desenvolvedor está melhor preparado para focalizar o que precisa ser implementado para passar no teste unitário. Nada estranho é adicionado (KIS). Uma vez completado o código, ele pode ser submetido imediatamente ao teste unitário, fornecendo assim feedback instantâneo para os desenvolvedores.

? O que é programação dos pares?

Um conceito-chave durante a atividade de codificação (e um dos aspectos mais comentados do XP) é a *programação em pares*. O XP recomenda que duas pessoas trabalhem juntas em uma estação de trabalho de computador para criar o código correspondente a uma história. Isso fornece um mecanismo de solução de problemas em tempo real (duas cabeças são freqüentemente melhores do que uma) e de garantia de qualidade em tempo real. Também mantém os desenvolvedores focados no problema em mãos. Na prática, cada pessoa assume um papel ligeiramente diferente. Por exemplo, uma pessoa poderia pensar nos detalhes de código de uma parte específica do projeto, enquanto a outra garante que as normas de codificação (parte necessária do XP) estão sendo seguidas e que o código gerado vai se "encaixar" no projeto mais amplo da história.

À medida que os pares de programadores completam seu trabalho, o código que eles desenvolvem é integrado ao trabalho de outros. Em alguns casos, isso é realizado diariamente por uma equipe de integração. Em outros casos, os pares de programadores têm responsabilidade de integração. Essa estratégia de "integração contínua" ajuda a evitar problemas de compatibilidade e interface e fornece um ambiente de "teste de fumaça" (Capítulo 13) que ajuda a descobrir rapidamente os erros.

Teste. Já mencionamos que a criação de um teste unitário[8] antes da codificação começar é um elemento-chave da abordagem XP. Os testes unitários que são criados devem ser implementados usando um arcabouço que lhes permita ser automatizados (conseqüentemente, eles podem ser executados fácil e repetidamente). Isso encoraja uma estratégia de teste de regressão (Capítulo 13) sempre que o código é modificado (o que é freqüente, considerando a filosofia de refabricação do XP).

7 Essa abordagem é análoga a saber as questões do exame antes de começar a estudar. Ela torna o estudo muito mais fácil focalizando a atenção apenas nas questões que serão formuladas.

8 O teste unitário, discutido em detalhes no Capítulo 13, focaliza um componente individual de software, exercitando a interface, as estruturas de dados e a funcionalidade do componente, em um esforço para descobrir erros que são locais para o componente.

Document shared on www.docsity.com

À medida que os testes unitários individuais são organizados em uma "seqüência universal de testes" [WEL99], o teste de integração e validação do sistema pode ocorrer diariamente. Isso fornece à equipe XP uma indicação contínua de progresso e também pode levantar sinais de alerta se as coisas estiverem se deteriorando. Wells [WEL99] afirma: "Resolver pequenos problemas a cada intervalo de umas poucas horas leva menos tempo do que resolver grandes problemas perto da data de entrega".

PONTO CHAVE

Os testes de aceitação XP derivam de histórias de usuário.

Os *testes de aceitação XP*, também chamados de *testes do cliente*, são especificados pelo cliente e focalizam as características e funcionalidades do sistema global que são visíveis e passíveis de revisão pelo cliente. Testes de aceitação são derivados das histórias do usuário que foram implementadas como parte de uma versão de software.

CASASEGURA

Considerando o Desenvolvimento Ágil de Softwares

A cena: escritório de Doug Miller.

Os personagens: Doug Miller, gerente de engenharia de software; Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software.

A conversa:

(Uma batida na porta)

Jamie: Doug, você tem um minuto?

Doug: Lógico, Jamie, o que há?

Jamie: Estivemos pensando em nossa discussão de processo de ontem... Você sabe, que processo vamos escolher para esse novo projeto *CasaSegura*.

Doug: E?

Vinod: Eu estava falando com um amigo de outra empresa, e ele me falou sobre Extreme Programming. É um modelo ágil de processo, você já ouviu falar?

Doug: Já, um pouco bem e um pouco mal.

Jamie: Bem, parece bastante bom para nós. Deixa a gente desenvolver software de modo realmente rápido, usa uma coisa chamada programação aos pares para fazer verificações de qualidade em tempo real... É bastante interessante, na minha opinião.

Doug: De fato, tem idéias realmente boas. Eu gosto do conceito de "programação aos pares", por exemplo, e da idéia de que os interessados devam fazer parte da equipe.

Jamie: Anh? Você quer dizer que o pessoal de marketing vai trabalhar na equipe de projeto conosco?

Doug (concordando): Eles são os interessados, não são?

Jamie: É mas... eles vão ficar fazendo pedidos de modificação a cada cinco minutos.

Vinod: Não necessariamente. Meu amigo disse que existem modos de "acolher" modificações durante um projeto XP.

Doug: Então vocês pensam que nós poderíamos usar o XP?

Jamie: Definitivamente, vale a pena considerar.

Doug: Concordo. E, mesmo que escolhamos um modelo incremental para a nossa abordagem, não há por que não possamos incorporar muito do que o XP tem a oferecer.

Vinod: Doug, antes você disse "um pouco bem e um pouco mal". O que era "mal"?

Doug: O que eu não gosto é o modo como o XP menospreza a análise e o projeto... parece dizer que é na escrita de código que a ação reside.

(Os membros da equipe se entreolham e sorriem.)

Doug: Então, você concorda com a abordagem XP?

Jamie (falando pelos dois): Escrever código é o que nós fazemos, chefe!

Doug (rindo): É verdade, mas eu gostaria de ver vocês gastando um pouco menos de tempo codificando e depois recodificando, e um pouco mais de tempo analisando o que tem de ser feito e projetando uma solução que funcione.

Vinod: Talvez nós possamos agir dos dois modos, agilidade com um pouco de disciplina.

Doug: Acredito que podemos, Vinod. Na verdade, estou certo disso.

4.3.2 DAS — Desenvolvimento Adaptativo de Software (Adaptative Software Development — ASD)

O DAS (*Desenvolvimento Adaptativo de Software*) foi proposto por Jim Highsmith [HIG00] como uma técnica para construção de sistemas e softwares complexos. O apoio filosófico do DAS concentra-se na colaboração humana e na auto-organização da equipe. Highsmith [HIG98] discute isso quando escreve:

> Auto-organização é uma propriedade de sistemas adaptativos complexos semelhantes a um "Ahhh" coletivo, aquele momento de energia criativa em que a solução para algum problema impertinente surge. A auto-organização aparece quando agentes individuais independentes (células em um corpo, espécies em um ecossistema, desenvolvedores em uma equipe) cooperam (colaboram) para criar resultados emergentes. Um resultado emergente é uma propriedade além da capacidade de qualquer agente individual. Por exemplo, neurônios individuais no cérebro não possuem conscientização, mas coletivamente a propriedade de conscientização emerge. Tendemos a considerar esse fenômeno de emergência coletiva como acidental ou, pelo menos, sem regra e não-confiável. O estudo da auto-organização está provando que essa consideração está errada.

Veja na Web

Recursos úteis para o DAS podem ser encontrados em www.adaptivesd.com.

Highsmith argumenta que uma abordagem de desenvolvimento ágil, adaptativa, baseada em colaboração é "tanto uma fonte de *ordem* em nossas interações complexas quanto a disciplina e a engenharia". Ele define um "ciclo de vida" DAS (Figura 4.2) que incorpora três fases: especulação, colaboração e aprendizado.

? Quais são as características dos ciclos adaptativos do DAS?

Especulação. Durante a *especulação, o projeto* é iniciado e o *planejamento do ciclo adaptativo* é conduzido. Planejamento do ciclo adaptativo usa informações de iniciação do projeto — a declaração de missão feita pelo cliente, as restrições do projeto (por exemplo, datas de entrega ou descrições de usuários) e requisitos básicos — para definir o conjunto de ciclos de versão (incrementos de software) que serão necessários para o projeto.[9]

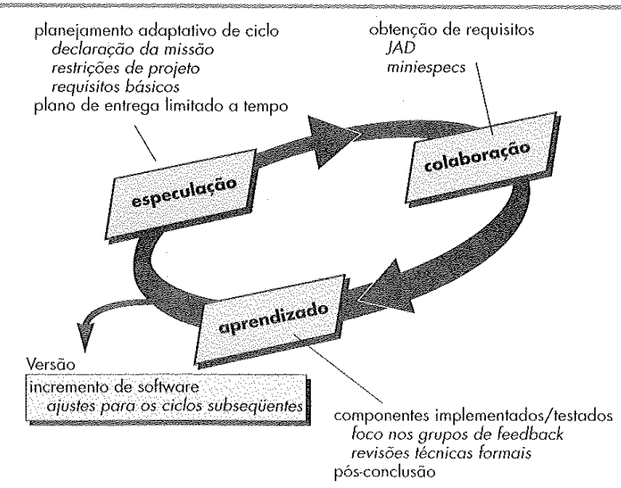
AVISO

Somente irá ocorrer colaboração efetiva com o seu cliente se você abandonar quaisquer atitudes "nós e eles".

Colaboração. Pessoal motivado trabalha junto de um modo que multiplica seus talentos e resultados criativos além de seu número absoluto. Essa abordagem colaborativa é um tema recorrente em todos os métodos ágeis. Mas a colaboração não é fácil. Ela não é apenas comunicação, embora a comunicação faça parte dela. Ela não é somente uma questão da equipe de trabalho, apesar de que uma equipe "aglutinada" (Capítulo 21) é essencial para que a colaboração realmente ocorra. Não é uma rejeição do individualismo, porque a criatividade individual desempenha um importante papel no raciocínio colaborativo. Ela é, acima de tudo, uma questão de confiança. Pessoas que trabalham juntas precisam confiar umas nas outras para (1) criticar sem animosidade; (2) ajudar sem ressentimentos; (3) trabalhar tão duro ou mais duro do que costumam; (4) ter o conjunto de habilidades para contribuir com o trabalho em mãos; e (5) comunicar problemas e/ou preocupações de um modo que conduza à ação efetiva.

FIGURA 4.2

Desenvolvimento adaptativo de softwares



9 Note que o plano do ciclo adaptativo pode e provavelmente será adaptado para modificar as condições do projeto e do negócio.

Document shared on www.docsity.com

> "Gosto de ouvir. Aprendi muito ouvindo cuidadosamente. A maioria das pessoas nunca ouve."
>
> **Ernest Hemingway**

Aprendizado. À medida que os membros de uma equipe DAS começam a desenvolver os componentes que fazem parte de um ciclo adaptativo, a ênfase está tanto no aprendizado quanto no progresso em direção a um ciclo completo. De fato, Highsmith [HIG00] alega que os desenvolvedores de software freqüentemente superestimam seu próprio entendimento (da tecnologia, do processo e do projeto) e que o aprendizado vai ajudá-los a melhorar seu nível de compreensão real. As equipes DAS aprendem de três modos:

1. **Foco nos grupos.** O cliente e/ou usuários finais fornecem feedback sobre os incrementos de software que são entregues indicando se o produto está ou não satisfazendo às necessidades do negócio.
2. **Revisões técnicas formais.** Os membros da equipe DAS revisam os componentes de software que são desenvolvidos, aperfeiçoando a qualidade e aprendendo à medida que prosseguem.
3. **Pós-conclusão.** A equipe DAS torna-se introspectiva, cuidando do seu próprio desempenho e processo (com a intenção de aprender e depois aperfeiçoar a sua abordagem).

É importante notar que a filosofia DAS tem mérito independentemente do modelo de processo que é usado. A ênfase global do DAS na dinâmica de equipes auto-organizadas, na colaboração interpessoal e no aprendizado individual e em equipe produz equipes de projeto de software que têm uma probabilidade de sucesso muito maior.

4.3.3 DSDM (Dynamic Systems Development Method — Método de Desenvolvimento Dinâmico de Sistemas)

Veja na Web
Recursos úteis para o DSDM podem ser encontrados em www.dsdm.org.

O *DSDM (Dynamic Systems Development Method)* [STA97] é uma abordagem ágil de desenvolvimento de software que "fornece um arcabouço para construir e manter sistemas que satisfazem às restrições de prazo apertadas por meio do uso de prototipagem incremental em um ambiente controlado de projeto" [CCS02]. Semelhante em alguns pontos ao processo RAD, discutido no Capítulo 3, o DSDM sugere uma filosofia que é emprestada de uma versão modificada do princípio de Pareto. Nesse caso, 80% de uma aplicação pode ser entregue em 20% do tempo que levaria para entregar a aplicação completa (100%).

Como o XP e o DAS, o DSDM sugere um processo iterativo de software. No entanto, a aplicação da abordagem DSDM a cada iteração segue a regra dos 80%, isto é, apenas um certo trabalho é necessário para que cada incremento facilite o avanço para o incremento seguinte. Os detalhes restantes podem ser completados depois, quando mais requisitos do negócio forem conhecidos ou quando modificações forem solicitadas e acomodadas.

O DSDM Consortium (*www.dsdm.org*) é um grupo mundial de empresas que assumiram, coletivamente, o papel de "zeladores" do método. O consórcio definiu um modelo ágil de processo, chamado de *ciclo de vida DSDM*. O ciclo de vida DSDM define três ciclos iterativos diferentes, precedidos por duas atividades adicionais de ciclo de vida:

Veja na Web
Um panorama útil do DSDM pode ser encontrado em www.cs3inc.com/DSDM.htm.

Estudo de viabilidade — estabelece os requisitos básicos e restrições do negócio associados à aplicação em construção e depois avalia se a aplicação é uma candidata viável ao processo DSDM.

Estudo do negócio — estabelece os requisitos funcionais e de informação que permitirão à aplicação fornecer valor de negócio; também define a arquitetura básica da aplicação e identifica os requisitos de manutenibilidade para a aplicação.

Iteração do modelo funcional — produz um conjunto de protótipos incrementais que demonstram a funcionalidade para o cliente (note que todos os protótipos DSDM são destinados a evoluir para a aplicação a ser entregue). O intuito durante esse ciclo iterativo é obter requisitos adicionais pela provocação de feedback dos usuários à medida que eles exercitam o protótipo.

Iteração de projeto e construção — revisita os protótipos construídos durante a iteração do modelo funcional para garantir que cada um tenha passado por engenharia, de modo que seja capaz de fornecer valor de negócio operacional para os usuários finais. Em alguns casos, a iteração do modelo funcional e a iteração de projeto e construção ocorrem simultaneamente.

Implementação — coloca o último incremento de software (um protótipo "operacionalizado") no ambiente operacional. Deve-se notar que: (1) o incremento pode não estar 100% completo ou (2) modificações podem ser solicitadas à medida que o incremento é colocado em ação. Em ambos os casos, o trabalho de desenvolvimento DSDM continua por meio do retorno à atividade de iteração do modelo funcional.

O DSDM pode ser combinado com o XP para fornecer uma abordagem combinada que defina um modelo de processo sólido (o ciclo de vida DSDM) com práticas instrumentais (XP) necessárias à construção de incrementos de software. Além disso, os conceitos de colaboração e equipes auto-organizadas do DAS podem ser adaptados para formar um modelo combinado de processo.

4.3.4 Scrum

O *Scrum* (o nome é derivado de uma atividade[10] que ocorre durante um jogo de *rugby*) é um modelo ágil de processo que foi desenvolvido por Jeff Sutherland e por sua equipe no início da década de 1990. Nos últimos anos foi realizado desenvolvimento adicional de métodos Scrum por Scwaber e Beedle [SCH01]. Os princípios Scrum [ADM96] são consistentes com o manifesto ágil:

- Pequenas equipes de trabalho são organizadas de modo a "maximizar a comunicação, minimizar a supervisão e maximizar o compartilhamento de conhecimento tácito informal".
- O processo precisa ser adaptável tanto a modificações técnicas quanto de negócios "para garantir que o melhor produto possível seja produzido".
- O processo produz freqüentes incrementos de software "que podem ser inspecionados, ajustados, testados, documentados e expandidos".
- O trabalho de desenvolvimento e o pessoal que o realiza é dividido "em partições claras, de baixo acoplamento, ou em pacotes".
- Testes e documentação constantes são realizados à medida que o produto é construído.
- O processo Scrum fornece a "habilidade de declarar o produto 'pronto' sempre que necessário (porque a concorrência acabou de entregar, porque a empresa precisa de dinheiro, porque o usuário/cliente precisa das funções, porque foi para essa data que foi prometido ...)" [ADM96].

Veja na Web
Informações e recursos úteis do Scrum podem ser encontrados em www.controlchaos.com.

Os princípios Scrum são usados para guiar as atividades de desenvolvimento dentro de um processo que incorpora as seguintes atividades de arcabouço: requisitos, análise, projeto, evolução e entrega. Em cada atividade de arcabouço, as tarefas de trabalho ocorrem dentro de um padrão de processo (discutido no parágrafo seguinte) chamado de *sprint*. O trabalho conduzido dentro de um *sprint* (a quantidade de *sprints* necessária para cada atividade de arcabouço varia dependendo da complexidade e do tamanho do produto) é adaptado ao problema em mãos e é definido e, freqüentemente, modificado em tempo real pela equipe Scrum. O fluxo global do processo é ilustrado na Figura 4.3.

> "O Scrum permite a construção de softwares mais flexíveis."
>
> **Mike Beetle et al.**

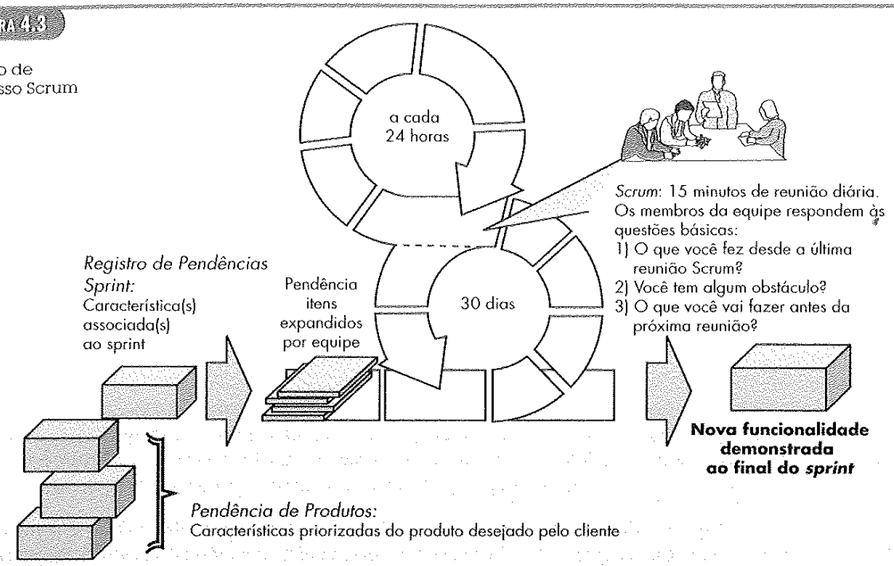
O Scrum enfatiza o uso de um conjunto de "padrões de processo de software" [NOI02] que se mostraram efetivos para projetos com prazos apertados, requisitos mutantes e criticalidade de negócio. Cada um desses padrões de processo define um conjunto de atividades de desenvolvimento:

10 Um grupo de jogadores se forma ao redor da bola e os companheiros de equipe trabalham juntos (algumas vezes violentamente!) para mover a bola pelo campo.

Document shared on www.docsity.com

FIGURA 4.3

O fluxo de processo Scrum



PONTO CHAVE

O Scrum incorpora um conjunto de padrões de processo que enfatiza prioridades de projeto, unidades de trabalho compartimentalizadas, comunicação e feedback freqüente do cliente.

Pendência — uma lista priorizada de requisitos ou características do projeto que fornecem valor de negócio para o cliente. Itens podem ser adicionados à pendência a qualquer momento (é assim que as modificações são introduzidas). O gerente de produto avalia a pendência e atualiza as prioridades quando necessário.

Sprints — consiste de unidades de trabalho que são necessárias para satisfazer a um requisito definido na pendência que precisa ser cumprido em um intervalo de tempo predefinido (tipicamente 30 dias). Durante o *sprint*, os itens em pendência a que as unidades de trabalho do *sprint* se destinam são congelados (isto é, não são introduzidas as modificações durante o sprint). Assim, o *sprint* permite que os membros da equipe trabalhem em um ambiente de curto prazo, mas estável.

Reuniões Scrum — são reuniões curtas (normalmente de 15 minutos) feitas diariamente pela equipe Scrum. Três questões-chave são formuladas e respondidas por todos os membros da equipe [NOY02]:

- O que você fez desde a última reunião de equipe?
- Que obstáculos você está encontrando?
- O que você planeja realizar até a próxima reunião de equipe?

Um líder de equipe, chamado de *Scrum master*, lidera a reunião e avalia as respostas de cada pessoa. Essas reuniões diárias ajudam a equipe a descobrir problemas potenciais tão cedo quanto possível. Elas levam também à "socialização do conhecimento" [BEE99] e promovem, assim, uma estrutura de equipe auto-organizada.

Demos — entrega o incremento de software ao cliente de modo que a funcionalidade implementada possa ser demonstrada e avaliada pelo cliente. É importante notar que a demo talvez não contenha toda a funcionalidade planejada, mas, em vez disso, as funções que podem ser entregues dentro do intervalo de tempo estabelecido.

Beedle e seus colegas [BEE99] apresentam uma discussão abrangente desses padrões na qual eles declaram: "O Scrum considera antecipadamente a existência do caos... "Os padrões de processo Scrum permitem a uma equipe de desenvolvimento de software trabalhar de modo bem-sucedido em um mundo em que a eliminação da incerteza é impossível.

4.3.5 Crystal

Veja na Web

Uma discussão abrangente do Crystal pode ser encontrada em www.crystalmethodologies.org.

Alistair Cockburn [COC02a] e Jim Highsmith [HIG02b] criaram a *família Crystal de métodos ágeis*,[11] a fim de conseguir uma abordagem de desenvolvimento de software que premia a "manobrabilidade" durante o que Cockburn caracteriza como "um jogo cooperativo de invenção e comunicação de recursos limitados, com o principal objetivo de entregar softwares úteis funcionando e com o objetivo secundário de preparar-se para o jogo seguinte" [COC02b].

Para conseguir a manobrabilidade, Cockburn e Highsmith definiram um conjunto de metodologias, cada qual com elementos centrais que são comuns a todas, e papéis, padrões de processos, produtos de trabalho e práticas específicas de cada uma. A família Crystal é, na verdade, um conjunto de processos ágeis que se mostraram efetivos para diferentes tipos de projeto. A intenção é permitir que equipes ágeis selecionem o membro da família Crystal mais apropriado para o seu projeto e ambiente.

4.3.6 FDD (Feature Driven Development — Desenvolvimento Guiado por Características)

Veja na Web

Uma grande variedade de artigos e apresentações sobre o FDD pode ser encontrada em www.thecoadletter.com.

O FDD (Desenvolvimento Guiado por Características) foi originalmente concebido por Peter Coad e seus colegas [COA99] como um modelo prático de processo para a engenharia de software orientada a objetos. Stephen Palmer e John Felsing [PAL02] estenderam e melhoraram o trabalho de Coad descrevendo um processo ágil e adaptativo que pode ser aplicado a projetos de software de tamanho moderado e grande.

No contexto do FDD, uma *característica* "é uma função valorizada pelo cliente que pode ser implementada em duas semanas ou menos" [COA99]. A ênfase na definição de características fornece os seguintes benefícios:

- Como as características são pequenos blocos de funcionalidade passíveis de entrega, os usuários podem descrevê-las mais facilmente, entender como elas se relacionam umas com as outras mais prontamente e revisá-las melhor quanto a ambigüidades, erros ou omissões.
- As características podem ser organizadas em um agrupamento hierárquico relacionado ao negócio.
- Como uma característica é um incremento de software passível de entrega do FDD, a equipe desenvolve características operacionais a cada duas semanas.
- Como as características são pequenas, suas representações de projeto e de código são mais fáceis de inspecionar efetivamente.
- Planejamento de projeto, cronogramação e monitoração são guiados pela hierarquia de características em vez de por um conjunto de tarefas de engenharia de software arbitrariamente adotado.

Coad e seus colegas [COA99] sugerem o seguinte gabarito para definir uma característica:

<ação> o **<resultado>** **<por | para | de | a>** um **<objeto>**

em que **<objeto>** é "uma pessoa, lugar ou coisa (inclusive papéis, momentos no tempo ou intervalos de tempo, ou descrições como entradas de catálogo)". Exemplos de características para uma aplicação de comércio eletrônico poderiam ser:

Adiciona o produto a um carrinho de compras.

Exibe as especificações técnicas de um produto.

Armazena as informações de remessa para um cliente.

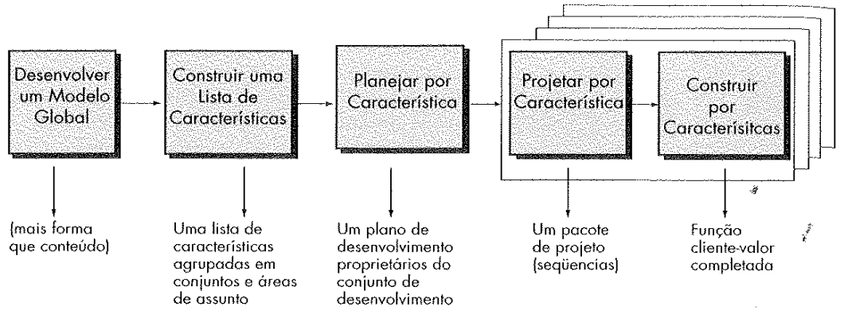
Um conjunto de características as agrupa em categorias relacionadas ao negócio e é definido [COA99] como:

11 O nome "crystal" é derivado das características de cristais geológicos, cada um com sua própria cor, forma e dureza.

Document shared on www.docsity.com

FIGURA 4.4

Desenvolvimento Guiado por Características (COA99) (usado com permissão)



<verbo no gerúndio (ação)> um **<objeto>**

Por exemplo: *Fazendo uma venda de produto* é um conjunto de características que incluiria as características mencionadas anteriormente e outras.

A abordagem FDD define cinco atividades de arcabouço "colaborativas" [COA99] (em FDD elas são denominadas "processos") como é mostrado na Figura 4.4.

O FDD coloca mais ênfase em diretrizes e técnicas de gestão de projeto do que muitos outros métodos ágeis. À medida que os projetos crescem em tamanho e complexidade, a gestão de projeto *ad hoc* é freqüentemente inadequada. É essencial que desenvolvedores, seus gerentes e o cliente entendam o estado do projeto — que avanços foram feitos e que problemas foram encontrados. Se a pressão quanto ao prazo de entrega for significativa, é crítico determinar se os incrementos de software (características) foram adequadamente cronogramados. Para conseguir isso, o FDD define seis marcos de referência durante o projeto e implementação de uma característica: "travessia do projeto, projeto, inspeção de projeto, código, inspeção de código, promoção para construção" [COA99].

4.3.7 Modelagem Ágil (Agile Modeling — AM)

Veja na Web

Informações abrangentes sobre a modelagem ágil podem ser encontradas em **www.agilemodeling.com**.

Há muitas situações em que os engenheiros de software precisam construir sistemas grandes, críticos para o negócio. O escopo e complexidade de tais sistemas devem ser modelados de modo que (1) todos os participantes possam entender melhor o que precisa ser realizado; (2) o problema possa ser particionado efetivamente entre o pessoal que deve resolvê-lo e (3) a qualidade possa ser avaliada a cada passo, à medida que o sistema passa por engenharia e é construído.

Nos últimos 30 anos, uma grande variedade de métodos e notação para modelagem de engenharia de software foi proposta para análise e projeto (tanto arquitetural quanto em nível de componente). Esses métodos têm mérito significativo, mas mostraram-se difíceis de aplicar e de manutenção desafiadora (ao longo de vários projetos). Parte do problema é o "peso" desses métodos de modelagem. Com isso, queremos dizer: o volume de notação exigido, o grau de formalismo sugerido, o tamanho dos modelos para projetos grandes e a dificuldade em manter o modelo à medida que as modificações ocorrem. No entanto, a modelagem de análise e de projeto apresenta benefícios substanciais para projetos grandes — se não por outra razão, pelo menos para tornar esses projetos intelectualmente possíveis de gerir. Há uma abordagem ágil para a modelagem de engenharia de software que poderia fornecer uma alternativa?

No site "The Official Agile Modeling Site", Scott Ambler [AMB02] descreve a *Modelagem Ágil* (AM) do seguinte modo:

> A Modelagem Ágil (AM) é uma metodologia baseada na prática, para modelagem e documentação efetiva de sistemas baseados em software. Colocado simplesmente, Modelagem Ágil é uma coleção de valores, princípios e práticas de modelagem de software que podem ser aplicados a um projeto de desenvolvimento de software de modo efetivo e leve. Os modelos ágeis são mais efetivos do que os modelos tradicionais, porque eles são apenas suficientemente bons, eles não precisam ser perfeitos.

Além dos valores que são consistentes com o Manifesto Ágil, Ambler sugere *coragem* e *humildade*. Uma equipe ágil precisa ter a coragem de tomar decisões que podem fazer que um projeto seja rejeitado e refabricado. Ela precisa ter a humildade de reconhecer que os tecnologistas não têm todas as respostas, que os especialistas no negócio e outros interessados devem ser respeitados e acolhidos.

Apesar da AM sugerir uma ampla gama de princípios de modelagem "centrais" e "suplementares", os que tornam a AM peculiar são [AMB02]:

Modelar com uma finalidade. Um desenvolvedor que usa a AM deve ter uma meta específica em mente (por exemplo, comunicar informações ao cliente ou ajudá-lo a entender melhor algum aspecto do software) antes de criar o modelo. Uma vez identificada a meta do modelo, o tipo de notação a ser usada e o nível de detalhe exigido serão mais óbvios.

Usar modelos múltiplos. Há muitos modelos e notações diferentes que podem ser usados para descrever softwares. Apenas um pequeno subconjunto é essencial para a maioria dos projetos. A AM sugere que, para fornecer a visão necessária, cada modelo apresente um aspecto diferente do sistema e que apenas aqueles modelos que ofereçam valor à sua pretensa audiência sejam usados.

Viajar leve. À medida que o trabalho de engenharia de software prossegue, conserve apenas aqueles modelos que fornecerão valor em longo prazo e livre-se do resto. Cada produto de trabalho conservado precisa ser mantido à medida que as modificações ocorrem. Isso representa trabalho que atrasa a equipe. Ambler [AMB02] nota que "cada vez que você decide conservar um modelo, você compromete a agilidade em nome da conveniência de ter aquela informação disponível para a sua equipe de modo abstrato (portanto, potencialmente melhorando a comunicação dentro de sua equipe, bem como com os interessados no projeto)".

AVISO

"Viajar leve" é uma filosofia apropriada para todo trabalho de engenharia de software. Construa apenas aqueles modelos que fornecem valor — nem mais, nem menos.

O conteúdo é mais importante que a representação. A modelagem deve levar informações à sua pretensa audiência. Um modelo sintaticamente perfeito que leva pouco conteúdo útil não é tão valioso como um modelo com notação defeituosa, mas que fornece conteúdo valioso para a sua audiência.

Conhecer os modelos e as ferramentas que você usa para criá-los. Entenda os pontos fortes e fracos de cada modelo e das ferramentas que são usadas para criá-los.

Adaptar localmente. A abordagem de modelagem deve ser adaptada às necessidades da equipe ágil.

FERRAMENTAS DE SOFTWARE

Desenvolvimento Ágil

Objetivo: O objetivo das ferramentas de desenvolvimento ágil é apoiar em um ou mais aspectos do desenvolvimento ágil com ênfase na facilitação da geração rápida de softwares operacionais. Essas ferramentas também podem ser usadas quando modelos prescritivos de processo (Capítulo 3) são aplicados.

Mecânica: A mecânica da ferramenta varia. Em geral, conjuntos de ferramentas ágeis incluem um apoio automatizado de planejamento de projeto, desenvolvimento de casos de uso e levantamento de requisitos, projeto rápido, geração de código e teste.

Ferramentas Representativas:[12]

Nota: Como o desenvolvimento ágil é um assunto quente, a maioria dos vendedores de ferramentas de software alegam vender ferramentas que apóiam a abordagem ágil. As ferramentas mencionadas abaixo têm características que as tornam particularmente úteis para projetos ágeis.

A *Actif Extreme*, desenvolvida pela Microtool (*www.microtool.com*), fornece apoio à gestão ágil de processo para várias atividades técnicas dentro do processo.

A *Ideogramic UML*, desenvolvida pela Ideogramic (*www.ideogramic.com*), é um conjunto de ferramentas UML criado especificamente para uso dentro de um processo ágil.

O *Together Tool Set*, distribuído pela Borland (*www.borland.com* ou *www.togethersoftware.com*), fornece um conjunto de ferramentas que apóia muitas atividades técnicas no XP e em outros processos ágeis.

12 As ferramentas mencionadas aqui não representam uma indicação, mas, em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas de seus respectivos desenvolvedores.

Document shared on www.docsity.com

4.4 Resumo

Uma filosofia ágil para a engenharia de software ressalta quatro tópicos-chave: a importância de equipes auto-organizadas que têm controle sobre o trabalho que executam; comunicação e colaboração entre os membros da equipe e entre os profissionais e seus clientes; um reconhecimento de que modificações representam uma oportunidade; e uma ênfase na entrega rápida de softwares que satisfaçam ao cliente. Os modelos ágeis de processo foram projetados para atender a cada um desses tópicos.

O XP (Extreme Programming) é o processo ágil mais amplamente usado. Organizado como quatro atividades de arcabouço — planejamento, projeto, codificação e teste —, o XP sugere um número de técnicas inovadoras e potentes que permitem a equipes ágeis criar freqüentemente versões de software que possuem características e funcionalidades descritas e priorizadas pelo cliente.

O DAS (Desenvolvimento Adaptativo de Software) ressalta a colaboração humana e a auto-organização da equipe. Organizado como três atividades de arcabouço — especulação, colaboração e aprendizado — o DAS usa um processo iterativo que incorpora planejamento do ciclo adaptativo, métodos relativamente rigorosos para o levantamento de requisitos e um ciclo de desenvolvimento iterativo que incorpora grupos enfocados nos clientes e revisões técnicas formais como mecanismos de feedback em tempo real. O DSDM (Método de Desenvolvimento Dinâmico de Sistemas) define três diferentes ciclos iterativos — iteração do modelo funcional, iteração de projeto e construção e implementação — precedidos por duas atividades de ciclo de vida adicionais — estudo de viabilidade e estudo do negócio. O DSDM recomenda o uso de cronogramação a cada intervalo de tempo e sugere que, em cada incremento de software, é necessário apenas o trabalho suficiente a fim de facilitar o avanço para o incremento seguinte.

O Scrum enfatiza o uso de um conjunto de padrões de processo de software que têm comprovada efetividade para projetos com prazos apertados, requisitos mutáveis e criticalidade de negócio. Cada padrão de processo define um conjunto de tarefas de desenvolvimento e permite à equipe Scrum construir um processo que seja adaptado às necessidades do projeto.

O Crystal é uma família de modelos ágeis de processo que podem ser adotados para as características específicas de um projeto. Como outras abordagens ágeis, o Crystal adota uma estratégia iterativa, mas ajusta o rigor do processo de modo a acomodar projetos de diferentes tamanhos e complexidades.

O FDD (Desenvolvimento Guiado por Características) é algo mais "formal" do que outros métodos ágeis, mas ainda mantém agilidade por concentrar a equipe de projeto no desenvolvimento das características — funções valiosas para o cliente que podem ser implementadas em duas semanas ou menos. O FDD fornece maior ênfase em gestão de projeto e qualidade do que outras abordagens ágeis. A Modelagem Ágil (AM) sugere que a modelagem é essencial para todos os sistemas, mas que a complexidade, tipo e tamanho do modelo devem estar sintonizados com o software a ser construído. Por meio da proposição de um conjunto de princípios de modelagem centrais e suplementares, a AM fornece um guia útil para os profissionais durante as tarefas de análise e do projeto.

Referências Bibliográficas

[ADM96] Advanced Development Methods, Inc., "Origins of Scrum", 1996, http://www.controlchaos.com/.

[AGI03] The Agile Alliance Home Page, http://www.agilealliance.org/home.

[AMB02] Ambler, S., "What Is Agile Modeling (AM)?", 2002, http://www.agilemodeling.com/index.htm.

[BEC99] Beck, K., *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.

[BEC01a] _____, *et al.*, "Manifesto for Agile Software Development", http://www.agilemanifesto.org/.

[BEC01b] _____ e Fowler, M. *Planning Extreme Programming*, Addison-Wesley, 2001.

[BEE99] Beedle, M., *et al.*, "Scrum: An Extension Pattern Language for Hyperproductive Software Development", incluído em *Pattern Languages of Program Design 4*, Addison-Wesley Longman, Reading, MA, 1999. Pode ser baixado em http://jeffsutherland.com/scrum/scrum_plop.pdf.

[BUS00] Buschmann, F., *et al.*, Pattern-Oriented Software Architecture, 2 v., Wiley, 1996, 2000.

[COA99] Coad, P., Lefebvre, E. e DeLuca, J., *Java Modeling in Color with UML*, Prentice-Hall, 1999.

[COC01] Cockburn, A. e Highsmith, "Agile Software Development: The People Factor", *IEEE Computer*, v. 34, n. 11, nov. 2001, p. 131-33.

[COC02a] _____, *Agile Software Development*, Addison-Wesley, 2002.

[COC02b] Cockburn, A., "What Is Agile and What Does It Imply?", apresentado no Agile Development Summit no Westminster College, em Salt Lake City, mar. 2002, http://crystalmethodologies.org/.

[CCS02] CS3 Consulting Services, 2002, http://www.cs3inc.com/DSDM.htm.

[DEM98] DeMarco, T. e Lister, T., *Peopleware*, 2ª ed., Dorset House, 1998.

[DEM02] _____ e Boehm, B., "The Agile Methods Fray", *IEEE Computer*, v. 35, n. 6, jun. 2002, p. 90-92.

[FOW00] Fowler, M., *et al.*, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 2000.

[FOW01] _____ e J. Highsmith, "The Agile Manifesto," *Softwre Development Magazine*, ago. 2001, http://www.sdmagazine.com/documents/s=844/sdm0108a/0108a.htm.

[FOW02] Fowler, M., "The New Methodology", jun. 2002, http://www.martinfowler.com/articles/newMethodology.html#N8B.

[HIG98] Highsmith, J., "Life — The Artificial and the Real", *Software Development*, 1998, http://www.adaptivesd.com/articles/order.html.

[HIG00] _____, Adaptive Software Development: An Evolutionary Approach to Managing Complex Systems, Dorset House Publishing, 1998.

[HIG01] _____, (ed.), "The Great Methodologies Debate: Part1", *Cutter IT Journal*, v. 14, n. 12, dez. 2001.

[HIG02a] _____, (ed.), "The Great Methodologies Debate: Part2", *Cutter IT Journal*, v. 15, n. 1, jan. 2002.

[HIG02b] Highsmith, J., *Agile Software Development Ecosystems*, Addison-Wesley, 2002.

[JAC02] Jacobson, I., "A Resounding 'Yes" to Agile Processes — But Also More", *Cutter IT Journal*, vol .15, n. 1, jan. 2002.

[JEF01] Jeffries, R., *et al.*, *Extreme Programming Installed*, Addison-Wesley, 2001.

[NOY02] Noyes, B. "Rugby, Anyone?", *Managing Development* (uma publicação on-line da Fawcette Technical Publications), jun. 2002, http://www.fawcette.com/resources/managingdev/methodologies/scrum/.

[PAL02] Palmer, S. e Felsing, J., *A Practical Guide to Feature Driven Development*, Prentice-Hall, 2002.

[SCH01] Schwaber, K. e Beedle, M., *Agile Software Development with SCRUM*, Prentice-Hall, 2002.

[SCH02] Schwaber, K., "Agile Processes and Self-Organization", Agile Alliance, 2002, http://www.aanpo.org/articles/index.

[STA97] Stapleton, J., DSDM — Dynamic System Development Method: The Method in Practice, Addison-Wesley, 1997.

[WEL99] Wells, D., "XP — Unit Tests," 1999, http://www.extremeprogramming.org/rules/unittests.html.

Problemas e Pontos a Considerar

4.1. Releia o "Manifesto para o Desenvolvimento Ágil de Software" apresentado no início deste capítulo. Você pode pensar em uma situação em que um ou mais dos quatro "valores" poderiam levar uma equipe de software a ter problemas?

4.2. Descreva a agilidade (para projetos de software) com suas próprias palavras.

4.3. Por que um processo iterativo torna mais fácil gerir modificações? Todo processo ágil discutido neste capítulo é iterativo? É possível completar um projeto em uma única iteração e ainda ser ágil? Explique suas respostas.

4.4. Pode cada um dos processos ágeis ser descrito usando as atividades genéricas de arcabouço vistas no Capítulo 2? Construa uma tabela que mapeie as atividades genéricas para as atividades definidas em cada um dos processos ágeis.

4.5. Tente encontrar um ou mais "princípios de agilidade" que ajudem uma equipe de engenharia de software a tornar-se ainda mais manobrável.

4.6. Selecione um princípio de agilidade apresentado na Seção 4.1 e tente determinar se cada um dos modelos de processos apresentados neste capítulo satisfaz ao princípio.

4.7. Por que os requisitos mudam tanto? Afinal, as pessoas não sabem o que querem?

4.8. Considere as sete características apresentadas na Seção 4.2.2. Ordene essas características em ordem decrescente de importância com base na sua percepção de quais são as mais importantes e quais as menos importantes.

4.9. A maioria dos modelos ágeis de processo recomenda a comunicação face a face. No entanto, hoje em dia, membros de uma equipe de software e seus clientes podem estar geograficamente separados uns dos outros. Você acha que isso implica que a separação geográfica é algo a se evitar? Você pode pensar em modos de solucionar esse problema?

Document shared on www.docsity.com

4.10. Escreva uma história de usuário XP que descreva os "locais favoritos" ou a característica "favoritos" que está disponível na maioria dos navegadores Web.

4.11. O que é uma solução de ponta em XP?

4.12. Descreva os conceitos de *refabricação* e *programação em pares* em XP, com suas próprias palavras.

4.13. Usando o gabarito de padrão de processo apresentado no Capítulo 2, desenvolva um padrão de processo para um dos padrões Scrum apresentados na Seção 4.3.4.

4.14. Por que o Crystal é chamado de *família de métodos ágeis*?

4.15. Usando o gabarito de características FDD descrito na Seção 4.3.6, defina um conjunto de características para um navegador Web. Depois desenvolva um conjunto de características para o conjunto de características.

4.16. Visite a página da Web Official Agile Modelling Site e faça uma lista completa de todos os princípios da AM centrais e suplementares.

Leituras e Fontes de Informação Adicionais

A filosofia global e os princípios subjacentes do desenvolvimento ágil de software são considerados em profundidade nos livros de Ambler (*Agile Modeling*, Wiley, 2002), Beck [BEC99], Cockburn [COC02] e Highsmith [HIG02].

Os livros de Beck [BEC99], Jeffries e seus colegas (*Extreme Programming Installed*, Addison-Wesley, 2000), Succi e Marchesi (*Extreme Programming Examined*, Addison-Wesley, 2001), Newrik e Martin (*Extreme Programming in Practice*, Addison-Wesley, 2001) e Auer e seus colegas (*Extreme Programming Applied: Play to Win*, Addison-Wesley, 2001) fornecem uma discussão prática do XP juntamente com diretrizes sobre como melhor aplicá-lo. McBreen (*Questioning Extreme Programming*, Addison-Wesley, 2003) dá uma olhada crítica no XP, definindo quando e onde ele é apropriado. Uma consideração mais aprofundada de programação em pares é apresentada por McBreen (*Pair Programming Illuminated*, Addison-Wesley, 2003).

Fowler e seus colegas (*Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999) abordam, em detalhes consideráveis, o importante conceito de refabricação em XP. McBreen (*Software Craftsmanship: The New Imperative*, Addison-Wesley, 2001) discute a habilidade para o trabalho de software e argumenta em favor das alternativas ágeis em oposição à engenharia de software tradicional.

O DAS é apresentado em profundidade por Highsmith [HIG00]. Um tratamento vantajoso do DSDM foi escrito por Stapleton (*DSDM: The Method in Practice*, Addison-Wesley, 1997). Palmer e Felsing [PAL02] apresentam um tratamento detalhado do FDD. Carmichael e Haywood (*Better Software Faster*, Prentice-Hall, 2002) apresentam um outro tratamento útil de FDD que inclui uma viagem passo-a-passo pela mecânica do processo. Schwaber e seus colegas (*Agile Software Development with SCRUM*, Prentice-Hall, 2001) apresentam um tratamento detalhado do Scrum.

Martin (*Agile Software Development*, Prentice-Hall, 2003) discute os princípios ágeis, padrões e práticas com ênfase no XP. Poppendieck e Poppendieck (*Lean Development: An Agile Toolkit for Software Development Managers*, Addison-Wesley, 2003) fornecem diretrizes para gestão e controle de projetos ágeis. Highsmith (*Agile Software Development Ecosystems*, Addison-Wesley, 2002) apresenta um levantamento valioso dos princípios, processos e práticas ágeis.

Uma ampla variedade de fontes de informação sobre o desenvolvimento ágil de softwares está disponível na Internet. Uma lista atual de referências da World Wide Web que são relevantes para o processo ágil pode ser encontrada no site destte livro: **http://www.mhhe.com/pressman.**

Document shared on www.docsity.com

PARTE

DOIS

PRÁTICA DE ENGENHARIA DE SOFTWARE

Nesta parte de *Engenharia de Software* você aprenderá sobre os princípios, conceitos e métodos que abrangem a prática de engenharia de software. Essas questões são tratadas nos capítulos seguintes:

- Que conceitos e princípios guiam a prática de engenharia de software?
- Como a engenharia de sistemas leva à efetiva engenharia de software?
- O que é engenharia de requisitos e quais são os conceitos subjacentes que conduzem a uma boa análise de requisitos?
- Como o modelo de análise é criado e quais são os seus elementos?
- O que é engenharia de projeto e quais são os conceitos subjacentes que conduzem a um bom projeto?
- Quais são os conceitos, modelos e métodos usados para criar os projetos arquitetural, de interface e de componente?
- Quais são as estratégias aplicáveis ao teste de software?
- Que métodos são usados para projetar casos de teste efetivos?
- Quais medidas e métricas podem ser usadas para avaliar a qualidade dos modelos de análise e projeto, código-fonte e casos de teste?

Uma vez respondidas essas questões, você ficará mais bem preparado para aplicar a prática de engenharia de software.

CAPÍTULO 5

PRÁTICA: UMA VISÃO GENÉRICA

CONCEITOS-CHAVE

princípios de:
análise 87
codificação 91
comunicação..... 82
engenharia de software 79
implantação 94
modelagem ágil..... 90
planejamento..... 84
projeto..... 89
teste..... 92
questões W^3HH 86
solução de problemas 79

Em um livro que explora as vidas e as idéias de engenheiros de software, Ellen Ullman [ULL97] mostra um relance da vida de um deles quando relata os pensamentos de um profissional sob pressão:

> Não tenho idéia de que horas são. Não há janelas nesse escritório nem relógio, apenas o pisca-pisca vermelho do display LED de um microondas que pisca 12:00, 12:00, 12:00, 12:00. Joel e eu temos programado durante dias. Temos um erro, um teimoso diabo de um erro. Assim, o pulsar vermelho sem tempo parece correto, como uma leitura de nossos cérebros que se sincronizaram de algum modo na mesma velocidade da piscada...
>
> Em que estamos trabalhando?... Os detalhes me escapam nesse momento. Nós podemos estar ajudando pessoas pobres, doentes ou ajustando um conjunto de rotinas de baixo nível para verificar bits em um protocolo de banco de dados distribuído — não importa. Eu deveria me importar; em toda parte do meu ser — mais tarde, talvez quando sairmos dessa sala cheia de computadores — vou me importar muito com por que, para quem e com que finalidade eu estou escrevendo softwares. Mas, agora não. Passei através de uma membrana em que o mundo real e seus usos não mais importam. Sou um engenheiro de software...

Uma imagem negra da prática de engenharia de software sem dúvida, mas após alguma reflexão, muitos dos leitores desse livro poderão se identificar com ela.

As pessoas que criam softwares de computador praticam a arte, ofício ou disciplina,[1] que é a engenharia de software. Mas o que é "prática" de engenharia de software? De modo genérico, a *prática* é uma coleção de conceitos, princípios, métodos e ferramentas da qual um engenheiro de software faz uso diariamente. A prática permite aos gerentes gerenciar projetos de software e, aos engenheiros de software, construir programas de computador. A prática preenche um modelo de processo de software com as receitas técnicas e gerenciais necessárias para fazer o serviço. Ela transforma uma abordagem aleatória, não enfocada, em alguma coisa que é mais organizada, efetiva e provável de alcançar sucesso.

PANORAMA

O que é? A prática é um amplo conjunto de conceitos, princípios, métodos e ferramentas que você pode considerar à medida que o software é planejado e desenvolvido. Ela representa os detalhes — considerações técnicas e como fazer — que estão sob a superfície do processo de software: coisas de que você vai precisar para realmente construir softwares de computador de alta qualidade.

Quem faz? A prática da engenharia de software é aplicada por engenheiros de software e por seus gerentes.

Por que é importante? O processo de software fornece a todos os envolvidos na criação de um sistema ou produto baseado em computador um roteiro para chegar a um destino de modo bem-sucedido. A prática fornece os detalhes de que você vai precisar para seguir caminho. Ela lhe diz onde estão as pontes, os bloqueios e as encruzilhadas. Ela o ajuda a entender os conceitos e princípios que precisam ser compreendidos e seguidos para dirigir com segurança e rapidez. Ela o instrui sobre como dirigir, onde diminuir a velocidade e onde aumentá-la. No contexto de engenharia de software, a prática é o que você faz diariamente à medida que o software evolui de uma idéia para uma realidade.

Quais são os passos? Três elementos da prática aplicam-se independentemente do modelo de processo escolhido. São eles: conceitos, princípios e métodos. Um quarto elemento da prática — as ferramentas — dá apoio à aplicação dos métodos.

1 Alguns autores preferem um desses termos em detrimento dos outros. Na realidade, a engenharia de software é o conjunto de todos os três.

Qual é o produto do trabalho? A prática engloba as atividades técnicas que produzem todos os produtos de trabalho definidos pelo modelo de processo de software escolhido.

Como tenho certeza de que fiz corretamente? Primeiro, tenha um entendimento sólido dos conceitos e princípios que se aplicam ao trabalho (por exemplo, projeto) que você está fazendo no momento. Depois, certifique-se de que você escolheu um método de trabalho adequado; certifique-se de que entendeu como aplicar o método e de usar as ferramentas automáticas quando elas são adequadas para a tarefa e seja exigente quanto à necessidade de técnicas para garantir a qualidade dos produtos de trabalho que são produzidos.

5.1 PRÁTICA DE ENGENHARIA DE SOFTWARE

Veja na Web

Diversas citações estimulantes sobre a prática de engenharia de software podem ser encontradas em www.literateprogramming.com.

No Capítulo 2 apresentamos um modelo genérico de processo de software composto de um conjunto de atividades que estabelecem um arcabouço para a prática de engenharia de software. As atividades genéricas de arcabouço — comunicação, planejamento, modelagem, construção e implantação — e as atividades guarda-chuva estabelecem um esqueleto de arquitetura para o trabalho de engenharia de software. Todos os modelos de processo de software apresentados nos Capítulos 3 e 4 podem ser mapeados nesse esqueleto de arquitetura. Entretanto, como a prática de *engenharia de software* se encaixa? Nas seções seguintes, consideramos os conceitos e princípios genéricos que se aplicam às atividades de arcabouço.[2]

AVISO

Você pode alegar que a abordagem de Polya é simplesmente bom senso. É verdade. Mas é incrível quão freqüentemente o bom senso é incomum no mundo de software.

5.1.1 A Essência da Prática

Em um livro clássico, *How to Solve It*, escrito antes que os computadores modernos existissem, George Polya [POL45] esboçou a essência da solução de problemas e, conseqüentemente, a essência da prática de engenharia de software:

1. *Entenda o problema* (comunicação e análise).
2. *Planeje uma solução* (modelagem e projeto de software).
3. *Execute o plano* (geração de código).
4. *Examine o resultado quanto à precisão* (teste e garantia de qualidade).

No contexto da engenharia de software, esses passos de bom senso levam a uma série de questões essenciais [adaptado de POL45]:

Entenda o problema.

- *Quem tem interesse na solução do problema?* Isto é, quem são os interessados?
- *Quais são as incógnitas?* Que dados, funções, características e comportamento são necessários para resolver adequadamente o problema?
- *O problema pode ser compartimentalizado?* É possível representar problemas menores que podem ser mais fáceis de entender?
- *O problema pode ser representado graficamente?* Um modelo de análise pode ser criado?

Planeje a solução.

- *Você já viu problemas análogos?* Existem padrões que são reconhecíveis em uma solução potencial? Há algum software que implemente os dados, funções, características e comportamento requeridos?
- *Um problema semelhante foi resolvido?* Em caso afirmativo, há elementos da solução reutilizáveis?

2 O leitor é incentivado a revisitar seções relevantes deste capítulo à medida que métodos e atividades específicos de engenharia de software forem discutidos mais adiante neste livro.

Document shared on www.docsity.com

- *Podem ser definidos subproblemas?* Em caso afirmativo, há soluções prontamente aparentes para os subproblemas?
- *Você pode representar uma solução de modo que leve à implementação efetiva?* Um modelo de projeto pode ser criado?

Execute o plano.

- *A solução está de acordo com o plano?* O código-fonte pode ser rastreado até o modelo de projeto?
- *Cada componente parte da solução está provavelmente correto?* O projeto e o código foram revisados, ou melhor, foram aplicadas provas de correção ao algoritmo?

Examine o resultado.

- *É possível testar cada componente que é parte da solução?* Foi implementada uma estratégia de teste razoável?
- *A solução produz resultados que estão de acordo com os dados, funções, características e comportamento que são necessários?* O software foi validado com base em todos os requisitos dos interessados?

> "Existe um grão de descoberta na solução de qualquer problema."
>
> **George Polya**

5.1.2 Princípios Centrais

O dicionário define a palavra *princípio* como "uma importante lei ou pressuposto subjacente exigido em um sistema de raciocínio". Ao longo deste livro, discutimos princípios em muitos níveis de abstração diferentes. Alguns enfocam a engenharia de software como um todo, outros consideram uma atividade genérica de arcabouço específica (por exemplo, comunicação com o cliente) e outros, ainda, enfocam as ações de engenharia de software (projeto arquitetural) ou tarefas técnicas (por exemplo, escrever um cenário de uso). Independentemente de seu nível de enfoque, os princípios nos ajudam a estabelecer uma mente voltada para a sólida prática de engenharia de software. Por essa razão é que eles são importantes.

AVISO

Antes de começar um projeto de software, certifique-se de que o software tem uma finalidade de negócio e de que os usuários percebem valor nisso.

David Hooker [Hoo96] propôs sete princípios centrais da prática e da engenharia de software como um todo. Eles são reproduzidos a seguir:[3]

O Primeiro Princípio: *A Razão Por que Tudo Existe*

Um sistema de software existe por uma razão: para *fornecer valor aos seus usuários*. Todas as decisões devem ser tomadas com isso em mente. Antes de especificar um requisito de sistema, antes de anotar uma parte da funcionalidade do sistema, antes de determinar as plataformas de hardware ou processos de desenvolvimento, formule questões para si próprio tais como: Isso adiciona valor real ao sistema? Se a resposta for não, não o faça. Todos os outros princípios apóiam esse.

O Segundo Princípio: *KISS (Keep It Simple, Stupid! — Mantenha a Coisa Simples!)*

Projeto de software não é um processo aleatório. Existem muitos fatores a considerar em qualquer esforço de projeto. *Todo projeto deve ser tão simples quanto possível, mas não mais simples.* Isso facilita ter um sistema mais fácil de entender e de manter, mas não quer dizer que características, mesmo características internas, devam ser descartadas em nome da simplicidade. Na verdade, os projetos mais elegantes costumam ser os mais simples. Simples também não significa "rápido e

3 Reproduzido com permissão do autor [HOO96]. Hooker define padrões para esses princípios em: *http://c2.com/cgi/wiki?SevenPrinciplesOfSoftwareDevelopment*.

sujo". De fato, freqüentemente, simplificar precisa de muito raciocínio e trabalho em várias iterações. A recompensa é um software que é mais manutenível e menos propenso a erro.

> "Existe uma certa majestade na simplicidade que está muito acima da clareza de raciocínio."
>
> **Alexander Pope (1688-1744)**

PONTO CHAVE

Se o software tiver valor, ele será modificado durante a sua vida útil. Por essa razão, o software precisa ser construído para ser manutenível.

O Terceiro Princípio: *Mantenha a Visão*

Uma visão clara é essencial para o sucesso de um projeto de software. Sem ela, um projeto quase sem falha acaba sendo "de duas [ou mais] mentes". Sem integridade conceitual, um sistema fica ameaçado de tornar-se uma colcha de retalhos de projetos incompatíveis, unidos pelo tipo errado de parafuso.

O comprometimento da visão arquitetural de um sistema de software enfraquece e pode eventualmente destruir um sistema bem projetado. Ter um arquiteto fortalecido que possa manter a visão e exigir que ela seja respeitada ajuda a garantir um projeto de software muito bem-sucedido

O Quarto Princípio: *O Que Você Produz Outros Vão Consumir*

Raramente um sistema de software de qualidade industrial é construído e usado em um vácuo. De um modo ou de outro alguém mais vai usar, manter, documentar ou precisará entender o seu sistema. Assim, *sempre especifique, projete e implemente sabendo que mais alguém terá de entender o que você está fazendo.* A audiência de qualquer produto do desenvolvimento de software é potencialmente grande. Especifique com um olho no usuário. Projete tendo em mente os implementadores. Codifique preocupando-se com aqueles que precisam manter e estender o sistema. Alguém pode ter de depurar o código que você escreve e isso o torna usuário do seu código. Tornar o trabalho deles mais fácil adiciona valor ao sistema.

O Quinto Princípio: *Esteja Aberto para o Futuro*

Um sistema com um longo tempo de vida tem mais valor. Nos ambientes de computação atuais, em que as especificações mudam de um momento para outro e as plataformas de hardware ficam obsoletas depois de apenas alguns meses, os tempos de vida do software são tipicamente medidos em meses em vez de anos. No entanto, os sistemas de software com verdadeira "força industrial" precisam durar muito mais. Para fazer isso com sucesso, eles precisam estar prontos para se adaptar a essas e outras modificações. Sistemas que fazem isso com sucesso são aqueles que foram projetados dessa forma desde o início. *Nunca projete a si mesmo em um beco sem saída.* Sempre pergunte "e se", e prepare-se para todas as possíveis respostas, criando sistemas que resolvem o problema geral, não apenas o específico[4]. Isso pode, muito provavelmente, levar ao reuso de um sistema inteiro.

O Sexto Princípio: *Planeje com Antecedência o Reuso*

Reuso poupa tempo e esforço[5]. Conseguir um alto nível de reuso é alegadamente a meta mais difícil de alcançar no desenvolvimento de um sistema de software. O reuso de código e de projetos tem sido proclamado como um importante benefício do uso de tecnologias orientadas a objetos. No entanto, o retorno desse investimento não é automático. Para alavancar as possibilidades de reuso oferecidas pela programação orientada a objetos [ou convencional], previsão e planejamento são exigidos. Existem muitas técnicas para realizar o reuso em todos os níveis do processo de desenvolvimento de sistemas. As relativas aos níveis de projeto detalhado e codificação são bem conhecidas e documentadas. Literatura recente está tratando do reuso de projetos na forma de padrões de software. No entanto, isso é apenas parte da batalha. A comunicação das oportunidades de reuso para outros na organização é vital. Como você pode reutilizar algo que não sabe que

4 Nota do Autor: Esse conselho pode ser perigoso se for levado ao extremo. Projetar para o "problema geral" exige, algumas vezes, comprometimento do desempenho e pode exigir mais esforço de projeto.

5 Nota do Autor: Apesar disso ser verdade para aqueles que reusam o software em projetos futuros, o reuso pode ser caro para aqueles que precisam projetar e construir componentes reusáveis. Estudos indicam que projetar e construir componentes reusáveis pode custar de 25% a 200% a mais do que o software-alvo. Em alguns casos, o diferencial de custo pode não ser justificável.

Document shared on www.docsity.com

existe? *Planejar o reuso com antecedência reduz o custo e aumenta o valor tanto dos componentes reusáveis quanto do sistema ao qual eles são incorporados.*

O Sétimo Princípio: *Pense!*

Este último princípio é provavelmente o mais esquecido. *Raciocinar clara e completamente antes da ação quase sempre produz melhores resultados.* Quando você pensa sobre algo, é mais provável que o faça corretamente. Você também adquire conhecimento sobre como fazê-lo certo novamente. Se pensa sobre algo e, ainda assim, o faz errado, obtém experiência valiosa. Um efeito colateral do raciocínio é aprender a reconhecer quando você não sabe alguma coisa, ponto em que você pode pesquisar a resposta. Quando o raciocínio límpido é colocado no sistema, o valor aparece. Aplicar os seis primeiros Princípios exige raciocínio intenso, para o qual a recompensa potencial é enorme.

Se todo engenheiro de software e toda equipe de software simplesmente seguissem os sete princípios de Hooker, muitas das dificuldades pelas quais passamos ao construir sistemas complexos baseados em computador seriam eliminadas.

5.2 Práticas de Comunicação

Antes que os requisitos do cliente possam ser analisados, modelados ou especificados, eles precisam ser coletados por meio de uma atividade de *comunicação* (também chamada de *levantamento de requisitos*). O cliente tem um problema que pode ser adequado a uma solução baseada em computador. O desenvolvedor responde à solicitação de ajuda do cliente. A comunicação começou. Mas o caminho da comunicação para o entendimento é freqüentemente cheio de buracos.

A comunicação efetiva (entre pares técnicos, com o cliente e outros interessados e com os gerentes de projeto) está entre as atividades mais desafiadoras com as quais se confronta um engenheiro de software. Nesse contexto, discutimos como os princípios e conceitos de comunicação se aplicam à comunicação com o cliente. No entanto, muitos dos princípios aplicam-se igualmente a todas as formas de comunicação que ocorrem dentro de um projeto de software.

AVISO

Antes da comunicação certifique-se de que entende o ponto de vista da outra parte, conhece um pouco de suas necessidades e, então, escute.

Princípio nº 1: *Escute.* Tente se concentrar nas palavras do interlocutor em vez de na formulação de sua resposta a essas palavras. Peça esclarecimentos se algo estiver obscuro, mas evite interrupções constantes. Nunca torne suas próprias palavras ou ações desagradáveis (por exemplo, virar os olhos ou sacudir a cabeça) quando uma pessoa estiver falando.

Princípio nº 2: *Prepare-se antes de se comunicar.* Gaste tempo em entender o problema antes de se encontrar com os outros. Se necessário, pesquise para entender o jargão dominante do negócio. Se você é responsável pela condução de uma reunião, prepare uma agenda com antecedência.

Princípio nº 3: *Alguém deve facilitar a atividade.* Toda reunião de comunicação deve ter um líder (facilitador) (1) para manter a conversa se movendo em uma direção produtiva; (2) para mediar qualquer conflito que ocorra; e (3) para garantir que os outros princípios sejam seguidos.

Princípio nº 4: *Comunicação face-a-face é melhor.* Mas costuma funcionar melhor quando alguma outra representação da comunicação relevante é apresentada. Por exemplo, um participante pode criar um desenho ou um documento "*provisório*" que serve como foco da discussão.

> "Questões simples e respostas simples formam o caminho mais curto para a maioria das perplexidades."
>
> **Mark Twain**

Princípio nº 5: *Faça anotações e documente as decisões.* As coisas têm um modo de escorrer por entre os dedos. Alguém que participe da comunicação deve servir como "registrador" e anotar todos os pontos e decisões importantes.

Princípio nº 6: *Busque colaboração.* Colaboração e consenso ocorrem quando o conhecimento coletivo dos membros da equipe é combinado para descrever funções ou características do produto ou do sistema. Cada pequena colaboração serve para construir confiança entre os membros da equipe e cria um objetivo comum para a equipe.

Início

A Diferença entre Clientes e Usuários Finais

Os engenheiros de software comunicam-se com muitos interessados diferentes, mas são os clientes e usuários finais que têm o impacto mais significativo no trabalho técnico que se segue. Em alguns casos, o cliente e o usuário final são a mesma pessoa, mas em muitos projetos o cliente e o usuário final são pessoas diferentes, trabalhando para diferentes gerentes em diversas organizações de negócio diferentes.

Um *cliente* é a pessoa ou grupo que: (1) originalmente solicitou o software a ser construído; (2) define os objetivos globais do negócio para o software; (3) fornece os requisitos básicos do produto; e (4) coordena a obtenção de recursos para o projeto. Em um negócio de produtos ou sistemas, o cliente é freqüentemente o departamento de marketing. Em um ambiente de TI, o cliente pode ser um componente ou departamento do negócio.

O *usuário final* é a pessoa ou grupo que: (1) efetivamente usará o software que está sendo construído para satisfazer algum objetivo do negócio, e (2) definirá os detalhes operacionais do software de modo que o objetivo do negócio possa ser alcançado.

? O que acontece se eu não conseguir chegar a um acordo com o cliente sobre algum tópico relacionado ao projeto?

Princípio nº 7: *Conserve-se enfocado, modularize sua discussão.* Quanto mais pessoas estiverem envolvidas em uma comunicação, mais provavelmente aquela discussão vai ficar saltando de um tópico para outro. O facilitador deve manter a conversa modular, abandonando um tópico apenas depois que tiver sido resolvido (no entanto, veja o Princípio nº 9).

Princípio nº 8: *Se algo não está claro, desenhe uma figura.* A comunicação verbal vai só até certo ponto. Um esboço ou um desenho pode, freqüentemente, fornecer esclarecimento quando as palavras não conseguem fazer o serviço.

Princípio nº 9: *(a) Quando você concorda com algo, prossiga; (b) Se você não pode concordar com algo, prossiga; (c) Se uma característica ou função não está clara e não pode ser esclarecida no momento, prossiga.* A comunicação, como qualquer atividade de engenharia de software, leva tempo. Em vez de iterar sem fim, as pessoas que participam devem reconhecer que

CASASEGURA

Erros de Comunicação

A cena: Local de trabalho da equipe de engenharia de software.

Os personagens: Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software; Ed Robbins, membro da equipe de software.

A conversa:

Ed: O que você sabe sobre esse projeto *CasaSegura*?

Vinod: A reunião inicial está programada para a próxima semana.

Jamie: Eu já fiz um pouco de investigação, mas não fui muito bem.

Ed: O que você quer dizer com isso?

Jamie: Bem, eu dei um telefonema para Lisa Perez. Ela é a pessoa de marketing nessa coisa.

Vinod: E...?

Jamie: Eu queria que ela me contasse quais as funções e características do *CasaSegura*... esse tipo de coisa. Em vez disso, ela começou a me fazer perguntas sobre sistemas de segurança, sistemas de vigilância... eu não sou especialista.

Vinod: E o que isso lhe diz?

(Jamie sacode os ombros.)

Vinod: Que o marketing vai precisar da gente agindo como consultores, e que é melhor nós fazermos a lição de casa sobre essa área de produto antes da reunião de partida. Doug disse que queria que "colaborássemos" com nosso cliente, assim, é melhor que a gente aprenda a fazer isso.

Ed: Provavelmente teria sido melhor se você tivesse ido até a sala dela. Telefonemas não funcionam tão bem para esse tipo de coisa.

Jamie: Vocês dois estão certos. Nós precisamos juntar a nossa ação ou as nossas comunicações iniciais vão ser uma luta.

Vinod: Eu vi Doug lendo um livro sobre "engenharia de requisitos". Aposto que lista alguns princípios de boa comunicação. Vou pedir emprestado o livro para ele.

Jamie: Boa idéia; depois você pode nos ensinar.

Vinod (sorrindo): É, está certo.

Document shared on www.docsity.com

muitos tópicos requerem discussão (veja o Princípio nº 2) e que "prosseguir" é, às vezes, o melhor modo de conseguir agilidade de comunicação.

Princípio, nº 10: ***Negociação não é um concurso ou um jogo. Funciona melhor quando ambas as partes ganham.*** Existem muitas instâncias nas quais o engenheiro de software e o cliente precisam negociar funções e características, prioridades e datas de entrega. Se a equipe tem colaborado bem, todas as partes têm uma meta em comum. Assim sendo, a negociação exige compromisso de todas as partes.

CONJUNTO DE TAREFAS

Conjunto Genérico de Tarefas para Comunicação

1. Identificar o cliente principal e outros interessados (Seção 7.3.1).
2. Reunir-se com os clientes principais para tratar de "questões livres de contexto" (Seção 7.3.4) que definem:
   - Necessidade do negócio e valores do negócio.
   - Características/necessidades dos usuários finais.
   - Saídas visíveis ao usuário requeridas.
   - Restrições do negócio.
3. Desenvolver uma declaração escrita de uma página sobre o escopo do projeto, sujeita à revisão (Seções 7.4.1 e 21.3.1).
4. Revisar a declaração de escopo com os interessados e corrigir conforme necessário.
5. Colaborar com os clientes/usuários finais para definir:
   - Cenários de uso visíveis ao cliente usando o formato padrão[6] (Seção 7.5).
   - Saídas e entradas resultantes.
   - Características, funções e comportamento importantes do software.
   - Riscos do negócio definidos pelo cliente (Seção 25.3).
6. Desenvolver uma breve descrição por escrito (por exemplo, um conjunto de listas) de cenários, saídas/entradas, características/funções e riscos.
7. Iterar com o cliente para refinar cenários, saídas/entradas, características/funções e riscos.
8. Atribuir prioridades definidas pelo cliente a cada cenário de usuário, característica, função e comportamento (Seção 7.4.2).
9. Revisar todas as informações coletadas durante a atividade de comunicação com o cliente e outros interessados, e corrigir quando necessário.
10. Preparar-se para a atividade de planejamento (Capítulos 23 e 24).

5.3 PRÁTICAS DE PLANEJAMENTO

A atividade de comunicação ajuda uma equipe de software a definir seus objetivos e metas gerais (sujeitos, certamente, a modificações com o passar do tempo). No entanto, entender essas metas e objetivos não é o mesmo que definir um plano para atingi-los. A atividade de *planejamento* inclui um conjunto de práticas gerenciais e técnicas que permite à equipe de software definir um roteiro enquanto ela se move em direção a sua meta estratégica e seus objetivos táticos.

> "Na preparação para uma batalha, sempre achei que planos são inúteis, mas planejamento é indispensável."
>
> **Dwight D. Eisenhower**

Existem muitas filosofias de planejamento diferentes. Algumas pessoas são "minimalistas", alegando que modificações freqüentemente obscurecem a necessidade de um plano detalhado. Outras são "tradicionalistas", alegando que o plano fornece um roteiro efetivo e quanto mais detalhes tiver, menos provavelmente a equipe vai ficar perdida. Outros ainda são "agilistas", alegando que um rápido "jogo de planejamento" pode ser necessário, porém o roteiro vai emergir à medida que "trabalho real" no software tiver início.

6 Os formatos para cenários de uso serão discutidos no Capítulo 8.

O que fazer? Em muitos projetos, o planejamento excessivo consome tempo e não produz frutos (muitas coisas são modificadas), mas falta de planejamento é uma receita para o caos. Como quase tudo na vida, o planejamento deve ser conduzido com moderação, suficiente para fornecer direção útil para a equipe — nem mais, nem menos.

Independentemente do rigor com o qual o planejamento é conduzido, os princípios a seguir sempre se aplicam:

Veja na Web

Um excelente repositório de informações sobre planejamento e gestão de projeto pode ser encontrado em www.4pm.com/repository.htm.

Princípio nº 1: ***Entenda o escopo do projeto.*** É impossível usar um roteiro se você não souber para onde está indo. O escopo fornece à equipe de software um destino.

Princípio nº 2: ***Envolva o cliente na atividade de planejamento.*** O cliente define prioridades e oferece restrições de projeto. Para acomodar essas realidades, os engenheiros de software precisam freqüentemente negociar ordem de entrega, prazos e outros tópicos relacionados ao projeto.

Princípio nº 3: ***Reconheça que o planejamento é iterativo.*** Um plano de projeto nunca é gravado em pedra. Quando o trabalho tem início, é muito provável que as coisas se modifiquem. Como conseqüência, o plano deve ser ajustado para acomodar essas modificações. Além disso, modelos de processo iterativos e incrementais determinam o replanejamento (após a entrega de cada incremento de software) baseado no feedback recebido dos usuários.

Princípio nº 4: ***Estime com base no que é sabido.*** A intenção de uma estimativa é fornecer uma indicação do esforço, do custo e da duração de tarefas com base no entendimento atual da equipe quanto ao trabalho a ser feito. Se a informação for vaga ou não confiável, as estimativas serão igualmente não confiáveis.

Princípio nº 5: ***Considere riscos à medida que você define o plano.*** Se a equipe tiver definido riscos que têm grande impacto e alta probabilidade, é necessário planejamento de contingência. Além disso, o plano do projeto (inclusive o cronograma) deve ser ajustado para acomodar a probabilidade de um ou mais desses riscos vir a ocorrer.

Princípio nº 6: ***Seja realista.*** As pessoas não trabalham 100% de cada dia. Sempre há ruído em qualquer comunicação humana. Omissões e ambigüidade são fatos da vida. Modificações ocorrerão. Mesmo os melhores engenheiros de software cometem erros. Essas e outras realidades devem ser consideradas quando um plano de projeto for estabelecido.

> "Sucesso é mais função de bom senso consistente do que de gênios."
>
> **An Wang**

PONTO CHAVE

O termo *granularidade* refere-se ao nível de detalhe com o qual algum elemento de planejamento é representado ou conduzido.

Princípio nº 7: ***Ajuste a granularidade à medida que você define o plano.*** *Granularidade* refere-se ao nível de detalhe que é introduzido à medida que um plano de projeto é desenvolvido. Um plano de "granularidade fina" fornece detalhes significativos das tarefas de trabalho que são planejadas por incrementos de tempo relativamente curtos (de modo que o acompanhamento e o controle ocorram freqüentemente). Um plano de "granularidade grossa" fornece tarefas de trabalho mais amplas, planejadas por períodos de tempo mais longos. Em geral, a granularidade move-se de fina para grossa à medida que a linha de tempo do projeto se afasta da data atual. Para as próximas poucas semanas ou meses, o projeto pode ser planejado em significativo detalhe. Atividades que não vão ocorrer nos próximos meses não exigem granularidade fina (muita coisa pode mudar).

Princípio nº 8: ***Defina como você pretende garantir a qualidade.*** O plano deve identificar como a equipe de software pretende garantir a qualidade. Se revisões técnicas formais[7] tiverem de ser conduzidas, elas devem estar nos cronogramas. Se a programação aos pares (Capítulo 4) tiver de ser usada durante a construção, isso deve ser explicitamente definido dentro do plano.

Princípio nº 9: ***Descreva como você pretende acomodar as modificações.*** Mesmo o melhor planejamento pode ser comprometido por modificações descontroladas. A equipe de software deve identificar como as modificações devem ser acomodadas à medida que o trabalho de engenharia de software prossegue. Por exemplo, o cliente pode solicitar uma modificação a qualquer momento?

7 As revisões técnicas formais serão discutidas no Capítulo 26.

Document shared on www.docsity.com

Se uma modificação for solicitada, a equipe é obrigada a implementá-la imediatamente? Como é avaliado o impacto e o custo da modificação?

Princípio nº 10: ***Acompanhe o plano com freqüência e faça ajustes quando necessário.*** Projetos de software se atrasam um dia de cada vez. Assim, faz sentido acompanhar o progresso diariamente, procurando áreas de problema e situações em que o trabalho programado não está de acordo com o trabalho real conduzido. Quando um desvio é encontrado, o plano é ajustado de acordo.

Para ser mais efetivo, todos da equipe de software devem participar da atividade de planejamento. Só depois, os membros da equipe "assinam" o plano.

Em um excelente trabalho sobre processos e projetos de software, Barry Boehm [BOE96] afirma: "Você precisa de um princípio de organização que admita diminuição de escala para fornecer planos [de projeto] simples para projetos simples". Boehm sugere uma abordagem com foco nos objetivos do projeto, marcos e cronogramas, responsabilidades, abordagens gerenciais e técnicas, e recursos necessários. Ele a denomina o princípio W^{5}HH, por causa de uma série de questões que levam à definição das características-chave do projeto e do plano de projeto resultante.

? Quais questões devem ser formuladas e respondidas a fim de desenvolver um plano de projeto realístico?

Por que (Why) o sistema está sendo desenvolvido? Todas as partes deveriam examinar a validade das razões comerciais para o trabalho de software. Dito de outra forma, a razão comercial justifica o gasto de pessoal, tempo e dinheiro?

O que (What) será feito? Identifique a funcionalidade a ser construída e, por implicação, as tarefas necessárias para que o serviço seja feito.

Quando (When) será concluído? Estabeleça um fluxo de trabalho e prazo para as tarefas-chave do projeto e identifique os marcos solicitados pelos clientes.

Quem (Who) é responsável por uma função? O papel e a responsabilidade de cada membro da equipe de software devem ser definidos.

Onde (Where) estão localizados na organização? Nem todos os papéis e responsabilidades situam-se na própria equipe de desenvolvimento de software. O cliente, usuários e outros interessados também têm responsabilidades.

Como (How) o trabalho será conduzido técnica e gerencialmente? Uma vez estabelecido o escopo do produto, uma estratégia gerencial e técnica para o projeto deve ser definida.

CONJUNTO DE TAREFAS

Conjunto Genérico de Tarefas para Planejamento

1. Reavaliar o escopo do projeto. (Seções 7.4 e 21.3).
2. Avaliar os riscos (Seção 25.4).
3. Desenvolver e/ou refinar cenários de usuário (Seções 7.5 e 8.5).
4. Extrair funções e características dos cenários (Seção 8.5).
5. Definir funções e características técnicas que formem a infra-estrutura do software.
6. Agrupar funções e características (cenários) por prioridade do cliente.
7. Criar um plano de projeto de granularidade grossa (Capítulos 23 e 24).
   - Definir o número de incrementos de software projetados.
   - Estabelecer um cronograma geral do projeto (Capítulo 24).
   - Estabelecer as datas de entregas projetadas para cada incremento.
8. Criar um plano de granularidade fina para a iteração atual (Capítulos 23 e 24).
   - Definir tarefas de trabalho para cada característica de função (Seção 23.6).
   - Estimar o esforço para cada tarefa de trabalho (Seção 23.6).
   - Atribuir responsabilidade a cada tarefa de trabalho (Seção 23.4).
   - Definir os produtos de trabalho a serem produzidos.
   - Identificar os métodos de garantia de qualidade a ser usados (Capítulo 26).
   - Descrever métodos para a gestão de modificações (Capítulo 27).
9. Acompanhar o progresso regularmente (Seção 24.5.2).
   - Anotar áreas de problema (por exemplo, atraso de cronograma).
   - Fazer ajustes conforme necessário.

Quanto (How much) é necessário de cada recurso? A resposta a esta questão deriva do desenvolvimento de estimativas (Capítulo 23) baseadas nas respostas às questões anteriores.

As respostas às questões W^{5}HH de Boehm são importantes independentemente do tamanho ou da complexidade de um projeto de software. Contudo, como o processo de planejamento tem início?

> "Cremos que os desenvolvedores de software estão omitindo uma verdade vital: a maioria das organizações não sabe o que está fazendo. Elas pensam que sabem, mas não sabem."
>
> Tom DeMarco

5.4 PRÁTICA DE MODELAGEM

Criamos modelos para obter um melhor entendimento da entidade real a ser construída. Quando a entidade é uma coisa física (por exemplo, um edifício, avião ou máquina), podemos construir um modelo que é idêntico em forma e aspecto, mas menor em escala. No entanto, quando a entidade é software, nosso modelo precisa assumir uma forma diferente. Ele precisa ser capaz de representar a informação que o software transforma, a arquitetura e funções que permitem que essa transformação ocorra, as características que os usuários desejam e o comportamento do sistema à medida que a transformação ocorre. Os modelos precisam satisfazer a esses objetivos em diferentes níveis de abstração — primeiro mostrando o software do ponto de vista dos clientes e, depois, representando o software em nível mais técnico.

PONTO CHAVE

Modelos de análise representam os requisitos do cliente. Modelos de projeto fornecem uma especificação concreta para a construção do software.

No trabalho de engenharia de software, duas classes de modelo são criadas: modelos de análise e modelos de projeto. Os *modelos de análise* representam os requisitos do cliente mostrando o software em três domínios diferentes: o domínio de informação, o domínio funcional e o domínio comportamental. Os *modelos de projeto* representam características de software que ajudam os profissionais a construí-lo efetivamente: a arquitetura (Capítulo 10), a interface do usuário (Capítulo 12) e detalhes em nível de componentes (Capítulo 11).

Nas seções seguintes apresentamos princípios e conceitos básicos que são relevantes à modelagem de análise e projeto. Os métodos e notações técnicas que permitem aos engenheiros de software criar modelos de análise e projeto serão apresentados em capítulos posteriores.

> "O primeiro problema dos engenheiros em qualquer situação de projeto é descobrir qual é o verdadeiro problema."
>
> Autor desconhecido

5.4.1 Princípios da Modelagem de Análise

Durante as últimas três décadas, um grande número de métodos de modelagem de análise foi desenvolvido. Pesquisadores identificaram problemas de análise e suas causas, e desenvolveram uma variedade de notações de modelagem e conjuntos de heurísticas correspondentes para resolvê-los. Cada método de análise tem um ponto de vista singular. No entanto, todos os métodos de análise estão relacionados a um conjunto de princípios operacionais:

Princípio nº 1: ***O domínio de informação de um problema precisa ser representado e entendido.*** O *domínio de informação* abrange os dados que fluem para dentro do sistema (vindos dos usuários finais, de outros sistemas ou de dispositivos externos), os dados que fluem para fora do sistema (pela interface do usuário, por interfaces de rede, relatórios, gráficos e outros meios) e os depósitos de dados que coletam e organizam os objetos de dados persistentes (isto é, os dados que são mantidos permanentemente).

Princípio nº 2. ***As funções a serem desenvolvidas pelo software devem ser definidas.*** As funções do software fornecem benefício direto aos usuários finais e também dão apoio interno às características que são visíveis ao usuário. Algumas funções transformam os dados que fluem para

Document shared on www.docsity.com

PONTO CHAVE

A modelagem de análise enfoca três atributos do software: a informação a ser processada, a função a ser desempenhada e o comportamento a ser exibido.

dentro do sistema. Em outros casos, as funções efetuam algum nível de controle sobre o processamento interno do software ou sobre elementos externos do sistema. As funções podem ser descritas em vários níveis de abstração diferentes que vão desde uma declaração geral de objetivo até uma descrição detalhada dos elementos de processamento que precisam ser invocados.

Princípio nº 3. ***O comportamento do software (como conseqüência de eventos externos) precisa ser representado.*** O comportamento do software de computador é guiado por suas interações com o ambiente externo. Entradas fornecidas pelos usuários finais, dados de controle fornecidos por um sistema externo ou monitoramento de dados coletados em uma rede, todos fazem que o software se comporte de um modo específico.

Princípio nº 4. ***Os modelos que mostram informação, função e comportamento devem ser particionados de um modo que revele detalhes em forma de camadas (ou hierarquias).*** A modelagem de análise é o primeiro passo na solução de um problema de engenharia de software. Ela permite que os profissionais entendam melhor o problema e estabelece uma base para a solução (projeto). Problemas complexos são difíceis de serem resolvidos como um todo. Por isso, usamos uma estratégia de dividir e conquistar. Um problema complexo, grande, é dividido em subproblemas, até que cada subproblema seja relativamente fácil de entender. Esse conceito é chamado de ***particionamento*** e é uma estratégia-chave na modelagem de análise.

Princípio nº 5. ***A tarefa de análise deve ir da informação essencial até os detalhes de implementação.*** A modelagem de análise começa descrevendo um problema na perspectiva do usuário final. A "essência" do problema é descrita sem qualquer consideração de como uma solução será implementada. Por exemplo, um videogame exige que o jogador "instrua" seu protagonista sobre em qual direção prosseguir quando entra em um ambiente perigoso. Essa é a essência do problema. Os detalhes de implementação (normalmente, descritos como parte do modelo de projeto) indicam como a essência será implementada. No videogame, pode-se usar uma entrada de voz. Alternativamente, um comando de teclado pode ser digitado ou um *joystick* (ou *mouse*) pode ser apontado em uma direção específica.

CONJUNTO DE TAREFAS

Conjunto Genérico de Tarefas para Modelagem de Análise

1. Revisar os requisitos do negócio, as características/necessidades dos usuários, as saídas visíveis ao usuário, as restrições do negócio e outros requisitos técnicos que foram determinados durante as atividades de comunicação com o cliente e o planejamento.
2. Expandir e refinar os cenários de usuário (Seção 8.5).
   - Definir todos os atores.
   - Representar como os atores interagem com o software.
   - Extrair funções e características dos cenários de usuário.
   - Revisar os cenários de usuário quanto à completude e precisão (Seção 26.4).
3. Modelar o domínio da informação (Seção 8.3).
   - Representar todos os principais objetos de informação.
   - Definir atributos para cada objeto de informação.
   - Representar os relacionamentos entre os objetos de informação.
4. Modelar o domínio funcional (Seção 8.6).
   - Mostrar como as funções modificam os objetos de dados.
   - Refinar as funções para fornecer detalhes de elaboração.
   - Escrever uma narrativa de processamento que descreva cada função e subfunção.
   - Revisar os modelos funcionais (Seção 26.4).
5. Modelar o domínio comportamental (Seção 8.8).
   - Identificar os eventos externos que causam mudanças comportamentais no sistema.
   - Identificar os estados que representam cada modo de comportamento observável externamente.
   - Mostrar como um evento faz o sistema se mover de um estado para outro.
   - Revisar os modelos comportamentais (Seção 26.4).
6. Analisar e modelar a interface do usuário (Capítulo 12).
   - Conduzir tarefa de análise.
   - Criar protótipos de imagem de tela.
7. Revisar todos os modelos quanto à completude, consistência e omissões.

5.4.2 Princípios de Modelagem de Projeto

O modelo de projeto é equivalente às plantas de arquitetura de uma casa. Ele começa com a representação da totalidade do objeto a ser construído (por exemplo, um desenho em três dimensões da casa) e lentamente refina o objeto para fornecer diretrizes para a construção de cada detalhe (por exemplo, a disposição dos encanamentos). Analogamente, o modelo de projeto que é criado para o software fornece uma variedade de diferentes visões do sistema.

> "Cuide primeiro para que o projeto seja sábio e justo; garantido isso, siga-o resolutamente; nem por um instante perca de vista a finalidade que você resolveu atingir."
>
> William Shakespeare

Não faltam métodos para derivar os vários elementos de um projeto de software. Alguns métodos são guiados pelos dados, permitindo que a estrutura de dados determine a arquitetura do programa e os componentes de processamento resultantes. Outros são guiados por padrões, usando informações sobre o domínio do problema (o modelo de análise) para desenvolver estilos arquitetônicos e padrões de processamento. Outros, ainda, são orientados a objetos, usando objetos do domínio do problema como a diretriz para a criação das estruturas de dados e dos métodos que as manipulam. No entanto, todos obedecem a um conjunto de princípios de projeto que podem ser aplicados independentemente do método a ser usado.

Princípio nº 1: ***O projeto deve estar relacionado ao modelo de análise.*** O modelo de análise descreve o domínio de informação do problema, funções visíveis ao usuário, o comportamento do sistema e um conjunto de classes de análise que empacotam objetos do negócio com os métodos que os servem. O modelo de projeto traduz essa informação em uma arquitetura: um conjunto de subsistemas que implementam as funções principais e um conjunto de projetos em nível de componente que são a realização das classes de análise. Com exceção do projeto associado à infra-estrutura do software, os elementos do modelo de projeto devem estar relacionados ao modelo de análise.

Veja na Web

Comentários profundos sobre o processo de projeto, junto com uma discussão da estética do projeto, podem ser encontrados em cs.wwc.edu/~aabyan/Design/.

Princípio nº 2: ***Sempre considere a arquitetura do sistema a ser construído.*** A arquitetura do software (Capítulo 10) é o esqueleto do sistema a ser construído. Ela afeta as interfaces, estruturas de dados, fluxo de controle e comportamento do programa, o modo pelo qual o teste pode ser conduzido, a manutenibilidade do sistema resultante e muito mais. Por todas essas razões, o projeto deve começar com considerações arquiteturais. Apenas depois de a arquitetura ser estabelecida, os tópicos em nível de componente devem ser considerados.

Princípio nº 3: ***O projeto dos dados é tão importante quanto o projeto de funções de processamento.*** O projeto dos dados é um elemento essencial do projeto arquitetural. A maneira pela qual os objetos de dados são realizados no projeto não pode ser deixada ao acaso. Um projeto de dados bem estruturado ajuda a simplificar o fluxo do programa, torna o projeto e implementação dos componentes de software mais fáceis e deixa o processamento global mais eficiente.

Princípio nº 4: ***As interfaces (tanto externas quanto internas) precisam ser projetadas com cuidado.*** A maneira como os dados fluem entre os componentes de um sistema tem muito a ver com a eficiência de processamento, a propagação de erros e a simplicidade de projeto. Uma interface bem projetada torna a integração mais fácil e ajuda o testador na validação das funções dos componentes.

Princípio nº 5: ***O projeto de interface do usuário deve estar sintonizado com as necessidades do usuário final. No entanto, em cada caso, ele deve enfatizar a facilidade de uso.*** A interface do usuário é a manifestação visível do software. Não importa quão sofisticadas sejam suas funções internas, quão abrangentes suas estruturas de dados, quão bem projetada sua arquitetura, um projeto de interface pobre conduz, muitas vezes, à percepção de que o software é "ruim".

Princípio nº 6: ***O projeto em nível de componente deve ser funcionalmente independente.*** A independência funcional é uma medida da "objetividade" de um componente de software. A

Document shared on www.docsity.com

funcionalidade fornecida por um componente deve ser *coesiva* — isto é, deve enfocar uma e apenas uma função ou subfunção.[8]

Princípio nº 7: *Os componentes devem ser fracamente acoplados uns aos outros e ao ambiente externo.* O *acoplamento* é conseguido de muitos modos — via interface de componentes, por mensagens, por meio de dados globais. À medida que o nível de acoplamento aumenta, a probabilidade de propagação de erros também aumenta e a manutenibilidade global do software diminui. Assim, o acoplamento de componentes deve ser mantido tão baixo quanto for razoável.

Princípio nº 8: *Representações de projeto (modelos) devem ser facilmente compreensíveis.* O objetivo do projeto é comunicar a informação para os profissionais que vão gerar código, para aqueles que vão testar o software, e para outros que podem vir a manter o software no futuro. Se o projeto for difícil de entender, ele não servirá como um meio de comunicação efetivo.

Princípio nº 9: *O projeto deve ser desenvolvido iterativamente. A cada iteração, o projetista deve lutar por maior simplicidade.* Como quase todas as atividades criativas, o projeto ocorre iterativamente. As primeiras iterações trabalham para refinar o projeto e corrigir erros, mas as últimas iterações devem procurar tornar o projeto o mais simples possível.

Quando esses princípios são aplicados adequadamente, o engenheiro de software cria um projeto que exibe fatores de qualidade tanto externos quanto internos. *Fatores de qualidade externos* são aquelas propriedades do software que podem ser prontamente observadas pelos usuários (por exemplo, velocidade, confiabilidade, correção, usabilidade). *Fatores de qualidade internos* são importantes para os engenheiros de software. Eles levam a um projeto de alta qualidade dentro de uma perspectiva técnica. Para satisfazer aos fatores de qualidade internos, o projetista precisa entender os conceitos básicos de projeto (Capítulo 9).

INFO

Modelagem Ágil

Em seu livro sobre modelagem ágil, Scott Ambler [AMB02] define um conjunto de princípios[9] que são aplicáveis quando análise e projeto são conduzidos no contexto da filosofia de desenvolvimento ágil de software (Capítulo 4):

Princípio nº 1: O objetivo principal da equipe de software é construir softwares, e não criar modelos.

Princípio nº 2: Seja econômico — não crie mais modelos do que você precisa.

Princípio nº 3: Procure produzir o modelo mais simples que descreva o problema ou o software.

Princípio nº 4: Construa modelos de modo que sejam passíveis de mudanças.

Princípio nº 5: Seja capaz de declarar um objetivo específico para cada modelo criado.

Princípio nº 6: Adapte os modelos que você desenvolveu para o sistema em mãos.

Princípio nº 7: Tente construir modelos úteis, mas não pense em construir modelos perfeitos.

Princípio nº 8: Não seja dogmático em relação à sintaxe do modelo. Se ele comunica o conteúdo com sucesso, a representação é secundária.

Princípio nº 9: Se o seu instinto lhe diz que o modelo não está certo, mesmo que ele pareça correto no papel, você provavelmente tem razão em se preocupar.

Princípio nº 10: Obtenha o feedback tão logo quanto possível.

Independentemente do processo de modelagem escolhido ou das práticas específicas de engenharia de software aplicadas, toda a equipe de software quer ser ágil. Assim, esses princípios podem, e devem, ser aplicados independentemente do modelo de processo de software escolhido.

8 Discussão adicional sobre coesão poderá ser encontrada no Capítulo 9.
9 Os princípios exibidos nesta seção foram abreviados e adaptados para os objetivos deste livro.

CONJUNTO DE TAREFAS

Conjunto Genérico de Tarefas para Projeto

1. Usando o modelo de análise, selecionar um estilo (padrão) arquitetural que seja apropriado para o software (Capítulo 10).
2. Particionar o modelo de análise em subsistemas de projeto e alocar esses subsistemas na arquitetura (Capítulo 10).
   - Certificar-se de que cada subsistema seja funcionalmente coesivo.
   - Projetar as interfaces dos subsistemas.
   - Alocar classes ou funções de análise a cada subsistema.
   - Usando o modelo de domínio de informação, projetar as estruturas de dados adequadas.
3. Projetar a interface do usuário (Capítulo 12.)
   - Rever os resultados das análises de tarefas.
   - Especificar a seqüência de ações com base nos cenários dos usuários.
   - Criar o modelo comportamental da interface.
   - Definir objetos de interface e mecanismos de controle.
   - Rever o projeto de interface e revisar conforme a necessidade (Seção 26.4).
4. Conduzir o projeto em nível de componentes (Capítulo 11).
   - Especificar todos os algoritmos em um nível de abstração relativamente baixo.
   - Refinar a interface de cada componente.
   - Definir as estruturas de dados em nível de componente.
   - Rever o projeto em nível de componente (Seção 26.4).
5. Desenvolver um modelo de implantação (Seção 9.4.5).

5.5 PRÁTICA DE CONSTRUÇÃO

A atividade de *construção* compreende um conjunto de tarefas de codificação e teste que levam ao software operacional que está pronto para ser entregue ao cliente ou ao usuário final. No trabalho moderno de engenharia de software, a codificação pode ser: (1) a criação direta de código-fonte em linguagem de programação; (2) a geração automática de código-fonte usando uma representação intermediária análoga ao projeto do componente a ser construído; (3) a geração automática de código executável usando uma linguagem de programação de quarta geração (por exemplo, Visual C++).

> "Durante grande parte da minha vida, tenho sido um 'voyeur' de softwares, espiando furtivamente o código sujo de outras pessoas. Ocasionalmente, encontro uma verdadeira jóia, um programa bem estruturado, escrito em estilo consistente, livre de gambiarras, desenvolvido de modo que cada componente seja simples e organizado, e projetado de forma que o produto seja fácil de modificar."
>
> **David Parnas**

O enfoque inicial do teste é nos componentes, freqüentemente chamado de *teste unitário*. Outros níveis de teste incluem: (1) *teste de integração* (conduzido à medida que o sistema é construído); (2) *teste de validação* que avalia se os requisitos foram satisfeitos pelo sistema completo (ou incremento de software); e (3) *teste de aceitação* que é conduzido pelo cliente em um esforço de exercitar todas as características e funções necessárias.

Um conjunto fundamental de princípios e conceitos é aplicável à codificação e ao teste. Eles são considerados nas seções seguintes.

5.5.1 Princípios e Conceitos de Codificação

Os princípios e conceitos que dirigem a tarefa de codificação são estilo de programação, linguagens de programação e métodos de programação rigorosamente definidos. No entanto, há um número de princípios fundamentais que podem ser enunciados:

Document shared on www.docsity.com

AVISO

Evite desenvolver um programa elegante que resolva o problema errado. Preste particular atenção ao primeiro princípio de preparação.

Princípios de preparação: *Antes de escrever uma linha de código, certifique-se de:*

1. Entender o problema que está tentando resolver.
2. Entender os princípios e conceitos básicos do projeto.
3. Escolher uma linguagem de programação que satisfaça às necessidades do software a ser construído e do ambiente em que ele vai operar.
4. Selecionar um ambiente de programação que fornece ferramentas para facilitar o seu trabalho.
5. Criar um conjunto de testes unitários que será aplicado tão logo o componente que você está codificando seja completado.

Veja na Web

Uma grande variedade de links para normas de codificação pode ser encontrada em www.literateprogramming.com/fpstyle.html.

Princípios de codificação: *Quando começar a escrever o código, certifique-se de:*

1. Restringir os seus algoritmos seguindo a prática de programação estruturada [BOH00].
2. Selecionar estruturas de dados que atendam às necessidades do projeto.
3. Entender a arquitetura do software e criar interfaces que sejam consistentes com ela.
4. Conservar a lógica condicional tão simples quanto possível.
5. Criar ciclos aninhados de modo que sejam facilmente testáveis.
6. Selecionar nomes significativos de variáveis e seguir outras normas locais de codificação.
7. Escrever código que é autodocumentado.
8. Criar uma disposição visual (por exemplo, alinhamento e linhas em branco) que auxilie o entendimento.

Princípios de validação: *Depois completar seu primeiro passo de codificação, certifique-se de:*

1. Conduzir uma inspeção de código quando adequado.
2. Realizar testes unitários e corrigir os erros descobertos.
3. Refabricar o código.

Os livros sobre codificação e os princípios que a guiam incluem trabalhos pioneiros sobre estilo de programação [KER78], construção prática de softwares [MCC93], pérolas de programação [BEN99], a arte de programar [KNU99], tópicos de programação pragmática [HUN99] e muitos, muitos outros.

CONJUNTO DE TAREFAS

Conjunto Genérico de Tarefas para Construção

1. Construir a infra-estrutura arquitetural (Capítulo 10).
   - Revisar o projeto arquitetural.
   - Codificar e testar os componentes que compõem a infra-estrutura arquitetural.
   - Adquirir padrões arquiteturais reusáveis.
   - Testar a infra-estrutura para garantir a integridade da interface.
2. Construir um componente de software (Capítulo 11).
   - Revisar o projeto em nível de componentes.
   - Criar um conjunto de testes de unidade para o componente (Seções 13.3.1 e 14.7).
   - Codificar as estruturas de dados e interface do componente.
   - Codificar algoritmos internos e funções de processamento relacionadas.
   - Revisar o código à medida que ele é escrito (Seção 26.4).
   - Buscar a correção.
   - Garantir que as normas de codificação sejam mantidas.
   - Garantir que o código seja autodocumentado.
3. Fazer teste unitário do componente.
   - Conduzir todos os testes unitários.
   - Corrigir os erros descobertos.
   - Reaplicar os testes unitários.
4. Integrar o componente completo à infra-estrutura arquitetural.

5.5.2 Princípios de Teste

Em um livro clássico sobre teste de software, Glen Myers [MYE79] enumera algumas regras que podem servir bem como objetivos de teste:

? Quais são os objetivos do teste de software?

- Teste é um processo de execução de um programa com a finalidade de encontrar um erro.
- Um bom caso de teste é aquele que tem alta probabilidade de encontrar um erro ainda não descoberto.
- Um teste bem-sucedido é aquele que descobre um erro ainda não descoberto.

Esses objetivos implicam uma dramática mudança de ponto de vista para alguns desenvolvedores de software. Eles vão contra a visão comum de que um teste bem-sucedido é aquele no qual não são encontrados erros. Nosso objetivo é projetar testes que descubram sistematicamente diferentes classes de erros e fazer isso com uma quantidade mínima de tempo e de esforço.

Davis [DAV95] sugere um conjunto de princípios de teste[10] que foram adaptados para serem usados neste livro:

AVISO

Em um contexto de projeto de software mais amplo, lembre-se de que começamos "no atacado" enfocando a arquitetura do software, e acabamos "no varejo" enfocando os componentes. Para o teste nós simplesmente revertemos o enfoque e testamos na direção oposta.

Princípio nº 1: ***Todos os testes devem estar relacionados aos requisitos do cliente.***[11] O objetivo do teste de software é descobrir erros. Segue-se daí que os defeitos mais severos (do ponto de vista do cliente) são aqueles que fazem que o programa deixe de satisfazer aos seus requisitos.

Princípio nº 2: ***Os testes devem ser planejados muito antes de serem iniciados.*** O planejamento de testes (Capítulo 13) pode começar assim que o modelo de análise for completado. A definição detalhada dos casos de teste pode começar tão logo o modelo de projeto tenha sido consolidado. Assim sendo, todos os testes podem ser planejados e projetados antes que qualquer código tenha sido gerado.

Princípio nº 3: ***O princípio de Pareto se aplica ao teste de software.*** Colocado simplesmente, o princípio de Pareto implica que 80% de todos os erros descobertos durante o teste estarão, provavelmente, relacionados a 20% de todos os componentes do programa. O problema, sem dúvida, é isolar esses componentes suspeitos e testá-los rigorosamente.

CONJUNTO DE TAREFAS

Conjunto Genérico de Tarefas para Teste

1. Projetar testes de unidade para cada componente de software (Seção 3.3.1).
   - Revisar cada teste unitário para garantir a cobertura adequada.
   - Conduzir o teste unitário.
   - Corrigir os erros descobertos.
   - Reaplicar os testes unitários.
2. Desenvolver uma estratégia de integração (Seção 13.3.2).
   - Estabelecer a ordem de integração e a estratégia a ser usada para tanto.
   - Definir "construções" e os testes necessários para exercitá-las.
   - Conduzir o teste fumaça diariamente.
   - Conduzir testes de regressão quando necessário.
3. Desenvolver uma estratégia de validação (Seção 13.5).
   - Estabelecer critérios de validação.
   - Definir testes necessários para validar o software.
4. Conduzir os testes de integração e validação.
   - Corrigir os erros descobertos.
   - Reaplicar os testes quando necessário.
5. Conduzir os testes de alta prioridade.
   - Executar testes de recuperação (Seção 13.6.1).
   - Executar testes de segurança (Seção 13.6.2).
   - Executar testes de estresse (Seção 13.6.3).
   - Executar testes de desempenho (Seção 13.6.4).
6. Coordenar os testes de aceitação com o cliente (Seção 13.5.3).

10 Apenas um pequeno subconjunto dos princípios de teste de Davis é mencionado aqui. Para mais informações, veja [DAV95].

11 Este princípio refere-se aos testes *funcionais*, isto é, testes que enfocam os requisitos. Testes *estruturais* (testes que enfocam os detalhes arquiteturais ou lógicos) podem não se referir diretamente a requisitos específicos.

Document shared on www.docsity.com

Princípio nº 4: *O teste deve começar "no varejo" e progredir até o teste "no atacado".* Os primeiros testes planejados e executados geralmente concentram-se nos componentes individuais. À medida que o teste progride, o foco se desloca numa tentativa de encontrar erros em conjuntos integrados de componentes e, finalmente, em todo o sistema.

Princípio nº 5: *Testes exaustivos não são possíveis.* A quantidade de permutações de caminho, mesmo para um programa de tamanho moderado, é excepcionalmente grande. Por essa razão, é impossível executar todas as combinações de caminhos durante o teste. É possível, no entanto, cobrir adequadamente a lógica do programa e garantir que todas as condições do projeto, em nível de componente, tenham sido exercitadas (Capítulo 14).

5.6 Implantação

Como observamos no Capítulo 2, a atividade de implantação engloba três ações: entrega, suporte e feedback. Como os modelos modernos de processo de software são evolutivos por natureza, a implantação ocorre não uma única vez, mas várias vezes, à medida que o software caminha para ficar completo. Cada ciclo de entrega fornece aos clientes e usuários finais um incremento de software operacional com funções e características utilizáveis. Cada ciclo de suporte fornece documentação e apoio humano às funções e características introduzidas durante todos os ciclos de implantação até então. Cada ciclo de feedback fornece à equipe de software diretrizes importantes que resultam em modificações das funções, das características e da abordagem considerada no incremento seguinte.

A entrega de um incremento de software representa um marco importante para qualquer projeto de software. Alguns princípios-chave devem ser seguidos à medida que a equipe se prepara para entregar um incremento:

AVISO

Certifique-se de que seu cliente sabe o que esperar antes de um incremento de software ser entregue. Caso contrário, você pode apostar que o cliente vai esperar mais do que você entrega.

Princípio nº 1: *As expectativas do cliente quanto ao software devem ser geridas.* Muito freqüentemente, o cliente espera mais do que a equipe prometeu entregar e o desapontamento é imediato. Isso resulta em feedback não-produtivo e arruina o moral da equipe. Em seu livro sobre expectativas de gestão, Naomi Karten [KAR94] afirma: "O ponto inicial da gestão de expectativas é tornar-se mais consciente do que você comunica e de como o faz". Ela sugere que um engenheiro de software deve ser cuidadoso com o envio de mensagens conflitantes ao cliente (por exemplo, prometer mais do que você pode realmente entregar na data combinada ou entregar mais do que prometeu em um incremento de software e, depois, menos do que prometeu no seguinte).

Princípio nº 2: *Um pacote completo de entrega deve ser montado e testado.* Um CD-ROM ou outra mídia contendo todo o software executável, arquivos de dados de suporte, documentos de suporte e outras informações relevantes deve ser montado e rigorosamente testado por testes beta com usuários reais. Todos os documentos de instalação e outras características operacionais devem ser seriamente exercitados em todas as possibilidades de configuração de computação (isto é, hardware, sistemas operacionais, dispositivos periféricos, arranjos de redes).

Princípio nº 3: *Um regime de suporte deve ser estabelecido antes de o software ser entregue.* Um usuário final espera receptividade e informação segura quando uma questão ou um problema surge. Se o suporte é *ad hoc* ou, pior, inexistente, o cliente ficará insatisfeito imediatamente. O suporte deve ser planejado, o material de suporte preparado e mecanismos adequados de registro devem ser estabelecidos, de modo que a equipe de software possa conduzir uma avaliação categórica das espécies de suporte necessárias.

Princípio nº 4: *Materiais institucionais adequados devem ser fornecidos aos usuários finais.* A equipe de software entrega mais do que o software em si. Ajuda de treinamento adequado (se necessária) deve ser desenvolvida, diretrizes de depuração devem ser fornecidas e uma descrição de "o-que-é-diferente-nesse-incremento-de-software" deve ser publicada.[12]

12 Durante a atividade de comunicação, a equipe de software deve determinar quais tipos de materiais de auxílio os usuários desejam.

Princípio nº 5: *Software defeituoso deve ser corrigido primeiro e, depois, entregue.* Pressionadas pelo tempo, algumas organizações de software entregam incrementos de baixa qualidade com um aviso ao cliente de que os defeitos "serão consertados na versão seguinte". Isso é um erro. Há um ditado no negócio de software que diz: "Os clientes esquecerão que você entregou um produto de alta qualidade alguns dias depois, mas eles nunca esquecerão os problemas que um produto de baixa qualidade lhes causou. O software os lembra a cada dia".

O software entregue fornece benefício ao usuário final, mas também fornece *feedback* útil para a equipe de software. Quando um incremento é colocado em uso, os usuários finais devem ser encorajados a comentar as características e funções, facilidade de uso, confiabilidade e quaisquer outras características que sejam adequadas. O feedback deve ser coletado e registrado pela equipe de software, e usado para (1) fazer modificações imediatas no incremento entregue (se necessário); (2) definir modificações a ser incorporadas no próximo incremento planejado; (3) fazer as modificações de projeto necessárias para acomodar as alterações; e (4) revisar o plano (inclusive cronograma de entrega) para que o próximo incremento reflita as modificações.

Conjunto de Tarefas

Conjunto Genérico de Tarefas de Implantação

1. Criar mídia de entrega.
   - Montar e testar todos os arquivos executáveis.
   - Montar e testar todos os arquivos de dados.
   - Criar e testar toda a documentação do usuário.
   - Implementar versões eletrônicas (por exemplo, pdf).
   - Implementar arquivos hipertextos de "ajuda".
   - Implementar um guia de correção de erros.
   - Testar a mídia entregue com um pequeno grupo de usuários representativos.
2. Estabelecer suporte humano pessoal ou em grupo.
   - Criar documentação e/ou ferramentas de suporte por computador.
   - Estabelecer mecanismos de contato (por exemplo, site, telefone, e-mail).
   - Estabelecer mecanismos de registro de problemas.
   - Estabelecer mecanismos de relato de problemas.
   - Estabelecer banco de dados de relato de problemas/erros.
3. Estabelecer mecanismos de feedback de usuários.
   - Definir o processo de feedback.
   - Definir formas de feedback (papel e eletrônica).
   - Estabelecer banco de dados de feedback.
   - Definir processo de avaliação de feedback.
4. Disseminar a mídia de entrega entre todos os usuários.
5. Conduzir funções de suporte permanentes.
   - Fornecer assistência de instalação e início.
   - Fornecer assistência contínua para correção de erros.
6. Coletar feedback do usuário.
   - Registrar feedback.
   - Avaliar feedback.
   - Comunicar-se com os usuários sobre o feedback.

5.7 Resumo

A prática de engenharia de software engloba conceitos, princípios, métodos e ferramentas que engenheiros de software aplicam durante o processo de software. Cada projeto de engenharia de software é diferente; no entanto, um conjunto de princípios e tarefas genéricas aplica-se a cada atividade de arcabouço de processo independentemente do projeto ou do produto.

Um conjunto de princípios básicos técnicos e de gestão é necessário se a boa prática de engenharia de software tiver de ser conduzida. Princípios básicos técnicos incluem a necessidade de entender áreas de incerteza de requisitos e de protótipos, e a necessidade de definir explicitamente a arquitetura do software e planejar a integração dos componentes. Princípios básicos de gestão incluem a necessidade de definir prioridades e um cronograma realístico que as reflita, a necessidade de gerir riscos ativamente, e a necessidade de definir medidas de controle de projeto adequadas quanto a qualidade e a modificações.

Document shared on www.docsity.com

Os princípios de comunicação com o cliente focalizam a necessidade de reduzir o ruído e aumentar a largura de banda à medida que a conversação entre desenvolvedor e cliente progride. Ambas as partes devem colaborar para que uma melhor comunicação ocorra.

Todos os princípios de planejamento enfocam em diretrizes para construir o melhor caminho para a obtenção de um sistema ou produto completo. O plano pode ser projetado somente para um único incremento de software, ou pode ser definido para o projeto todo. Independentemente disso, ele deve incluir o que será feito, quem irá fazê-lo e quando o trabalho será completado.

A modelagem engloba tanto a análise quanto o projeto, descrevendo representações do software que se tornam progressivamente mais detalhadas. A intenção dos modelos é consolidar o entendimento do trabalho a ser feito e fornecer diretrizes técnicas para aqueles que implementarão o software.

A construção incorpora um ciclo de codificação e teste no qual o código-fonte de um componente é gerado e testado para descobrir erros. A integração combina componentes individuais e envolve uma série de testes que enfocam funções globais e tópicos locais de interface. Princípios de codificação definem ações genéricas que devem ocorrer antes que o código seja escrito, enquanto ele estiver sendo criado e depois que ele tiver sido completado. Embora existam muitos princípios de teste, somente um é dominante: teste é o processo de execução de um programa com o objetivo de encontrar um erro.

Durante um desenvolvimento de software evolutivo, a implantação ocorre a cada incremento de software que é apresentado ao cliente. Princípios-chave para a entrega consideram a gestão das expectativas do cliente e lhe fornecem as informações de suporte apropriadas para o software. O suporte exige preparação antecipada. O feedback permite ao cliente sugerir modificações que tenham valor de negócio e forneçam ao desenvolvedor entrada para o ciclo iterativo seguinte da engenharia de software.

Referências Bibliográficas

[AMB02] Ambler, S. e Jeffries, R., *Agile Modeling*, Wiley, 2002.

[BEN99] Bentley, J., *Programming Pearls*, 2ª ed., Addison-Wesley, 1999.

[BOE96] Boehm, B., "Anchoring the Software Process", *IEEE Software*, v. 13, n. 4, jul. 1996, p. 73-82.

[BOH00] Bohl, M., e Rynn, M., *Tools for Structured Design: An Introduction to Programming Logic*, 5ª ed., Prentice-Hall, 2000.

[DAV95] Davis, A., *201 Principles of Software Development*, McGraw-Hill, 1995.

[FOW99] Fowler, M., *et al.*, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[GAR95] Garlan, D. e Shaw, M., "An Introduction to Software Architecture", *Advances in Software Engineering and Knowledge Engineering*, v. I (V. Ambriola e G. Tortora, eds.), World Scientific Publishing Company, 1995.

[HIG00] Highsmith, J., *Adaptive Software Development: An Evolutionary Approach to Managing Complex Systems*, Dorset House Publishing, 2000.

[HOO96] Hooker, D., "Seven Principles of Software Development", set. 1996, disponível em http://c2.com/cgi/wikiSevenPrinciplesOfSoftwareDevelopment.

[HUN95] Hunt, D., Bailey, A. e Taylor, B., *The Art of Facilitation*, Perseus Book Group, 1995.

[HUN99] Hunt, A., Thomas, D., e Cunningham, W., *The Pragmatic Programmer*, Addison-Wesley, 1999.

[JUS99] Justice, T., *et al.*, *The Facilitator's Fieldbook*, AMACOM, 1999.

[KAN93] Kaner, C., Falk, J. e Nguyen, H. Q., *Testing Computer Software*, 2ª ed., Van Nostrand-Reinhold, 1993.

[KAN96] Kaner, S., *et al.*, *The Facilitator's Guide to Preparatory Decision Making*, New Society Publishing, 1996.

[KAR94] Karten, N., *Managing Expectations*, Dorset House, 1994.

[KER78] Kernighan, B. e Plauger, P., *The Elements of Programming Style*, 2ª ed., McGraw-Hill, 1978.

[KNU98] Knuth, D., *The Art of Computer Programming*, 3 vols, Addison-Wesley, 1998.

[MCC93] McConnell, S., *Code Complete*, Microsoft Press, 1993.

[MCC97] _____, "Software's Ten Essentials", *IEEE Software*, v. 14, n. 2, mar./abr., 1997, p. 143-44.

[MYE78] Myers, G., *Composite Structured Design*, Van Nostrand, 1978.

[MYE79] _____, *The Art of Software Testing*, Wiley, 1979.

[PAR72] Parnas, D. L., "On Criteria to Be Used in Decomposing Systems into Modules", *CACM*, v. 14, n. 1, abr.1972, p. 221-27.

[POL45] Polya, G., *How to Solve It*, Princeton University Press, 1945.

[ROS75] Ross, D., Goodenough, J. e Irvine, C., "Software Engineering: Process, Principles and Goals", *IEEE Computer*, v. 8, n. 5, maio 1975.

[SHA95a] Shaw, M. e Garlan, D., "Formulations and Formalisms in Software Architecture," *Volume 1000 — Lecture Notes in Computer Science*, Springer-Verlag, 1995.

[SHA95b] _____, *et al.*, "Abstractions for Software Architecture and Tools to Support Them", *IEEE Trans. Software Engineering*, v. SE-21, n. 4,abr.1995, p. 314-35.

[STE74] Stevens, W., Myers, G. e Constantine, L., "Structured Design", *IBM Systems Journal*, v. 13, n. 2, 1974, p. 115-39.

[TAY90] Taylor, D. A., *Object-Oriented Technology: A Manager's Guide*, Addison-Wesley, 1990.

[ULL97] Ullman, E., *Close to the Machine: Technophilia and its Discontents*, City Lights Books, 1997.

[WIR71] Wirth, N., "Program Development by Stepwise Refinement", *CACM*, v. 14, n. 4, 1971, p. 221-27.

[WOO95] Wood, J., e Silver, D., *Joint Application Design*, Wiley, 1995.

[ZAH90] Zahniser, R. A., "Building Software in Groups", *American Programmer*, v. 3, n. 7-8, jul./ago. 1990.

Problemas e Pontos a Considerar

5.1. Tente resumir os "Sete Princípios de Desenvolvimento de Software" de David Hooker (Seção 5.1) em um breve parágrafo. Tente refinar suas diretrizes em poucas sentenças, sem usar as palavras dele.

5.2. Há outros "princípios básicos" técnicos que podem ser recomendados para a engenharia de software? Enuncie cada um deles e explique por que você os incluiu.

5.3. Há outros "princípios básicos" de gestão que podem ser recomendados para a engenharia de software? Enuncie cada um deles e explique por que você os incluiu.

5.4. Um importante princípio de comunicação estabelece "prepare-se antes de fazer a comunicação". Como essa preparação deve se manifestar no trabalho inicial que você faz? Que produtos de trabalho podem resultar como conseqüência dessa preparação antecipada?

5.5. Faça alguma pesquisa sobre "facilitação" para a atividade de comunicação (use as referências fornecidas ou outras) e prepare um conjunto de diretrizes que enfoquem somente a facilitação.

5.6. Como a comunicação ágil difere da comunicação tradicional de engenharia de software? Em que é similar?

5.7. Por que é necessário "prosseguir"?

5.8. Faça alguma pesquisa sobre "negociação" para a atividade de comunicação e prepare um conjunto de diretrizes que focalize somente a negociação.

5.9. Descreva o que significa *granularidade* no contexto de um cronograma de projeto.

5.10. Por que os modelos são importantes no trabalho de engenharia de software? Eles são sempre necessários? Existem qualificadores para a sua resposta sobre a necessidade?

5.11. Quais os três "domínios" considerados durante a modelagem de análise?

5.12. Tente acrescentar um princípio adicional àqueles estabelecidos para a codificação na Seção 5.6.

5.13. O que é um teste bem-sucedido?

5.14. Você concorda ou discorda da seguinte afirmação: "Como entregamos múltiplos incrementos para o cliente, por que devemos nos preocupar com a qualidade nos incrementos iniciais — podemos corrigir problemas nas iterações posteriores."? Explique sua resposta.

5.15. Por que o feedback é importante para a equipe de software?

Leituras e Fontes de Informação Adicionais

A comunicação com o cliente é uma atividade criticamente importante em engenharia de software, no entanto, poucos profissionais gastam tempo lendo sobre isso. Os livros de Pardee (*To Satisfy and Delight Your Customer*, Dorset House, 1996) e de Karten [KAR94] fornecem muitas informações sobre métodos para a interação efetiva com o cliente. Os conceitos e princípios de comunicação e planejamento são considerados em muitos livros sobre gestão de projetos. Ofertas úteis sobre gestão de projetos incluem: Hughs e Cotterell (*Software Project Management*, 2ª ed., McGraw-Hill, 1999), Phillips (*The Software Project Manager's Handbook*,

Document shared on www.docsity.com

IEEE Computer Society Press, 1998), McConnell (Software Project Survival Guide, Microsoft Press, 1998), e Gilb (*Principles of Software Engineering Management*, Addison-Wesley, 1988).

Praticamente todos os livros sobre engenharia de software contêm uma discussão útil sobre os conceitos e princípios de análise, projeto e teste. Entre as melhores ofertas estão os livros de Endres e seus colegas (*Handbook of Software and Systems Engineering*, Addison-Wesley, 2003), Sommerville (*Software Engineering*, 6ª ed., Addison Wesley, 2000), Pfleeger (*Software Engineering: Theory and Practice*, Prentice-Hall, 2001) e Schach (*Object-Oriented and Classical Software Engineering*, McGraw-Hill, 2001). Uma excelente coleção dos princípios de engenharia de software foi compilada por Davis [DAV95].

Os conceitos e princípios de modelagem são considerados em muitos livros dedicados à análise de requisitos e/ou projeto de software. Young (*Effective Requirements Practices*, Addison-Wesley, 2001) enfatiza uma "equipe conjunta" de clientes e desenvolvedores que desenvolvem colaborativamente os requisitos. Weigers (*Software Requirements*, Microsoft Press, 1999) apresenta muitas práticas-chave de engenharia de requisitos e de gestão de requisitos. Somerville e Kotonya (*Requirements Engineering: Processes and Techniques*, Wiley, 1998) discutem os conceitos e técnicas de "levantamento" e outros princípios de engenharia de requisitos.

O livro de Norman (*The Design of Everyday Things*, Currency/Doubleday, 1990) é leitura obrigatória para todo engenheiro de software que pretenda fazer trabalhos de projeto. Winograd e seus colegas (*Bringing Design to Software*, Addison-Wesley, 1996) editaram uma excelente coleção de ensaios que tratam de tópicos práticos de projeto de software. Constantine e Lockwood (*Software for Use*, Addison-Wesley, 1999) apresentam os conceitos associados ao "projeto centrado no usuário". Tognazzini (*Tog on Software Design*, Addison-Wesley, 1995) apresenta uma valiosa discussão filosófica sobre a natureza do projeto e o impacto das decisões na qualidade e na habilidade de uma equipe de produzir softwares que tenham alto valor para o cliente.

Centenas de livros tratam de um ou mais elementos da atividade de construção. Kernighan e Plauger [KER78] escreveram um texto clássico sobre estilo de programação, McConnell [MCC93] apresenta diretrizes pragmáticas para a construção prática de softwares, Bentley [BEN99] sugere uma ampla variedade de pérolas de programação, Knuth [KNU98] escreveu uma série clássica, em três volumes, sobre a arte de programar e Hunt [HUN99] sugere diretrizes pragmáticas de programação. A literatura sobre testes tem crescido nas últimas décadas. Myers [MYE79] permanece sendo um clássico. Os livros de Whittaker (*How to Break Software*, Addison-Wesley, 2002), Kaner e seus colegas (*Lessons Learned in Software Testing*, Wiley, 2001) e Marick (*The Craft of Software Testing*, Prentice-Hall, 1997) apresentam, cada um deles, importantes conceitos e princípios de teste e muitas diretrizes pragmáticas.

Uma ampla variedade de fontes de informação sobre práticas de engenharia de software está disponível na Internet. Uma lista atualizada de referências da World Wide Web que são relevantes pode ser encontrada no site deste livro: **http://www.mhhe.com/pressman**.

ENGENHARIA DE SISTEMAS

CAPÍTULO 6

CONCEITOS-CHAVE

macroelementos 101
EPN – engenharia de processo de negócio..... 104
engenharia de produto ... 105
gabaritos 108
modelos UML 110
sistemas:..... 100
arquitetura..... 105
características..... 101
elementos 100
hierarquia 101
modelagem 108
simulação..... 104
modelagem..... 108

Há quase 500 anos, Maquiavel disse: "Não há nada mais difícil de levar, mais perigoso de conduzir ou de sucesso mais incerto do que tomar a liderança na introdução de uma nova ordem das coisas". Durante os últimos 50 anos, os sistemas baseados em computador introduziram uma nova ordem. Apesar de a tecnologia ter feito grandes avanços desde essa citação de Maquiavel, suas palavras continuam a soar como verdadeiras.

A engenharia de software ocorre como conseqüência de um processo chamado *engenharia de sistemas*. Em vez de se concentrar somente no software, a engenharia de sistemas focaliza diversos elementos, analisando, projetando e organizando-os em um sistema que pode ser um produto, um serviço ou uma tecnologia para a transformação da informação ou do controle.

O processo de engenharia de sistemas toma diferentes formas dependendo do domínio de aplicação em que ele é aplicado. A *engenharia de processo de negócio* é conduzida quando o contexto do trabalho de engenharia focaliza uma empresa de negócios. Quando um produto (nesse contexto, um produto inclui tudo, de um telefone sem fio a um sistema de controle de tráfego aéreo) deve ser construído, o processo é chamado *engenharia de produto*.

Tanto a engenharia de processo de negócio quanto a engenharia de produto tentam colocar ordem no desenvolvimento de sistemas baseados em computador. Apesar de cada uma delas estar focalizada em um domínio de aplicação diferente, ambas buscam contextualizar o software. Isto é, tanto a engenharia de processo de negócio quanto a engenharia de produto[1] trabalham para atribuir um papel ao software de computador e, ao mesmo tempo, estabelecer as ligações que amarram o software a outros elementos de um sistema baseado em computador.

Neste capítulo nos concentraremos nos aspectos gerenciais e nas atividades específicas de processo que permitem a uma organização de software garantir que está fazendo as coisas certas, na hora certa e do modo certo.

PANORAMA

O que é? Antes que o software possa ser submetido a engenharia o "sistema" no qual ele reside deve ser entendido. Para conseguir isso, o objetivo geral do sistema deve ser determinado: o papel do hardware, software, pessoal, base de dados, procedimentos e outros elementos do sistema devem ser identificados; e requisitos operacionais devem ser conseguidos, analisados, especificados, modelados, validados e gerenciados. Essas atividades são a base da engenharia de sistemas.

Quem faz? Um engenheiro de sistemas trabalha para entender os requisitos do sistema, interagindo com o cliente, futuros usuários e outros interessados.

Por que é importante? Há um provérbio antigo que diz: "Você não pode ver a floresta pelas árvores". Nesse contexto a "floresta" é o sistema e as árvores são os elementos tecnológicos (incluindo o software), que são necessários para formá-lo. Se você se apressar a construir os elementos tecnológicos antes de entender o sistema, vai sem dúvida cometer erros que irão desapontar seu cliente. Antes de se preocupar com as árvores, entenda a floresta.

Quais são os passos? Objetivos e requisitos operacionais mais detalhados são identificados através de informação do cliente; requisitos são analisados para avaliar sua clareza, completeza e consistência; uma especificação, fre-

1 Na verdade, o termo *engenharia de sistemas* é freqüentemente usado nesse contexto. No entanto, para as finalidades deste livro, a engenharia de sistemas é genérica e usada para englobar tanto a engenharia de processo de negócio como a engenharia de produto.

Document shared on www.docsity.com

qüentemente incorporando um modelo do sistema, é criada e depois validada tanto pelos profissionais quanto pelos clientes. Finalmente, os requisitos são gerenciados para garantir que as modificações sejam controladas adequadamente.

Qual é o produto do trabalho? Uma representação efetiva do sistema deve ser produzida como conseqüência da engenharia de sistemas. Ela pode ser um protótipo, uma especificação ou até um modelo simbólico, mas deve comunicar as características operacionais, funcionais e comportamentais do sistema a ser construído e dar uma idéia da arquitetura do sistema.

Como tenho certeza de que fiz corretamente? Revise todos os produtos do trabalho de engenharia de sistemas quanto a clareza, completeza e consistência. Igualmente importante, espere modificações nos requisitos do sistema e as gerencie usando métodos sólidos de gestão de modificações (Capítulo 27).

6.1 Sistemas Baseados em Computador

A palavra *sistema* é possivelmente o termo mais usado e abusado do vocabulário técnico. Falamos de sistemas políticos e sistemas educacionais, de sistemas de aviação e sistemas de fabricação, de sistemas bancários e sistemas metroviários. A palavra nos diz pouco. Usamos o adjetivo que descreve o sistema para entender o contexto no qual a palavra é usada. O *Webster Dictionary* define sistema do seguinte modo:

> 1. Um conjunto ou arranjo de coisas relacionadas com o objetivo de formar uma unidade ou um todo orgânico; 2. um conjunto de fatos, princípios, regras etc., classificados e organizados de modo ordenado, de forma a mostrar um plano lógico ligando as várias partes; 3. método ou plano de classificação ou arranjo; 4. um modo estabelecido de fazer alguma coisa; método; procedimento...

Cinco definições adicionais são fornecidas no dicionário, embora nenhum sinônimo preciso seja sugerido. *Sistema* é uma palavra especial. Com base na definição do Webster, definimos um *sistema baseado em computador* como

> Um conjunto ou arranjo de elementos organizados para atingir alguma meta predefinida por meio do processamento da informação.

A meta pode ser apoiar alguma função de negócio ou desenvolver um produto que possa ser vendido para gerar receita. Para alcançar a meta, um sistema baseado em computador faz uso de diversos elementos do sistema:

Software. Programas de computador, estruturas de dados e produtos de trabalho correlacionados que servem para realizar o método lógico, procedimento ou controle necessário.

Hardware. Dispositivos eletrônicos que fornecem capacidade computacional, dispositivos de interconectividade (por exemplo, computadores de rede, dispositivos de telecomunicações) que possibilitam o fluxo de dados e dispositivos eletromecânicos (por exemplo, sensores, motores, bombas) que fornecem as funções do mundo externo.

Pessoal. Usuários e operadores de hardware e de software.

Banco de dados. Uma coleção de informações grande e organizada à qual se tem acesso por intermédio de software e persiste ao longo do tempo.

Documentação. Informações descritivas (por exemplo, modelos, especificações, manuais impressos, arquivos de ajuda on-line, sites) que mostram o uso e/ou operação do sistema.

Procedimentos. Os passos que definem o uso específico de cada elemento do sistema ou o contexto de procedimento no qual o sistema reside.

AVISO

Não se deixe enganar de modo a assumir um ponto de vista "centrado no software". Comece considerando todos os elementos do sistema antes de se concentrar no software.

Esses elementos se combinam de diversos modos para transformar a informação. Por exemplo, um departamento de vendas transforma dados brutos de venda em um perfil do comprador típico de um produto; um robô transforma um arquivo de comando contendo instruções específicas em um conjunto de sinais de controle que provoca alguma ação física específica. A criação de um sistema de informação para apoiar o departamento de vendas e a criação de um software de controle para apoiar o robô precisam, ambas, da engenharia de sistemas.

> "Nunca confie em um computador que você não pode jogar pela janela."
>
> Steve Wozniak

Uma característica complicadora dos sistemas baseados em computador é que os elementos que constituem um sistema também podem representar um macroelemento de um sistema ainda maior. O macroelemento é um sistema baseado em computador que faz parte de um sistema baseado em computador ainda maior. Como exemplo, considere um "*sistema de automação de fábrica*", que é essencialmente uma hierarquia de sistemas. No nível mais baixo da hierarquia temos uma máquina de controle numérico, robôs e dispositivos de entrada de dados. Cada um é, isoladamente, um sistema baseado em computador. Os elementos da máquina de controle numérico incluem hardware eletrônico e eletromecânico (por exemplo, processador e memória, motores, sensores), softwares (de comunicação e de controle de máquina), pessoal (o operador da máquina), um banco de dados (o programa de controle numérico armazenado), documentação e procedimentos. Uma decomposição semelhante poderia ser aplicada ao robô e ao dispositivo de entrada de dados. Cada um é um sistema baseado em computador.

PONTO CHAVE

Sistemas complexos são, na verdade, uma hierarquia de macroelementos que são sistemas em si próprios.

No nível de hierarquia seguinte, uma célula de fabricação é definida. A *célula de fabricação* é um sistema baseado em computador que pode ter elementos em si mesmo (por exemplo, computadores, dispositivos mecânicos) e também integra os macroelementos que chamamos máquina de controle numérico, robô e dispositivo de entrada de dados.

Para resumir, a célula de fabricação e seus macroelementos são, cada qual, compostos de elementos de sistema com rótulos genéricos: software, hardware, pessoal, banco de dados, procedimentos e documentação. Em alguns casos, os macroelementos compartilham um elemento genérico. Por exemplo, o robô e a máquina de controle numérico poderiam ser ambos administrados por um único operador (o elemento pessoal). Em outros casos, elementos genéricos são exclusivos de um sistema.

O papel do engenheiro de sistemas é definir os elementos de um sistema baseado em computador específico no contexto da hierarquia global de sistemas (macroelementos). Nas seções seguintes, examinaremos as tarefas que constituem a engenharia de sistemas de computador.

6.2 A Hierarquia da Engenharia de Sistemas

Veja na Web

O Conselho Internacional de Engenharia de Sistemas (INCOSE – International Council of System Engineering) fornece muitos recursos úteis em www.incose.org.

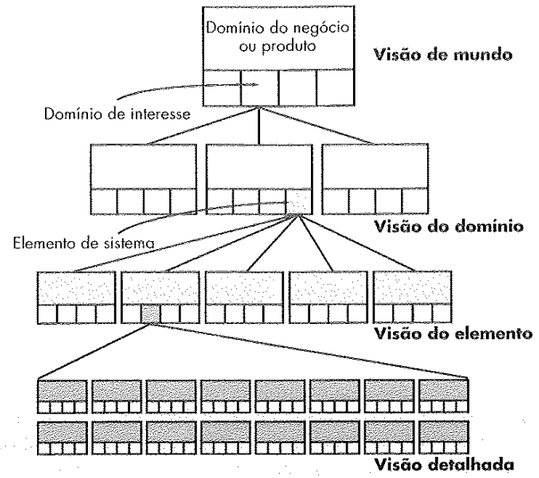
Independentemente do seu domínio de enfoque, a engenharia de sistemas abrange uma coleção de métodos descendentes (*top-down*) e ascendentes (*bottom-up*) para navegar na hierarquia ilustrada na Figura 6.1. O processo de engenharia de sistemas usualmente começa com uma "visão do mundo". Isto é, todo o domínio do negócio ou do produto é examinado para garantir que o contexto adequado do negócio ou da tecnologia possa ser estabelecido. A visão do mundo é refinada para focalizar mais completamente um domínio de interesse específico. Dentro do domínio específico, a necessidade de elementos para o sistema-alvo (por exemplo, dados, software, hardware e pessoal) é analisada. Finalmente, a análise, projeto e construção de um elemento do sistema-alvo é iniciada. No topo da hierarquia, um contexto amplo é estabelecido e, na base são conduzidas, atividades técnicas detalhadas, realizadas pela disciplina de engenharia pertinente (por exemplo, engenharia de hardware ou de software).[2]

2 Em algumas situações, no entanto, os engenheiros de sistemas devem considerar primeiramente os elementos individuais do sistema. Usando essa abordagem, os subsistemas são descritos de modo ascendente, considerando primeiro os componentes detalhados que constituem o subsistema.

Document shared on www.docsity.com

FIGURA 6.1

A hierarquia da engenharia de sistemas



PONTO CHAVE

A boa engenharia de sistemas começa com um entendimento claro do contexto — a visão de mundo — e depois estreita progressivamente o foco até que o detalhe técnico seja entendido.

Dito de uma maneira um tanto mais formal, a *visão de mundo* (*world view*, WV) é composta de um conjunto de domínios (D_i), que pode, por sua vez, ser um sistema ou sistema de sistemas:

$$WV = \{D_1, D_2, D_3, \ldots, D_n\}$$

Cada domínio é composto de *elementos* específicos (E_j), que desempenham individualmente algum papel para atingir o objetivo e as metas do domínio ou do componente:

$$D_i = \{E_1, E_2, E_3, \ldots, E_m\}$$

Finalmente, cada elemento é implementado especificando os *componentes* técnicos (C_k), que realizam a função necessária para um elemento:

$$E_j = \{C_1, C_2, C_3, \ldots, C_k\}$$

No contexto de software, um componente pode ser um programa de computador, um componente de programa reutilizável, um módulo, uma classe ou objeto, ou até um comando de uma linguagem de programação.

> "Sempre projete uma coisa considerando-a no seu contexto imediatamente superior — uma cadeira em uma sala, uma sala em uma casa, uma casa em um bairro, um bairro na planta de uma cidade."
>
> **Eliel Saarinen**

É importante notar que o engenheiro de sistemas estreita o foco do trabalho à medida que se move de forma descendente na hierarquia anteriormente descrita. No entanto, a visão de mundo retrata uma definição clara da funcionalidade global que permitirá ao engenheiro entender o domínio e, em última análise, o sistema ou produto, no contexto adequado.

6.2.1 Modelagem de Sistemas

A *modelagem de sistemas* é um elemento importante do processo de engenharia de sistemas. Quer o foco esteja na visão de mundo ou em uma visão detalhada, o engenheiro cria modelos que [MOT92]:

- Definem os processos, servindo às necessidades da visão que está sendo considerada.
- Representam o comportamento dos processos e os pressupostos nos quais o comportamento está baseado.

Document shared on www.docsity.com

? O que um modelo de engenharia de sistemas realiza?

- Definem explicitamente tanto entradas exógenas quanto endógenas[3] para o modelo.
- Representam todas as ligações (inclusive saídas), que permitirão ao engenheiro entender melhor a visão.

Para construir um modelo de sistema, o engenheiro deve considerar um certo número de fatores restritivos:

1. *Pressupostos* que reduzem a quantidade de permutações e variações possíveis, permitindo assim a um modelo refletir o problema de modo razoável. Considere, por exemplo, um produto de acabamento de desenho tridimensional usado pela indústria de entretenimento para criar animações realísticas. Um domínio do produto permite a representação de formas humanas em três dimensões. A entrada para esse domínio abrange a habilidade de especificar o movimento de um ator humano ao vivo, por vídeo ou pela criação de modelos gráficos. O engenheiro de sistemas faz algumas pressuposições sobre o âmbito do movimento humano permissível (por exemplo, pernas não podem se entrelaçar em volta do tronco) de modo que a possibilidade de variação das entradas e do processamento possa ser limitada.
2. *Simplificações* que permitem que o modelo seja criado no prazo adequado. Para ilustrar, considere uma empresa de produtos de escritório que vende e dá manutenção para uma ampla gama de copiadoras, scanners e equipamentos relacionados. O engenheiro de sistemas está modelando as necessidades da empresa de manutenção, trabalhando para entender o fluxo de informação envolvido em uma ordem de serviço. Apesar de uma ordem de serviço poder vir de diferentes origens, o engenheiro categoriza apenas duas fontes: demanda interna e solicitação externa. Isso possibilita uma partição simplificada das entradas necessárias para gerar a ordem de serviço.

   Um engenheiro de sistemas considera os seguintes fatores quando desenvolve soluções alternativas: pressupostos, simplificações, limitações, restrições e preferências do cliente.
3. *Limitações* que ajudam a delimitar o sistema. Por exemplo, um sistema de aviônica de uma aeronave que esteja modelado para uma aeronave de próxima geração. Como a aeronave vai seguir o projeto de bimotor, o domínio de monitoração da propulsão será modelado para acomodar um máximo de dois motores e os sistemas redundantes associados.
4. *Restrições* que guiarão o modo pelo qual o modelo é criado e a abordagem adotada quando o modelo é implementado. Por exemplo, a infra-estrutura tecnológica do sistema de acabamento de desenho tridimensional descrito previamente usa processadores duais baseados em G5. A complexidade computacional dos problemas deve ser restrita para se acomodar aos limites de processamento impostos pelos processadores.
5. *Preferências* que indicam a arquitetura preferida para todos os dados, funções e tecnologia. A solução preferida entra em conflito, algumas vezes, com outros fatores restritivos. No entanto, a satisfação do cliente é freqüentemente prejudicada em relação ao grau em que a abordagem preferida é realizada.

O modelo de sistema resultante (em qualquer visão) pode adotar uma solução completamente automatizada, uma solução semi-automatizada ou uma abordagem não automatizada. Na realidade, é freqüentemente possível caracterizar modelos de cada tipo que servem como soluções alternativas para o problema em mãos. Na essência, o engenheiro de sistemas simplesmente modifica a influência relativa de diferentes elementos do sistema (pessoal, hardware e software) para derivar modelos de cada tipo.

> "Coisas simples devem ser simples. Coisas complexas devem ser possíveis."
>
> **Allan Kay**

3 Entradas *exógenas* ligam uma parte de uma determinada visão a outras partes do mesmo nível ou de outros níveis; entradas *endógenas* ligam componentes individuais de uma parte em uma determinada visão.

AVISO

Se a capacidade de simulação não estiver disponível para um sistema reativo, o risco do projeto aumenta. Considere o uso de um modelo incremental de processo que lhe permita entregar um produto funcionando na primeira iteração e depois usar outras iterações para ajustar o seu desempenho.

6.2.2 Simulação do Sistema

Muitos sistemas baseados em computador interagem com o mundo real de um modo reativo. Isto é, eventos do mundo real são monitorados pelo hardware e pelo software que formam o sistema baseado em computador e, com base nesses eventos, o sistema impõe controle sobre as máquinas, os processos e até as pessoas que causam a ocorrência dos eventos. Sistemas em tempo real e embutidos freqüentemente se enquadram na categoria de sistemas reativos.

Muitos sistemas na categoria reativa controlam máquinas e/ou processos (por exemplo, aeronaves comerciais ou refinarias de petróleo) que precisam operar com um grau de confiabilidade extremamente alto. Se o sistema falhar, perdas humanas ou econômicas significativas podem ocorrer. Por essa razão, ferramentas de modelagem e simulação de sistemas são usadas para ajudar a eliminar surpresas quando sistemas reativos baseados em computador são construídos. Essas ferramentas são aplicadas durante o processo de engenharia de sistemas, enquanto os papéis do hardware e do software, dos banco de dados e do pessoal são especificados. As ferramentas de modelagem e simulação permitem ao engenheiro de sistemas agir como "piloto de testes" da especificação do sistema.

FERRAMENTAS DE SOFTWARE

Ferramentas de Simulação de Sistemas

Objetivo: As ferramentas de simulação de sistemas fornecem ao engenheiro de software a habilidade de prever o comportamento de um sistema de tempo real antes que ele seja construído. Além disso, essas ferramentas permitem que o engenheiro de software desenvolva bonecos do sistema de tempo real, permitindo ao cliente enxergar melhor as funções, operações e respostas antes da implementação real.

Mecânica: As ferramentas nessa categoria permitem a uma equipe definir os elementos de um sistema baseado em computador e, então, executar uma variedade de simulações para melhor entender as características operacionais e o desempenho global do sistema. Duas amplas categorias de ferramentas de simulação de sistemas existem: (1) ferramentas de propósito geral, que podem modelar virtualmente qualquer sistema baseado em computador e (2) ferramentas de propósitos especiais, que são projetadas para atender a um domínio de aplicação específico (por exemplo, sistemas aviônicos de aeronaves, sistemas de fabricação, sistemas eletrônicos).

Ferramentas Representativas[4]

CSIM, desenvolvido pela Lockheed Martin Advanced Technology Labs (www.atl.external.lmco.com), é um simulador de eventos discretos de propósito geral para sistemas orientados a diagramas de blocos.

Simics, desenvolvida pela Virtutech (www.virtutech.com), é uma plataforma de simulação de sistemas que pode modelar e analisar tanto hardwares quanto sistemas baseados em software.

SLX, desenvolvida pela Wolverine Software (www.wolverinesoftware.com), fornece blocos construtivos de propósito geral para modelar o desempenho de uma grande variedade de sistemas.

Um conjunto útil de *links* para uma ampla gama de recursos de sistemas de simulação pode ser encontrado em http://www.idsia.ch/~andrea/simtools.html.

6.3 ENGENHARIA DE PROCESSO DE NEGÓCIO: PANORAMA

O objetivo da *engenharia de processo de negócio — EPN (business process engineering, BPE)* é definir arquiteturas que permitam a um negócio usar informações de maneira efetiva. Quando se assume uma visão de mundo das necessidades de tecnologia da informação de uma empresa, não há dúvida de que a engenharia de sistemas é necessária. Não apenas a especificação da arquitetura computacional adequada é necessária, mas também a arquitetura do software que integra a configuração singular de recursos computacionais precisa ser desenvolvida. A engenharia de processo de negócio é uma abordagem para criar um plano global para a implementação da arquitetura computacional [SPE93].

4 As ferramentas mencionadas aqui não representam uma recomendação, mas, em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas de seus respectivos desenvolvedores.

? Quais arquiteturas são definidas e desenvolvidas como parte da EPN?

Três arquiteturas diferentes precisam ser analisadas e projetadas dentro do contexto dos objetivos e metas do negócio:

- Arquitetura de dados
- Arquitetura de aplicações
- Infra-estrutura tecnológica

A *arquitetura de dados* fornece a base, a estrutura, para as necessidades de informação de um negócio ou de uma função do negócio. Os blocos construtivos individuais da arquitetura são os objetos de dados usados pelo negócio. Um objeto de dados contém um conjunto de atributos que define certos aspectos, qualidades e características ou um identificador dos dados que estão sendo descritos.

Uma vez definido um conjunto de objetos de dados, suas relações são identificadas. Uma *relação* indica como os objetos são ligados uns aos outros. Por exemplo, considere os objetos: **cliente** e **produtoA**. Os dois objetos podem ser ligados pela relação *compra*; isto é, um **cliente** *compra* o **produtoA** ou o **produtoA** *é comprado por* um **cliente**. Os objetos de dados (pode haver centenas ou até milhares para uma atividade importante de negócio) fluem entre as funções do negócio, são organizados em um banco de dados e são transformados para fornecer a informação que serve às necessidades do negócio.

AVISO

Como engenheiro de softwares, você nunca deve se envolver em ISP ou BAA. No entanto, se ficar claro que essas atividades não foram feitas, informe aos interessados que o risco do projeto é muito alto.

A *arquitetura da aplicação* abrange os elementos de um sistema que transformam objetos da arquitetura de dados em alguma finalidade do negócio. No contexto deste livro, consideramos a arquitetura da aplicação como o sistema de programas (software) que realiza essa transformação. No entanto, em um contexto mais amplo, a arquitetura da aplicação pode incorporar o papel do pessoal (que são os transformadores e usuários da informação) e os procedimentos do negócio que não foram automatizados.

A *infra-estrutura tecnológica* provê os fundamentos para as arquiteturas de dados e de aplicação. A infra-estrutura abrange o hardware e o software usados para apoiar as aplicações e os dados. Isso inclui computadores, sistemas operacionais, redes, conexões de telecomunicação, tecnologias de armazenamento e a arquitetura (por exemplo, cliente/servidor) que foi projetada para implementar essas tecnologias.

Para modelar as arquiteturas de sistemas, uma hierarquia de atividades de engenharia de processos de negócio é definida e ilustrada na Figura 6.2.

6.4 ENGENHARIA DE PRODUTO: UM PANORAMA

AVISO

O modelo concorrente de processo (Capítulo 3) é freqüentemente usado nesse contexto. Cada disciplina de engenharia trabalha em paralelo. Certifique-se de que a comunicação seja encorajada à medida que cada disciplina realiza seu trabalho.

A meta da *engenharia de produto* é traduzir o desejo do cliente de um conjunto de capacidades definidas em um produto em funcionamento. Para alcançar essa meta, a engenharia de produto — como a engenharia de processo de negócio — deve derivar a arquitetura e a infra-estrutura. A arquitetura abrange quatro componentes de sistema distintos: software, hardware, dados (e bancos de dados) e pessoal. Uma infra-estrutura de suporte é estabelecida e inclui a tecnologia necessária para aglutinar os componentes e a informação (por exemplo, documentos, CD-ROM e vídeo) que é usada para apoiar os componentes.

Com referência à Figura 6.3, a visão de mundo é obtida por intermédio da engenharia de requisitos (Capítulo 7). Os requisitos globais do produto são extraídos do cliente. Esses requisitos abrangem necessidades de informação e de controle, função e comportamento do produto, desempenho global do produto, restrições de projeto e de interface, e outras necessidades especiais. Uma vez conhecidos os requisitos, o serviço da engenharia de requisitos é alocar função e comportamento a cada um dos quatro componentes mencionados anteriormente.

Document shared on www.docsity.com

FIGURA 6.2

A hierarquia da engenharia de processo de negócio (MAR90)

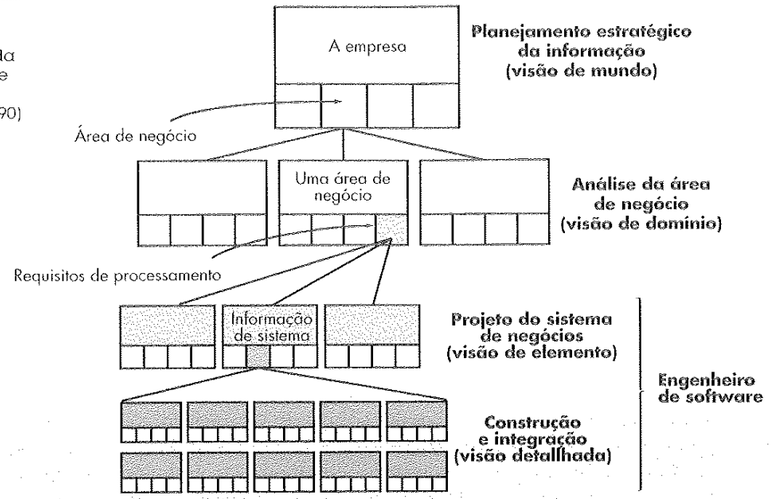
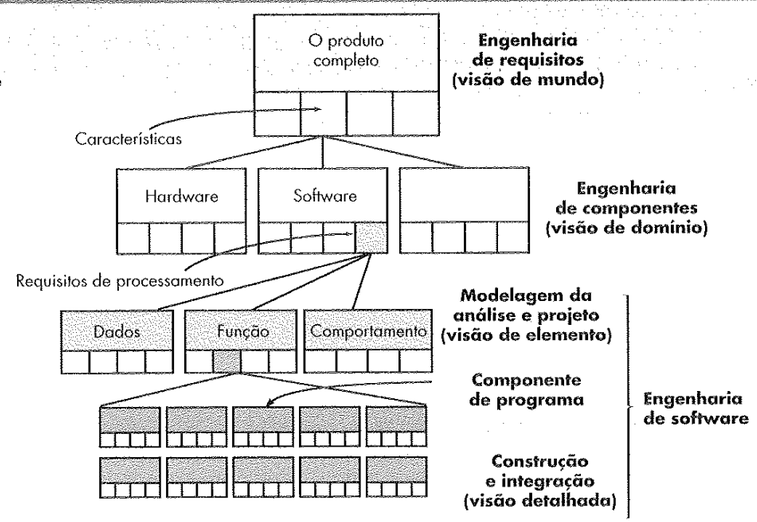


FIGURA 6.3

Hierarquia da engenharia de produto



Uma vez feita a alocação, começa a engenharia de componentes do sistema. A engenharia de componentes do sistema é, de fato, um conjunto de atividades concorrentes que trata separadamente de cada um dos componentes do sistema: engenharia de software, engenharia de hardware, engenharia humana e engenharia de banco de dados. Cada uma dessas disciplinas de engenharia adota uma visão específica de domínio, mas é importante notar que as disciplinas de engenharia devem estabelecer e manter comunicação ativa umas com as outras. Parte do papel da engenharia de requisitos é estabelecer os mecanismos de interface que permitem que isso aconteça.

A visão de elemento da engenharia de produto é a disciplina de engenharia propriamente dita aplicada ao componente alocado. Para a engenharia de software, isso significa atividades de modelagem de análise e de projeto (abordadas em detalhes nos capítulos posteriores) e atividades de

construção e implantação, que incluem as tarefas de geração de código, teste e suporte. A tarefa de análise modela os requisitos alocados em representações de dados, funções e comportamento. O projeto faz a correspondência do modelo de análise para os projetos nos níveis de dados, arquitetura, interface e componentes de software.

CASASEGURA

Engenharia de Sistema Preliminar

A cena: O espaço de trabalho da equipe de engenharia de software depois do encontro inicial ter ocorrido.

Os participantes: Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software; Ed Robbins, membro da equipe de software.

A conversa:

Ed: Eu penso que foi muito bem.

Vinod: Sim... mas tudo o que fizemos foi olhar o sistema global — temos bastante trabalho de levantamento de requisitos a fazer para o software.

Jamie: É por isso que temos reuniões adicionais programadas para os próximos cinco dias. Aliás, sugeri que dois dos "clientes" se deslocassem até aqui por algumas das próximas semanas. Você sabe, viver conosco de modo que possamos realmente nos comunicar, quer dizer, colaborar.

Vinod: Como foi isso?

Jamie: Bem, eles me olharam como se eu fosse louco, mas Doug [o gerente de engenharia de software] gosta da idéia — ela é agil — assim ele está falando com eles.

Ed: Fiz anotações usando meu PDA durante a reunião, e cheguei a uma lista de funções básicas.

Jamie: Excelente, vamos ver.

Ed: Já mandei uma cópia por e-mail a vocês dois. Dêem uma olhada e depois conversamos.

Vinod: Que tal depois do almoço?

[Jamie e Vinod receberam o seguinte de Ed]

Anotações preliminares sobre a estrutura/funcionalidade do *CasaSegura*:

- O sistema fará uso de um ou mais PCs, vários painéis montados nas paredes e/ou painéis de controle portáteis e vários sensores e controladores de eletrodomésticos/dispositivos.
- Todos se comunicarão via protocolos sem fio (por exemplo, 802.11b) e serão projetados para construção de casas novas e para aplicação em casas existentes.
- Todo hardware, com exceção de nossa nova caixa sem fio, será de prateleira.

Funcionalidades básicas do software que pude depreender da conversa inicial:

Funções de segurança da residência:

- Monitoração por sensor de movimento/janela/porta padrão para o acesso não autorizado (furtos).
- Monitoração de níveis de CO, fogo, fumaça.
- Monitoração de níveis de água em porões (por exemplo, inundação ou quebra de cano de água).
- Monitoração de movimento externo.
- Modificação da configuração de segurança via Internet.

Funções de vigilância da residência:

- Conectar a uma ou mais câmeras de vídeo colocadas dentro/fora da casa.
- Controlar afastamento/aproximação das câmeras.
- Definir zonas de monitoração das câmeras.
- Mostrar as visões das câmeras em PC.
- Ter acesso às visões da câmera via Internet.
- Gravar seletivamente a saída da câmera digitalmente.
- Rever a gravação de saída da câmera.

Funções de gestão da residência:

- Controlar iluminação.
- Controlar aparelhos eletrodomésticos.
- Controlar HVAC*.
- Controlar equipamentos de áudio/vídeo espalhados pela casa.
- Permitir a colocação da casa no "modo férias/viagem" com acionamento de um botão.
  - Ativar eletrodomésticos/iluminação/HVAC, respectivamente.
  - Colocar mensagem na secretária eletrônica.
  - Contatar vendedores para interromper correspondência, propaganda etc.

Funções de gestão de comunicação:

Funções da secretária eletrônica:

- Lista dos que ligaram por identificador de chamadas.

* N. de R.T.: HVAC é uma sigla norte-americana que se refere a aparelhos de aquecimento, refrigeração e ar condicionado.

Document shared on www.docsity.com

- Mensagens com data e hora anotadas.
- Texto da mensagem por meio do sistema de reconhecimento de voz.
- Funções de e-mail (todas as funções padrão de e-mail)
  - Mostrador padrão de e-mail.
  - Leitura por voz de e-mails via acesso telefônico.
- Catálogo de telefones pessoal.
- Link para PDA

Outras funções:

Ainda indefinidas.

Todas as funções são acessíveis via Internet com a proteção de senha adequada.

6.5 Modelagem de Sistemas

Como um sistema pode ser representado em diferentes níveis de abstração (por exemplo, a visão de mundo, a visão de domínio, a visão de elemento), os *modelos de sistemas* tendem a ser de natureza hierárquica ou em camadas. No topo da hierarquia, um modelo completo do sistema é apresentado (a visão de mundo). Os principais objetos de dados, funções de processamento e comportamentos são representados sem considerar os componentes do sistema que implementarão os elementos do modelo da visão de mundo. À medida que a hierarquia é refinada ou posta em camadas, detalhes dos componentes (nesse caso, representações de hardware, software etc.) são modelados. Finalmente, os modelos de sistema evoluem para modelos de engenharia (que são ainda mais refinados) específicos da disciplina de engenharia adequada.

6.5.1 Modelagem Hatley-Pirbhai

Todo sistema baseado em computador pode ser modelado como uma transformação de informação usando um gabarito entrada–processamento–saída. Hatley e Pirbhai [HAT87] estenderam essa visão para incluir duas características adicionais do sistema — processamento e manutenção da interface do usuário, e autoteste. Apesar de essas características adicionais não estarem presentes em todo sistema baseado em computador, elas são muito comuns e sua especificação torna mais robusto qualquer modelo de sistema.

PONTO CHAVE

O modelo de Hatley-Pirbhai mostra entrada, processamento e saída juntamente com a interface do usuário e manutenção/autoteste.

Usando uma representação de entrada, processamento, saída, processamento da interface do usuário e processamento de autoteste, um engenheiro de sistemas pode criar um modelo de componentes do sistema que estabeleça a fundação para os passos posteriores em cada uma das disciplinas de engenharia.

Para desenvolver o modelo de sistema, é usado um gabarito de modelos de sistema [HAT87]. O engenheiro de sistemas distribui os elementos do sistema em cada uma das cinco regiões de processamento dentro do gabarito: (1) interface do usuário, (2) entrada, (3) função e controle do sistema, (4) saída e (5) manutenção e autoteste.

Como praticamente todas as técnicas de modelagem usadas na engenharia de sistemas e de software, o gabarito de modelo do sistema permite ao analista criar uma hierarquia de detalhes. Um *diagrama de contexto do sistema* (*system context diagram* — SCD) fica no nível mais alto da hierarquia. O diagrama de contexto "estabelece o limite de informação entre o sistema que está sendo implementado e o ambiente no qual o sistema deve operar" [HAT87]. Isto é, o SCD define todos os produtores externos da informação usada pelo sistema, todos os consumidores externos da informação criada pelo sistema e todas as entidades que se comunicam através da interface ou realizam manutenção e autoteste.

Para ilustrar o uso do SCD, considere um sistema de classificação CLSS (conveyor line sorting system — por esteira rolante), descrito com a seguinte declaração de objetivos (um tanto nebulosa) para o CLSS:

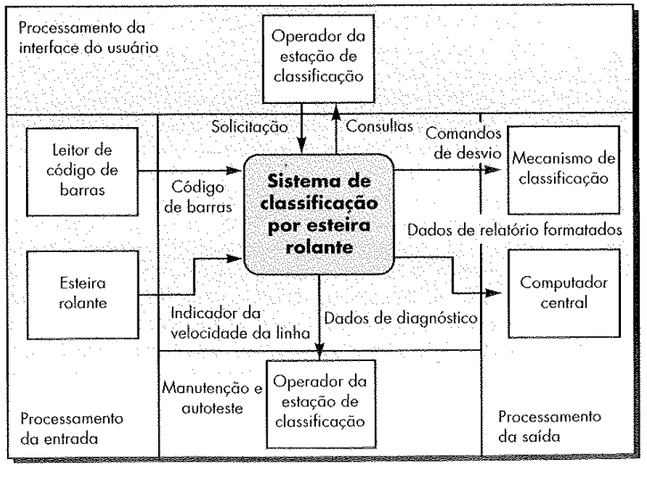
> O CLSS deve ser desenvolvido de modo que caixas que passam ao longo de uma esteira rolante sejam identificadas e classificadas em um dos seis escaninhos que ficam no fim da linha. As caixas passarão por uma estação de classificação na qual serão identificadas. Com base em um número de identificação impresso na lateral da caixa e em um código de barras, as caixas serão desviadas para o escaninho adequado. As caixas passam em ordem aleatória e são igualmente espaçadas. A linha se move lentamente.
>
> Um computador pessoal localizado na estação de classificação executa todo o software do CLSS, interage com o leitor de código de barras para ler os números de peça em cada caixa, interage com o equipamento de monitoração da esteira rolante para obter a velocidade da esteira, guarda todos os números de peça ordenados, interage com o operador da estação de classificação para produzir uma variedade de relatórios e diagnósticos, envia sinais de controle para o hardware de desvio a fim de classificar as caixas, e se comunica com o sistema central de automação da fábrica.

O SCD para o CLSS é mostrado na Figura 6.4. O diagrama é dividido em cinco partes principais. A parte superior representa o processamento da interface do usuário e as partes à direita e à esquerda referem-se ao processamento de entrada e de saída, respectivamente. A parte central contém as funções de controle e de processo, e a parte inferior enfoca a manutenção e o autoteste. Cada retângulo mostrado na figura representa uma *entidade externa* — isto é, um produtor ou um consumidor de informação do sistema. Por exemplo, o leitor de código de barras produz informações que entram no sistema CLSS. O símbolo para todo o sistema (ou, em níveis mais baixos, para os principais subsistemas) é um retângulo de cantos arredondados. Assim, o CLSS é representado na região de processamento e controle ao centro do SCD. As setas rotuladas mostradas no SCD representam a informação (dados e controle) à medida que ela se move do ambiente externo para dentro do sistema CLSS. A entidade externa leitor de código de barras produz informações de entrada que são rotuladas como código de barras. Essencialmente, o SCD coloca qualquer sistema no contexto do seu ambiente externo.

O engenheiro de sistemas refina o diagrama de contexto do sistema considerando o retângulo sombreado da Figura 6.4 em mais detalhes. São identificados os principais subsistemas que permitem ao sistema de classificação por esteira rolante funcionar no contexto definido pelo SCD. Os principais subsistemas são definidos em um *diagrama de fluxo do sistema* (*system flow diagram* — SFD) que é derivado do SCD. O fluxo de informação pelas regiões do SCD é usado para guiar o engenheiro de sistemas no desenvolvimento do SFD — um esquema "mais detalhado" do CLSS. O diagrama de fluxo do sistema mostra os principais subsistemas e as linhas importantes de fluxo de informação (dados e controle). Além disso, o gabarito do sistema particiona o processamento dos subsistemas em cada uma das cinco regiões discutidas anteriormente. Nesse estágio, cada um dos subsistemas pode conter um ou mais elementos do sistema (por exemplo, hardware, software e pessoal), que foram distribuídos pelo engenheiro de sistemas.

FIGURA 6.4

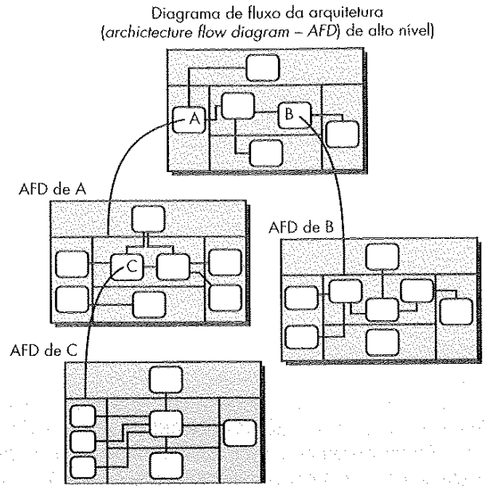
Diagrama de contexto do sistema para o CLSS



Document shared on www.docsity.com

FIGURA 6.5

Construção de uma hierarquia de SFD



O diagrama de fluxo inicial do sistema transforma-se no nó superior de uma hierarquia de SFDs. Cada retângulo de cantos arredondados no SFD original pode ser expandido em outro gabarito de arquitetura dedicado somente a ele. Esse processo é ilustrado esquematicamente na Figura 6.5. Cada um dos SFDs do sistema pode ser usado como ponto de partida para etapas de engenharia subseqüentes no subsistema que foi descrito.

Os subsistemas e a informação que flui entre eles podem ser especificados (limitados) para o trabalho de engenharia subseqüente. Uma descrição narrativa de cada subsistema e uma definição de todos os dados que fluem entre eles tornam-se elementos importantes da Especificação do Sistema.

6.5.2 Modelagem de Sistema com a UML

A UML fornece uma gama de diagramas que podem ser usados para análise e projeto tanto em nível de sistema quanto de software[5]. Para o sistema CLSS são modelados quatro importantes elementos de sistema: (1) o hardware que habilita o CLSS; (2) o software que implementa o acesso e a ordenação do banco de dados; (3) o operador que submete vários requisitos ao sistema; e (4) o banco de dados que contém as informações relevantes de código de barras e de destino.

O hardware do CLSS pode ser modelado em nível de sistema usando um *diagrama de implantação* UML, como ilustrado na Figura 6.6. Cada caixa 3-D refere-se a um elemento de hardware que faz parte da arquitetura física do sistema. Em alguns casos, os elementos de hardware terão de ser projetados e construídos como parte do projeto. Em muitos casos, entretanto, os elementos de hardware podem ser adquiridos de prateleira. O desafio para a equipe de engenharia é interfacear adequadamente os elementos de hardware.

Veja na Web

Uma especificação completa da sintaxe e da semântica da UML (a ser discutida nos capítulos posteriores) pode ser encontrada em www.rational.com/uml/index.jsp.

Os elementos de software para CLSS podem ser mostrados de vários modos usando a UML. Aspectos procedurais do software CLSS podem ser representados usando um *diagrama de atividades* (Figura 6.7). Essa notação UML é similar à de um fluxograma e é usada para representar o que acontece quando o sistema executa suas funções. Retângulos de cantos arredondados definem uma função específica do sistema; setas definem o fluxo através do sistema; o losango de decisão representa uma decisão de desvio (cada seta partindo do losango é rotulada); linhas sólidas horizontais definem atividades paralelas que estão ocorrendo.

Outra notação UML que pode ser usada para modelar softwares é o *diagrama de classe* (assim como os muitos diagramas relacionados a classes discutidos mais adiante neste livro). No nível

5 Uma discussão mais detalhada sobre diagramas UML é apresentada do Capítulo 8 ao 11. Para uma discussão abrangente sobre UML, o leitor interessado deverá ver [SCH02], [LAR01] ou [BEN99].

FIGURA 6.6

Diagrama de implantação para o hardware CLSS

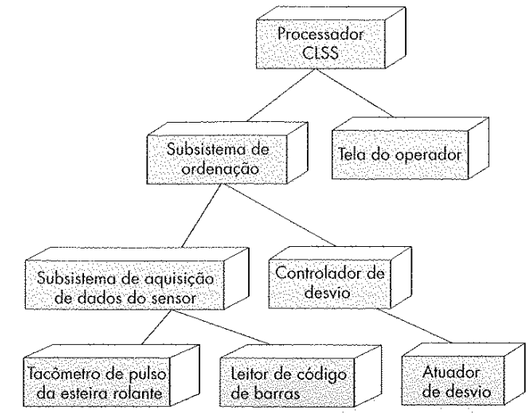
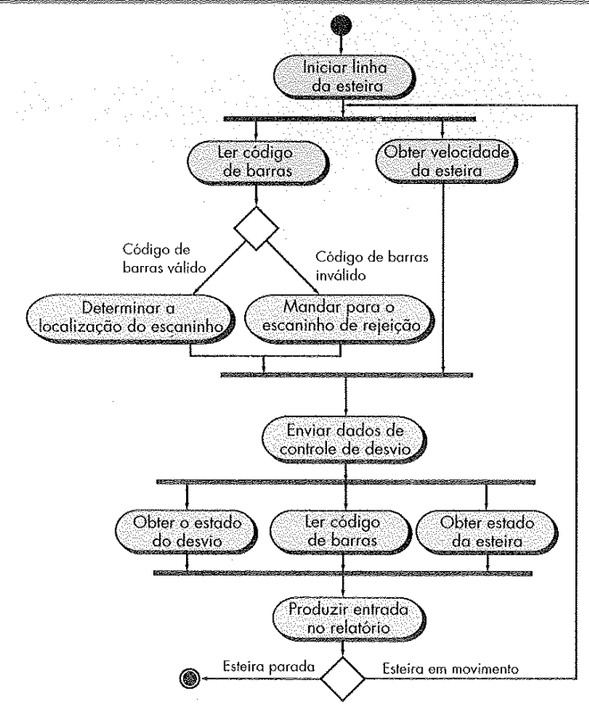


FIGURA 6.7

Diagrama de atividades para CLSS



de engenharia de sistemas, as classes[6] são extraídas do enunciado do problema. Para o CLSS, as classes candidatas poderiam ser: **Caixa, TrilhoDaEsteira, LeitorDeCodigoDeBarras, Contro-**

6 Nos capítulos anteriores, observamos que uma classe representa um conjunto de entidades que faz parte do domínio do sistema. Essas entidades podem ser transformadas ou armazenadas pelo sistema ou podem servir como um produtor ou consumidor da informação produzida pelo sistema.

Document shared on www.docsity.com

ladorDeDesvio, RequisicaoDeOperador, Relatorio, Produto e outras. Cada classe encapsula um conjunto de atributos que mostra todas as informações necessárias sobre a classe. Uma descrição de classe também contém um conjunto de operações que são aplicadas à classe no contexto do sistema CLSS. Um diagrama de classe UML para **Caixa** pode ser visto na Figura 6.8.

O operador CLSS pode ser modelado com um diagrama de caso de uso UML como mostrado na Figura 6.9. O diagrama de caso de uso ilustra a maneira como um ator (nesse caso, o operador, representado pela figura do boneco) interage com o sistema. Cada elipse dentro da caixa (que representa o limite do sistema CLSS) representa um caso de uso — um cenário em texto que descreve uma interação com o sistema.

FIGURA 6.8

Diagrama de classe UML para a classe Caixa

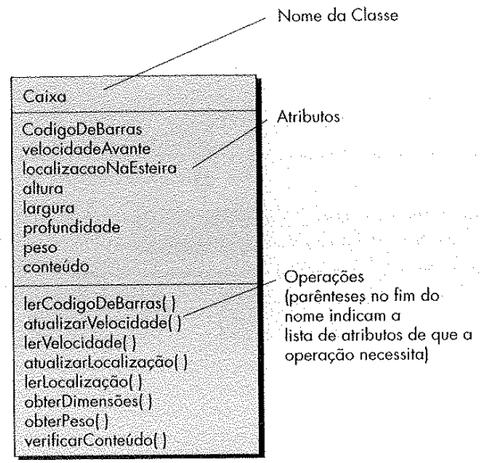
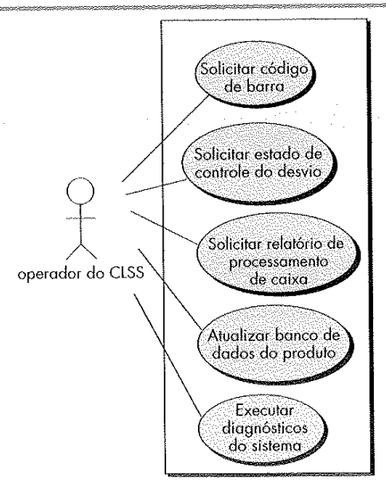


FIGURA 6.9

Diagrama de caso de uso para o operador CLSS



FERRAMENTAS DE SOFTWARE

Ferramentas de Modelagem de Sistema

Objetivo: As ferramentas de modelagem de sistema fornecem ao engenheiro de software a habilidade de modelar todos os elementos de um sistema baseado em computador usando uma notação que é específica para aquela ferramenta.

Mecânica: A mecânica da ferramenta varia. Em geral, as ferramentas nessa categoria possibilitam a um engenheiro de software modelar (1) a estrutura de todos os elementos funcionais do sistema; (2) o comportamento estático e dinâmico do sistema; e (3) a interface homem-máquina.

Ferramentas Representativas[7]

Describe, desenvolvido pela Embarcadero Technologies (www.embarcadero.com), é um conjunto de ferramentas de modelagem com base na UML que pode representar softwares ou sistemas completos.

Rational XDE e Rose, desenvolvidos pela Rational Technologies (www.rational.com), fornecem um conjunto de ferramentas de modelagem e desenvolvimento com base na UML amplamente usado para sistemas baseados em computador.

Real-Time Studio, desenvolvido pela Artisan Software (www.artisansw.com), é um conjunto de ferramentas de modelagem e desenvolvimento que apóia o desenvolvimento de sistemas em tempo real.

Telelogic Tau, desenvolvido pela Telelogic (www.telelogic.com), é um conjunto de ferramentas com base na UML que apóia a modelagem de análise e de projeto, bem como vínculos para características de construção de software.

6.6 RESUMO

Um sistema de alta tecnologia inclui vários elementos: software, hardware, pessoal, banco de dados, documentação e procedimentos. A engenharia de sistemas ajuda a traduzir as necessidades do cliente em um modelo de sistema que faz uso de um ou mais desses elementos.

A engenharia de sistemas começa adotando uma "visão de mundo". Um domínio de negócios ou de produto é analisado a fim de estabelecer todos os requisitos básicos do negócio. O foco é então estreitado para uma "visão de domínio", em que cada um dos elementos do sistema é analisado individualmente. Cada elemento é alocado em um ou mais componentes de engenharia, que são, então, desenvolvidos pela disciplina de engenharia correspondente.

A engenharia de processo de negócio é uma abordagem da engenharia de sistemas usada para definir arquiteturas que permitam ao negócio usar efetivamente as informações. O objetivo da engenharia de processo de negócio é derivar uma arquitetura de dados, uma arquitetura da aplicação e uma infra-estrutura tecnológica abrangentes, que satisfarão às necessidades da estratégia do negócio e aos objetivos e metas de cada área do negócio.

A engenharia de produto é uma abordagem da engenharia de sistemas que começa com a análise do sistema. O engenheiro de sistemas identifica as necessidades do cliente, determina a viabilidade econômica e técnica e atribui funções e desempenho ao software, hardware, pessoal e bancos de dados — os componentes-chave de engenharia.

REFERÊNCIAS BIBLIOGRÁFICAS

[BEN99]Bennett, S., McRobb, S. e Farmer, R., *Object-Oriented Systems Analysis and Design Using UML*, McGraw-Hill, 1999.

[HAR93] Hares, J. S., *Information Engineering for the Advanced Practitioner*, Wiley, 1993, p. 12-13.

[HAT87] Hatley, D. J. e Pirbhai, I. A., *Strategies for Real-Time System Specification*, Dorset House, 1987.

7 As ferramentas mencionadas aqui não representam uma recomendação, mas, em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos os nomes das ferramentas são marcas registradas de seus respectivos desenvolvedores.

Document shared on www.docsity.com

[LAR01] Larman, C., *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, 2ª ed., Prentice-Hall, 2001.

[MAR90] Martin, J., *Information Engineering: Book II — Planning and Analysis*, Prentice-Hall, 1990.

[MOT92] Motamarri, S., "Systems Modeling and Description", *Software Engineering Notes*, vol. 17, n. 2, abr. 1992, p. 57-63.

[SCH02] Schmuller, J., *Teach Yourself UML in 24 Hours*, 2ª ed., Sams Publishing, 2002.

[SPE93] Spewak, S., *Enterprise Architecture Planning*, QED Publishing, 1993.

[THA97] Thayer, R. H. e Dorfman, M., *Software Requirements Engineering*, 2ª ed., IEEE Computer Society Press, 1997.

Problemas e Pontos a Considerar

6.1. Encontre para a palavra *sistema* todos os sinônimos de uma única palavra que você puder. Boa sorte!

6.2. Construa um "sistema de sistemas" hierárquico para um sistema, produto ou serviço com o qual você esteja familiarizado. Sua hierarquia deve descer até os elementos simples do sistema (hardware, software etc.) ao longo de pelo menos um ramo da "árvore".

6.3. Selecione um sistema ou produto de grande porte com o qual você esteja familiarizado. Defina o conjunto de domínios que descreve a visão de mundo do sistema ou do produto. Descreva o conjunto de elementos que constituem um ou dois domínios. Para um elemento, identifique os componentes técnicos que devem passar por engenharia.

6.4. Selecione um sistema ou produto de grande porte com o qual você esteja familiarizado. Declare as suposições, simplificações, limitações, restrições e preferências que teriam de ser feitas para construir um modelo de sistema efetivo (e realizável).

6.5. A engenharia de processo de negócio busca definir tanto *dados* e *arquitetura de dados* quanto *infra-estrutura tecnológica*. Descreva o que cada um desses termos significa e forneça um exemplo.

6.6. Um engenheiro de sistemas pode vir de uma de três fontes: desenvolvedor do sistema, cliente ou alguma organização externa. Discuta os prós e os contras que se aplicam a cada fonte. Descreva um engenheiro de sistemas "ideal".

6.7. Seu instrutor distribuirá uma descrição de alto nível de um sistema ou produto baseado em computador:

a. Desenvolva um conjunto de questões que você formularia como engenheiro de sistemas.

b. Proponha pelo menos duas alocações diferentes para o sistema com base nas respostas às suas questões.

c. Na sala de aula, compare a sua alocação com a de seus colegas.

6.8. Desenvolva um diagrama de contexto para um sistema baseado em computador de sua escolha (ou um designado pelo seu instrutor).

6.9. Embora a informação neste ponto seja muito incompleta, tente desenvolver um diagrama de implantação, um diagrama de atividade, um diagrama de classe e um diagrama de caso de uso UML para o produto *CasaSegura*.

6.10. Pesquise a literatura e escreva uma monografia curta descrevendo como as ferramentas de modelagem e simulação funcionam. Alternativa: reúna literatura de dois ou mais fornecedores de ferramentas de modelagem e simulação e avalie suas semelhanças e diferenças.

6.11. Há características de um sistema que não podem ser estabelecidas durante as atividades de engenharia de sistemas? Descreva as características, se existirem, e explique por que a sua consideração precisa ser adiada até passos posteriores da engenharia.

6.12. Há situações nas quais a especificação formal do sistema pode ser resumida ou inteiramente eliminada? Explique.

Leituras e Fontes de Informação Adicionais

Os livros de Hatley e seus colegas (*Process for Systems Architecture and Requirements Engineering*, Dorset House, 2000), Buede (*The Engineering Design of Systems: Models and Methods*, Wiley, 1999), Weiss e seus colegas (*Software Product-Line Engineering*, Addison-Wesley, 1999), Blanchard e Fabrycky (*System Engineering and Analysis*, terceira edição, Prentice-Hall, 1998), Armstrong e Sage (*Introduction to Systems Engineering*, Wiley, 1997) e Martin (*Systems Engineering Guidebook*, CRC Press, 1996) apresentam o processo de engenharia de sistemas (com ênfases específicas de engenharia) e fornecem diretrizes valiosas. Blanchard (*System Engineering Management*, segunda edição, Wiley, 1997) e Lacy (*System Engineering Management*, McGraw-Hill, 1992) discutem tópicos de gestão de engenharia de sistemas.

Chorafas (*Enterprise Architecture and New Generation System*, St. Lucie Press, 2001) apresenta arquiteturas de sistemas e de engenharia de informação para a "próxima geração" de soluções de TI incluindo sistemas baseados na Internet. Wallnau e seus colegas (*Building Systems from Commercial Components*, Addison-Wesley, 2001) apresentam tópicos de engenharia de sistemas baseada em componentes para sistemas de informação e produtos. Lozinsky (*Enterprise-Wide Software Solutions: Integration Strategies and Practices*, Addison-Wesley, 1998) apresenta o uso de pacotes de software como uma solução que permite a uma empresa migrar de sistemas legados para processos modernos de negócio. Uma discussão valiosa sobre risco e engenharia de sistemas é apresentada por Bradley (*Elimination of Risk in Systems*, Tharsis Books, 2002).

Davis (*Business Process Modeling with Aris: A Practical Guide*, Springer-Verlag, 2001), Bustard e seus colegas (*System Models for Business Process Improvement*, Artech House, 2000), e Scheer (*Business Process Engineering: Reference Models for Industrial Enterprises*, Springer-Verlag, 1998) descrevem métodos de modelagem de processos de negócio para sistemas de informação que atendem toda a empresa.

Davis e Yen (*The Information System Consultant´s Handbook: Systems Analysis and Design*, CRC Press, 1998) apresentam cobertura enciclopédica de tópicos de análise e projeto de sistemas no domínio de sistemas de informação. Um excelente tutorial do IEEE por Thayer e Dorfman [THA97] discute o interrelacionamento entre tópicos de análise de requisitos no nível de sistemas e de software.

Law e seus colegas (*Simulation Modeling and Analysis*, McGraw-Hill, 1999) discutem técnicas de modelagem e simulação de sistemas para uma grande variedade de domínios de aplicação.

Para aqueles leitores ativamente envolvidos em trabalhos de sistemas ou interessados em um tratamento mais sofisticado desse tópico, os livros de Gerald Weinberg (*An Introduction to General System Thinking*, Wiley-Interscience, 1976 e *On the Design of Stable Systems*, Wiley-Interscience, 1979) tornaram-se clássicos e fornecem uma excelente discussão sobre o "pensamento geral de sistemas" que implicitamente leva a uma abordagem geral da análise e do projeto de sistemas. Livros mais recentes de Weinberg (*General Principles of Systems Design*, Dorset House, 1988 e *Rethinking Systems Analysis and Design*, Dorset House, 1988) continuam a tradição de seu trabalho inicial.

Uma grande variedade de fontes de informação sobre engenharia de sistemas e assuntos relacionados está disponível na Internet. Uma lista atualizada de referências relevantes da World Wide Web sobre engenharia de sistemas, engenharia de informação, engenharia de processo de negócio e engenharia de produtos pode ser encontrada no site: **http://www.mhhe.com/pressman.**

Document shared on www.docsity.com

CAPÍTULO 7

ENGENHARIA DE REQUISITOS

CONCEITOS-CHAVE

modelo de análise
construção 134
elaboração 119
elementos 134
levantamento 118
QFD (QFD) 127
miniespecs 126
negociação 119
padrões de análise 137
casos de uso 129
especificação 120
gestão requisitos 121
rastreamento 121
validação 120

Entender os requisitos de um problema está entre as tarefas mais difíceis enfrentadas por um engenheiro de software. Quando você começa a pensar sobre isso, a engenharia de requisitos não parece tão difícil. Afinal de contas, o cliente não sabe o que é necessário? Os usuários finais não deveriam ter um bom entendimento das características e funções que vão oferecer benefícios? Surpreendentemente, em muitos casos, a resposta a essas perguntas é não. E mesmo que clientes e usuários finais sejam explícitos quanto às suas necessidades, essas vão se modificar ao longo do projeto. A engenharia de requisitos é difícil.

No prefácio de um livro de Ralph Young [YOU01] sobre práticas efetivas de requisitos, eu escrevi:

> É seu pior pesadelo. Um cliente entra no seu escritório, senta-se, olha você direto nos olhos e diz: "Eu sei que você pensa que entende o que eu disse, mas o que você não entende é que, o que eu disse, não é o que eu queria dizer". Invariavelmente, isso acontece no final de um projeto, depois que os compromissos de prazo de entrega foram feitos, que as reputações estão envolvidas e que dinheiro sério está em jogo.
>
> Todos nós que temos trabalhado no negócio de sistemas e software há alguns anos, vivemos esse pesadelo e, no entanto, poucos de nós aprenderam a se livrar dele. Lutamos quando tentamos levantar requisitos de nossos clientes. Temos dificuldade de entender a informação que conseguimos. Freqüentemente registramos os requisitos de maneira desorganizada e gastamos muito pouco tempo verificando o que de fato registramos. Permitimos que as modificações nos controlem, em vez de estabelecer mecanismos para controlar as modificações. Em resumo, falhamos em estabelecer uma fundamentação sólida para o sistema ou software. Cada um desses problemas é um desafio. Quando eles se combinam, a perspectiva é danosa mesmo para os gerentes e profissionais mais experientes. Mas soluções de fato existem.

PANORAMA

O que é? A engenharia de requisitos ajuda os engenheiros de software a compreender melhor o problema que eles vão trabalhar para resolver. Ela inclui o conjunto de tarefas que levam a um entendimento de qual será o impacto do software sobre o negócio, do que o cliente quer e de como os usuários finais vão interagir com o software.

Quem faz? *Engenheiros de software* (algumas vezes conhecidos como *engenheiros de sistemas* ou *analistas* no mundo da TI) e outros interessados (gerentes, clientes e usuários finais), todos participam da engenharia de requisitos.

Por que é importante? Projetar e construir um programa de computador elegante que resolva o problema errado não serve às necessidades de ninguém. Essa é a razão por que é importante entender o que o cliente deseja antes de começar a projetar e construir um sistema baseado em computador.

Quais são os passos? A engenharia de requisitos começa com a concepção — tarefa que define o escopo e a natureza do problema a ser resolvido. Ela avança para o levantamento — tarefa que ajuda o cliente a definir o que é necessário; e depois para a elaboração — em que os requisitos básicos são refinados e modificados. À medida que o cliente define o problema, ocorre a negociação — quais são as prioridades, o que é essencial, o que é necessário? Finalmente, o problema é especificado de algum modo e depois revisado ou validado de modo a garantir que o seu entendimento e o entendimento do cliente sobre o problema coincidam.

Qual é o produto do trabalho? O objetivo do processo de engenharia de requisitos é fornecer a todas as partes um entendimento escrito do problema. Isso pode ser conseguido por meio de um certo número de produtos de trabalho: cenários de usuário, listas de funções e de características, modelos de análise ou uma especificação.

Como tenho certeza de que fiz corretamente? Os produtos de trabalho da engenharia de requisitos são revisados com o cliente e com o usuário final para garantir que o que você compreendeu é o que eles realmente queriam dizer. Um pequeno alerta: mesmo depois de todas as partes concordarem, as coisas vão mudar e elas vão continuar mudando ao longo de todo o projeto.

Eu seria desonesto se chamasse a engenharia de requisitos de "solução" para os desafios anteriormente mencionados. Mas ela nos fornece uma sólida abordagem para enfrentar esses desafios.

7.1 UMA PONTE PARA O PROJETO E A CONSTRUÇÃO

Projetar e construir softwares de computador é desafiador, criativo e gostoso. De fato, construir softwares é tão compulsivo que muitos desenvolvedores de software desejam começar logo, antes de terem um entendimento claro do que é necessário. Eles alegam que as coisas vão ficar mais claras à medida que eles constroem; que os interessados no projeto vão poder entender melhor as necessidades apenas após examinar as primeiras iterações do software; que as coisas se modificam tão rapidamente que a engenharia de requisitos é uma perda de tempo; que o limite inferior é produzir um programa funcionando e que tudo mais é secundário. O que torna esses argumentos sedutores é que eles contêm elementos de verdade.[1] No entanto, cada um deles está errado e todos podem levar a um projeto falho de software.

> "A parte individual mais difícil da construção de um sistema de software é decidir o que construir. Nenhuma parte do trabalho danifica tanto o sistema resultante se for feita errado. Nenhuma outra parte é mais difícil de consertar depois."
>
> **Fred Brooks**

A engenharia de requisitos, como todas as outras atividades de engenharia de software, precisa ser adaptada às necessidades do processo, do projeto, do produto e do pessoal que está fazendo o trabalho. Na perspectiva do processo de software, a engenharia de requisitos (ER) é uma ação de engenharia de software que começa durante a atividade de comunicação e continua durante atividade de modelagem.

Em alguns casos, uma abordagem abreviada pode ser escolhida. Em outros, cada tarefa definida para a engenharia de requisitos abrangente precisa ser realizada rigorosamente. De um modo geral, a equipe de software precisa adaptar sua abordagem à ER. Mas adaptação não significa abandono. É essencial que a equipe de software faça um esforço real para entender os requisitos de um problema *antes* de tentar resolvê-lo.

PONTO CHAVE

A engenharia de requisitos estabelece uma base sólida para o projeto e a construção. Sem ela, o software resultante tem uma alta probabilidade de não satisfazer às necessidades dos clientes.

A engenharia de requisitos constrói uma ponte para o projeto e a construção. Entretanto, onde a ponte tem início? Alguém pode alegar que ela começa nos pés dos interessados no projeto (por exemplo, gerentes, clientes, usuários finais), onde a necessidade do negócio é definida, os cenários de uso são descritos, as funções e características são delineadas e as restrições de projeto são identificadas. Outros poderiam sugerir que ela começa com uma ampla definição do sistema em que o software é apenas um componente (Capítulo 6) do domínio mais amplo do sistema. Mas, independentemente do ponto inicial, a passagem pela ponte nos leva muito acima do projeto, permitindo à equipe de software examinar o contexto do trabalho de software a ser realizado; as necessidades específicas que o projeto e a construção precisam satisfazer; as prioridades que estabelecem a ordem na qual o trabalho deve ser completado e a informação, funções e comportamentos que terão um impacto profundo no projeto resultante.

1 Isso é particularmente verdadeiro para pequenos projetos (de menos que um mês) e esforços de software menores relativamente simples. À medida que o software cresce em tamanho e complexidade, esses argumentos começam a se desfazer.

Document shared on www.docsity.com

7.2 Tarefas da Engenharia de Requisitos

AVISO

Espere fazer um pouco de projeto durante o trabalho de requisitos e um pouco de trabalho de requisitos durante o projeto.

A engenharia de requisitos fornece o mecanismo apropriado para entender o que o cliente deseja, analisando as necessidades, avaliando a exeqüibilidade, negociando uma condição razoável, especificando a solução de modo não ambíguo, validando a especificação e gerindo os requisitos à medida que eles são transformados em um sistema operacional [THA97]. O processo de engenharia de requisitos é realizado por meio da execução de sete funções distintas: *concepção, levantamento, elaboração, negociação, especificação, validação* e *gestão*.

É importante notar que algumas dessas funções de engenharia de requisitos ocorrem em paralelo e que todas são adaptadas às necessidades do projeto. Todas tentam definir o que o cliente deseja e servem para estabelecer uma fundação sólida para o projeto e a construção do que o cliente obtém.

7.2.1 Concepção

Como um projeto de software é iniciado? Existe um evento único que se torna catalisador de um novo sistema ou produto baseado em computador, ou a necessidade evolui com o tempo? Não há respostas definitivas para essas questões.

> "As sementes dos principais desastres de software são usualmente lançadas nos primeiros três meses de início do projeto de software."
>
> **Capers Jones**

Em alguns casos, uma conversa casual é tudo o que é necessário para precipitar um esforço importante de engenharia de software. Mas, em geral, a maioria dos projetos começa quando uma necessidade de negócio é identificada ou um mercado ou serviço potencialmente novo é descoberto. Interessados da comunidade de negócios (por exemplo, gerentes do negócio, pessoal de marketing, gerentes de produto) definem um caso de negócio para a idéia, tentam identificar a abrangência e a profundidade do mercado, fazem uma análise de viabilidade grosseira e identificam uma descrição que funciona do escopo do projeto. Todas essas informações estão sujeitas a modificações (uma ocorrência provável), mas são suficientes para precipitar discussões com a organização da engenharia de software.[2]

Na *concepção*[3] do projeto, os engenheiros de software perguntam uma série de questões livres de contexto, conforme será discutido na Seção 7.3.4. A intenção é estabelecer um entendimento básico do problema, o pessoal que quer uma solução, a natureza da solução desejada e a efetividade da comunicação e colaboração preliminares entre cliente e desenvolvedor.

7.2.2 Levantamento

Certamente parece muito simples — pergunte ao cliente, aos usuários e aos outros quais são os objetivos do sistema ou do produto, o que precisa ser conseguido, como o sistema ou o produto se encaixa nas necessidades do negócio e, finalmente, como o sistema ou o produto será usado no dia-a-dia. Mas não é simples — é muito difícil.

Christel e Kang [CRI92] identificam vários problemas que nos ajudam a compreender por que o *levantamento* de requisitos é difícil:

- **Problemas de escopo.** O limite do sistema é mal definido ou o cliente/usuário especifica detalhes técnicos desnecessários que podem confundir, em vez de esclarecer, os objetivos globais do sistema.
- **Problemas de entendimento.** Os clientes/usuários não estão completamente certos do que é necessário, têm pouca compreensão das capacidades e limitações de seu ambiente computacional, não têm pleno entendimento do domínio do problema, têm dificuldade de informar as necessidades ao engenheiro de sistemas, omitem informações que acreditam ser "óbvias", especificam requisitos conflitantes com as necessidades de outros clientes/usuários ou especificam requisitos que são ambíguos ou impossíveis de testar.
- **Problemas de volatilidade.** Os requisitos mudam ao longo do tempo.

2 Se um sistema baseado em computador deve ser desenvolvido, as discussões começam com a engenharia de sistemas, atividade que define a visão de mundo e a visão de domínio (Capítulo 6) do sistema.

3 Leitores do Capítulo 3 vão recordar que o Processo Unificado define uma "fase de concepção" mais abrangente que inclui as tarefas de concepção, levantamento e elaboração discutidas neste capítulo.

? Por que é tão difícil obter um entendimento claro do que o cliente deseja?

Para ajudar a contornar esses problemas, os engenheiros de sistemas devem abordar a atividade de coleta dos requisitos de um modo organizado.

AVISO

A elaboração é uma boa coisa, mas você tem de saber quando parar. A chave é descrever o problema de modo que uma base firme para o projeto seja estabelecida. Se você trabalhar além desse ponto, estará fazendo o projeto.

7.2.3 Elaboração

As informações obtidas do cliente durante a concepção e o levantamento são expandidas e refinadas durante a *elaboração*. Essa atividade da engenharia de requisitos enfoca o desenvolvimento de um modelo técnico refinado das funções, características e restrições do software.

A elaboração é uma ação de modelagem de análise (Capítulo 8) composta de várias tarefas de modelagem e refinamento. Ela é guiada pela criação e refinamento de cenários do usuário que descrevem como o usuário final (e outros atores) vão interagir com o sistema. Cada cenário é escrutinado para extrair as classes de análise — entidades do domínio de negócio que são visíveis ao usuário final. Os atributos de cada classe de análise são definidos e os *serviços*[4] requeridos por cada classe são identificados. Os relacionamentos e a colaboração entre classes são identificados e uma variedade de diagramas UML suplementares é produzida.

O resultado final da elaboração é um modelo de análise que define o domínio do problema informacional, funcional e comportamental.

INFO

Modelagem do Sistema

Considere por um momento que lhe foi solicitado especificar todos os requisitos para a construção de uma cozinha *gourmet*. Você sabe as dimensões da sala, a localização das portas e janelas e o espaço de parede disponível.

A fim de especificar totalmente o que deve ser construído, você poderia listar todos os gabinetes e eletrodomésticos (seus fabricantes, número do modelo, dimensões). Você então poderia especificar os revestimentos dos balcões (laminado, granito etc.), peças do encanamento, pavimentação etc. Essas listas poderiam fornecer uma especificação útil, mas elas não fornecem um modelo completo do que você quer. Para completar o modelo, você poderia criar uma apresentação tridimensional que mostrasse a posição dos gabinetes e dos eletrodomésticos e os relacionamentos entre eles. A partir do modelo, seria relativamente fácil avaliar a eficiência do fluxo de trabalho (um requisito para todas as cozinhas) e a "aparência" estética da sala (um requisito pessoal, mas muito importante).

Construímos modelos de análise por motivos muito semelhantes àqueles pelos quais desenvolveríamos uma planta ou uma apresentação em 3D da cozinha. É importante avaliar os componentes do sistema em relação uns com os outros, para determinar como os requisitos se encaixam nesse quadro e avaliar a "estética" do sistema tal como foi concebida.

AVISO

Não deve haver ganhador nem perdedor em uma negociação efetiva. Ambos os lados ganham, pois um "acordo" com o qual ambos podem conviver é solidificado.

7.2.4 Negociação

Não é incomum que clientes e usuários peçam mais do que pode ser conseguido, considerando os recursos limitados do negócio. É também relativamente comum que diferentes clientes ou usuários proponham requisitos conflitantes, argumentando que sua versão é "essencial para nossas necessidades especiais".

O engenheiro de requisitos precisa reconciliar esses conflitos por intermédio de um processo de *negociação*. Clientes, usuários e outros interessados são solicitados a ordenar os requisitos e depois discutir os conflitos de prioridade. Os riscos associados a cada requisito são identificados e analisados (veja o Capítulo 25 para mais detalhes). "Estimativas" grosseiras do esforço de desenvolvimento são feitas e usadas para avaliar o impacto de cada requisito no custo do projeto e no

4 Os termos *operações* e *métodos* também são usados.

Document shared on www.docsity.com

prazo de entrega. Usando uma abordagem iterativa, requisitos são eliminados, combinados e/ou modificados de modo que cada parte alcance algum grau de satisfação.

7.2.5 Especificação

PONTO CHAVE

A formalidade e o formato de uma especificação variam conforme o tamanho e a complexidade do software a ser construído.

No contexto de sistemas (e softwares) baseados em computador, o termo especificação significa coisas diferentes para pessoas diferentes. Uma especificação pode ser um documento escrito, um modelo gráfico, um modelo matemático formal, uma coleção de cenários de uso, um protótipo ou qualquer combinação desses elementos.

Algumas pessoas sugerem que um "gabarito padrão" [SOM97] deve ser desenvolvido e usado para a especificação de sistemas, argumentando que isso leva a requisitos que são apresentados de um modo consistente e, conseqüentemente, mais inteligível. No entanto, algumas vezes é necessário permanecer flexível quando uma especificação precisa ser desenvolvida. Para sistemas grandes, um documento escrito, combinando descrições em linguagem natural e modelos gráficos pode ser a melhor abordagem. Cenários de uso podem, entretanto, ser tudo o que é necessário para produtos ou sistemas menores que residam em ambientes técnicos bem entendidos.

A especificação é o produto de trabalho final produzido pelo engenheiro de requisitos. Ela serve como fundamento das atividades de engenharia de software subseqüentes. Ela descreve a função e o desempenho de um sistema baseado em computador e as restrições que governarão o seu desenvolvimento.

7.2.6 Validação

AVISO

Uma preocupação-chave durante a validação dos requisitos é a consistência. Use o modelo de análise para garantir que os requisitos foram consistentemente declarados.

Os produtos de trabalho resultantes da engenharia de requisitos são avaliados quanto à qualidade durante o passo de *validação*. A validação de requisitos examina a especificação para garantir que todos os requisitos do software tenham sido declarados de modo não ambíguo; que as inconsistências, omissões e erros tenham sido detectados e corrigidos e que os produtos de trabalho estejam de acordo com as normas estabelecidas para o processo, o projeto e o produto.

O principal mecanismo de validação de requisitos é a revisão técnica formal (Capítulo 26). A equipe de revisão que valida os requisitos inclui engenheiros de software, clientes, usuários e outros interessados que examinam a especificação procurando por erros de conteúdo ou de interpretação, áreas em que esclarecimentos podem ser necessários, informações omissas, inconsistências (um problema importante quando produtos ou sistemas de grande porte passam por engenharia), requisitos conflitantes ou requisitos irrealísticos (inatingíveis).

INFO

Checklist de Validação dos Requisitos

É freqüentemente útil examinar cada requisito em face de um conjunto de questões do tipo checklist. Eis aqui um pequeno subconjunto de questões que poderiam ser formuladas:

- Os requisitos foram claramente estabelecidos? Eles podem ser mal interpretados?
- A fonte (por exemplo, pessoa, regulamento, documento) do requisito foi identificada? A declaração final do requisito foi examinada pela fonte original ou com ela?
- O requisito está limitado em termos quantitativos?
- Que outros requisitos se relacionam a este requisito? Eles foram claramente anotados em uma matriz de referência cruzada ou em outro mecanismo?
- O requisito viola alguma restrição do domínio?
- O requisito pode ser testado? Em caso positivo, podemos especificar os testes (algumas vezes chamados *critérios de validação*) para exercitar o requisito?
- Pode-se relacionar o requisito a qualquer modelo de sistema que tenha sido criado?
- O requisito está relacionado aos objetivos globais do sistema/produto?
- A especificação do sistema está estruturada de modo que leve a fácil entendimento, fácil referenciação e fácil tradução em produtos de trabalho mais técnicos?
- Foi criado um índice para a especificação?
- Os requisitos associados ao desempenho, ao comportamento e às características operacionais do sistema foram claramente declarados? Que requisitos parecem estar implícitos?

7.2.7 Gestão de Requisitos

Veja na Web

Uma variedade de informações úteis sobre gestão de requisitos pode ser obtida em www.jiludwig.com.

AVISO

Quando um sistema é grande e complexo, determinar as "conexões" entre os requisitos pode ser uma tarefa difícil. Use tabelas de rastreamento para tornar o serviço um pouco mais fácil.

No Capítulo 6, mencionamos que requisitos para sistemas baseados em computador mudam e que o desejo de mudar os requisitos persiste ao longo da vida do sistema. A *gestão de requisitos* é um conjunto de atividades que ajudam a equipe de projeto a identificar, controlar e rastrear requisitos e modificações de requisitos em qualquer época, à medida que o projeto prossegue.[5] Muitas dessas atividades são idênticas às técnicas de gestão de configuração de software (GCS) discutidas no Capítulo 27.

A gestão de requisitos começa com a identificação. A cada requisito é atribuído um modo identificador. Uma vez identificados os requisitos, tabelas de rastreamento são desenvolvidas. Mostradas esquematicamente na Figura 7.1, cada *tabela de rastreamento* relaciona os requisitos identificados a um ou mais aspectos do sistema ou de seu ambiente. Entre muitas tabelas de rastreamento possíveis estão as seguintes:

Tabela de rastreamento de características. Mostra como os requisitos se relacionam a características importantes do sistema/produto observáveis pelo cliente.

Tabela de rastreamento de fontes. Identifica a fonte de cada requisito.

Tabela de rastreamento de dependência. Indica como os requisitos estão relacionados uns aos outros.

Tabela de rastreamento de subsistemas. Caracteriza os requisitos pelo(s) subsistema(s) que eles governam.

Tabela de rastreamento de interface. Mostra como os requisitos se relacionam com as interfaces internas e externas do sistema.

Em muitos casos, essas tabelas de rastreamento são mantidas como parte do banco de dados de requisitos, de modo que elas possam ser pesquisadas rapidamente para entender como a modificação em um requisito afetará diferentes aspectos do sistema a ser construído.

FERRAMENTAS DE SOFTWARE

Engenharia de Requisitos

Objetivo: As ferramentas de engenharia de requisitos apóiam a obtenção, modelagem, gestão e validação de requisitos.

Mecânica: A mecânica das ferramentas varia. Em geral, as ferramentas de engenharia de requisitos constroem uma variedade de modelos gráficos (por exemplo, UML) que representam os aspectos informacional, funcional e comportamental de um sistema. Esses modelos formam a base para todas as outras atividades do processo de software.

Ferramentas Representativas[6]

Uma lista razoavelmente abrangente (e atualizada) de ferramentas de engenharia de requisitos foi preparada por The Atlantic Systems Guide, Inc. e pode ser encontrada em http://www.systemsguild.com/GuildSite/Robs/retools.html. As ferramentas de modelagem de requisitos serão discutidas no Capítulo 8. As ferramentas mencionadas a seguir enfocam a gestão de requisitos.

EasyRM, desenvolvida pela Cybernetic Inteligence GmbH (www.easy-rm.com), constrói um dicionário/glossário específico para o projeto contendo descrições e atributos detalhados dos requisitos.

OnYourMark Pro, desenvolvida pela Omni-Vista (www.omni-vista.com) constrói um banco de dados de requisitos, estabelece relacionamentos entre os requisitos e permite aos usuários analisar o relacionamento entre requisitos e cronogramas/custos.

Rational RequisitePro, desenvolvida pela Rational Software (www.rational.com), permite aos usuários construir um banco de dados de requisitos, representar os relacionamentos entre os requisitos e organizar, priorizar e rastrear requisitos.

RTM, desenvolvida pela Integrated Chipware (www.chipware.com), é uma ferramenta de descrição e rastreamento de requisitos que também apóia certos aspectos do controle de modificações e gestão de testes.

Deve-se notar que muitas tarefas de gestão de requisitos podem ser realizadas usando uma simples planilha ou um pequeno sistema de banco de dados.

5 A gestão formal de requisitos é iniciada somente para grandes projetos com centenas de requisitos identificáveis. Para projetos pequenos, essa função de engenharia de requisitos é consideravelmente menos formal.

6 As ferramentas mencionadas aqui não representam um endosso, mas, em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas de seus respectivos desenvolvedores.

Document shared on www.docsity.com

FIGURA 7.1

Tabela de rastreamento genérico

Requisito \ Aspecto específico do sistema ou do seu ambiente	A01	A02	A03	A04	A05				Aii
R01			✓		✓				
R02	✓		✓						
R03	✓			✓					✓
R04		✓			✓				
R05	✓	✓		✓					✓
Rnn	✓		✓						

7.3 Início do Processo de Engenharia de Requisitos

Em um ambiente ideal, clientes e engenheiros de software trabalhariam juntos na mesma equipe.[7] Em tais casos, a engenharia de requisitos é simplesmente uma questão de conduzir conversas significativas com colegas que são membros bem conhecidos da equipe. Mas a realidade é muitas vezes bem diferente.

O(s) cliente(s) pode(m) estar em uma cidade ou país diferente, pode(m) ter apenas uma vaga idéia do que é necessário, ter opiniões conflitantes sobre o sistema a ser construído, ter conhecimento técnico limitado e tempo limitado para interagir com o engenheiro de requisitos. Nenhuma dessas coisas é desejável, mas todas são bastante comuns e a equipe de software é freqüentemente forçada a trabalhar dentro das restrições impostas por essa situação.

Nas seções seguintes discutiremos os passos necessários para iniciar a engenharia de requisitos — para colocar o projeto em andamento com o objetivo de mantê-lo em movimento na direção de uma solução bem-sucedida.

7.3.1 Identificação dos Interessados

PONTO CHAVE

Um interessado é quem quer que tenha um interesse direto no sistema a ser desenvolvido ou que se beneficie com ele.

Sommerville e Sawyer [SOM97] definem um *interessado* como "quem quer que se beneficie de modo direto ou indireto do sistema que está sendo desenvolvido". Já identificamos os suspeitos usuais: gerentes de operações do negócio, gerentes de produto, pessoal de marketing, clientes externos e internos, usuários finais, consultores, engenheiros de produto, engenheiros de software, engenheiros de suporte e manutenção e outros. Cada interessado tem uma visão diferente do sistema, obtém diferentes benefícios quando o sistema é desenvolvido com sucesso e está exposto a diferentes riscos se o esforço de desenvolvimento falhar.

Na concepção, o engenheiro de requisitos deve criar uma lista de pessoas que fornecerão entradas à medida que os requisitos forem levantados (Seção 7.4). A lista inicial vai crescer à medida que os interessados forem contactados porque, a cada interessado, será perguntado "com quem mais você acha que eu deveria falar?".

7.3.2 Reconhecimento de Diversos Pontos de Vista

Como existem muitos interessados diferentes, os requisitos do sistema serão explorados a partir de muitos pontos de vista diferentes. O grupo de marketing está interessado em funções e características que excitem o mercado em potencial, tornando o novo sistema fácil de vender. Os gerentes de negócio estão interessados em um conjunto de características que possam ser construídas dentro do orçamento e que estejam prontas a satisfazer janelas de mercado definidas. Os usuários

7 Essa abordagem é recomendada para todos os projetos e é parte integral da filosofia de desenvolvimento ágil de software.

finais podem querer características que lhes sejam familiares e fáceis de aprender e de usar. Engenheiros de software podem estar preocupados com funções que permitam à infra-estrutura dar suporte a mais funções e características vendáveis. Engenheiros de suporte e manutenção podem enfocar a manutenibilidade do software.

> "Coloque três interessados em uma sala e pergunte a eles que tipo de sistema eles desejam. Você provavelmente obterá quatro ou mais opiniões diferentes."
>
> Autor desconhecido

Cada um desses participantes (e outros) contribuirá com informações para o processo de engenharia de requisitos. À medida que a informação de vários pontos de vista é coletada, os requisitos emergentes podem ser inconsistentes ou podem conflitar uns com os outros. O trabalho do engenheiro de requisitos é categorizar todas as informações dos interessados (inclusive requisitos inconsistentes e conflitantes) de modo a permitir que os tomadores de decisão escolham um conjunto de requisitos do sistema internamente consistente.

7.3.3 Trabalho em Busca da Colaboração

Nos capítulos anteriores, notamos que os clientes (e outros interessados) deveriam *colaborar* entre eles (evitando pequenas batalhas vãs) e com os profissionais de engenharia de software se um sistema bem-sucedido precisar ser produzido. Contudo, como essa colaboração é conseguida?

O trabalho do engenheiro de requisitos é identificar áreas de concordância (isto é, requisitos com os quais todos os interessados concordam) e áreas de conflito ou inconsistência (isto é, requisitos que são desejados por um interessado, mas conflitam com as necessidades de outro interessado). É, sem dúvida, esta última categoria que apresenta um desafio.

INFO

Uso de "Pontos de Prioridade"

Um modo de resolver requisitos conflitantes e, ao mesmo tempo, entender melhor a importância relativa de todos os requisitos é usar um esquema "de votação" baseado em *pontos de prioridade*. A todos os interessados é fornecida uma certa quantidade de pontos de prioridade que podem ser "gastos" com qualquer número de requisitos. Uma lista de requisitos é apresentada e cada interessado indica a importância relativa de cada um (de acordo com o seu ponto de vista) gastando um ou mais pontos de prioridade com ele. Os pontos gastos não podem ser reusados. Quando os pontos de prioridade de um interessado se esgotam, nenhuma outra ação sobre requisitos pode ser tomada por essa pessoa. O total de pontos gastos em cada requisito por todos os interessados fornece uma indicação da importância global de cada requisito.

Colaboração não significa necessariamente que os requisitos são definidos por uma comissão. Em muitos casos, os interessados colaboram dando seu ponto de vista dos requisitos, mas um forte "campeão do projeto" (por exemplo, o gerente de negócio ou o técnico sênior) pode tomar a decisão final sobre quais os requisitos que passam pelo corte.

7.3.4 Formulação das Primeiras Questões

Anteriormente neste capítulo, mencionamos que as questões formuladas na concepção do projeto devem ser "livres de contexto" [GAU89]. O primeiro conjunto de questões livres de contexto focaliza o cliente e outros interessados, os objetivos globais e os benefícios. Por exemplo, o engenheiro de requisitos poderia perguntar:

- Quem está por trás da solicitação deste trabalho?
- Quem vai usar a solução?

Document shared on www.docsity.com

- Qual será o benefício econômico de uma solução bem-sucedida?
- Há outra fonte para a solução que você necessita?

Essas questões ajudam a identificar todos os que têm interesse no software a ser construído. Além disso, as questões identificam o benefício mensurável de uma implementação bem-sucedida e possíveis alternativas para o desenvolvimento de software sob encomenda.

> "É melhor saber algumas das questões a todas as respostas."
>
> **James Thurber**

O conjunto de questões a seguir possibilita à equipe de software obter um melhor entendimento do problema e permite que o cliente verbalize sua percepção de uma solução:

? Quais questões vão ajudá-lo a obter um entendimento preliminar do problema?

- Como você caracterizaria "boas" saídas, que seriam geradas por uma solução bem-sucedida?
- Que problema(s) essa solução enfrentaria?
- Você pode me mostrar (ou descrever) o ambiente de negócios no qual a solução será usada?
- Tópicos ou restrições especiais de desempenho afetarão o modo pelo qual a solução é abordada?

O conjunto final de questões focaliza a efetividade da própria atividade de comunicação. Gause e Weinberg [GAU89] as chamam de "metaquestões" e propõem a seguinte lista (resumida):

- Você é a pessoa certa para responder a essas questões? Suas respostas são "oficiais"?
- Minhas questões são relevantes ao problema que você tem?
- Estou formulando muitas questões?
- Alguém mais pode fornecer informações adicionais?
- Devo perguntar-lhe mais alguma coisa?

> "Aquele que formula uma questão é um tolo durante cinco minutos. Aquele que não formula nenhuma questão é um tolo para sempre."
>
> **Provérbio chinês**

Essas questões (e outras) vão ajudar a "quebrar o gelo" e iniciar a comunicação, que é essencial para o levantamento bem-sucedido. Entretanto, uma reunião de perguntas e respostas não é uma abordagem que tem sido sempre um sucesso.

7.4 Levantamento de Requisitos

A forma P&R descrita na Seção 7.3.4 é útil na concepção, mas não é uma abordagem que tenha tido marcante sucesso para o levantamento de requisitos mais detalhados. De fato, a seção de P&R deve ser usada apenas para o primeiro encontro, e depois substituída por uma forma de levantamento de requisitos que combine elementos de solução de problemas, elaboração, negociação e especificação. Uma abordagem desse tipo é apresentada na seção seguinte.

7.4.1 Coleta Colaborativa de Requisitos

A fim de encorajar uma abordagem colaborativa e orientada a equipes para a coleta de requisitos, uma equipe de interessados e desenvolvedores trabalha em conjunto para identificar o

problema, propor elementos da solução, negociar diferentes abordagens e especificar um conjunto preliminar de requisitos da solução [ZAH90].[8]

Muitas abordagens diferentes da *coleta colaborativa de requisitos* têm sido propostas. Cada uma faz uso de um cenário ligeiramente diferente, mas todas aplicam alguma variante das seguintes diretrizes básicas:

? Quais são as diretrizes básicas para conduzir uma reunião de coleta colaborativa de requisitos?

- As reuniões são conduzidas e assistidas por engenheiros de software e por clientes (juntamente com outros interessados). São estabelecidas regras para a preparação e a participação. É sugerida uma agenda suficientemente formal para cobrir todos os pontos importantes, porém suficientemente informal para encorajar o livre fluxo de idéias.
- Um "facilitador" (que pode ser um cliente, um desenvolvedor ou uma pessoa de fora) controla a reunião.
- Um "mecanismo de definição" é usado (podem ser folhas de rascunho, *flip charts* ou papel auto-adesivo, ou um quadro de avisos eletrônico, salas de conversa ou fórum virtual).
- A meta é identificar o problema, propor elementos da solução, negociar diferentes abordagens e especificar um conjunto preliminar de requisitos da solução em uma atmosfera que propicie que a meta seja alcançada.

Para entender melhor o fluxo de eventos à medida que eles ocorrem, apresentamos um cenário resumido que delineia a seqüência de eventos que provocam a reunião de coleta de requisitos e que ocorrem durante e após a reunião.

> "Gastamos muito tempo — a maioria do esforço do projeto — não implementando ou testando, mas tentando decidir o que construir."
>
> **Brian Lawrence**

Veja na Web

O desenvolvimento conjunto de aplicações (JAD — Joint Application Development) é uma técnica popular de coleta de requisitos. Uma boa descrição pode ser encontrada em www.carolla.com/wp-jad.htm.

Durante a concepção (Seção 7.3), perguntas e respostas básicas estabelecem o escopo do problema e a percepção geral de uma solução. Como resultado dessas reuniões iniciais, os interessados redigem uma "solicitação de produto" de uma ou duas páginas. São selecionados um local, data e hora de reunião e é escolhido um facilitador. Membros da equipe de software e de outros departamentos interessados são convidados a comparecer. A solicitação de produto é distribuída a todos os participantes antes da data da reunião.

Enquanto revê a solicitação nos dias que precedem a reunião, é solicitado a cada participante que faça uma lista dos objetos que fazem parte do ambiente que cerca o sistema, outros objetos que devem ser produzidos pelo sistema e objetos que são usados pelo sistema para desempenhar suas funções. Além disso, é solicitado a cada participante que faça outra lista de serviços (processos ou funções) que manipulam ou interagem com os objetos. Finalmente, são também desenvolvidas listas de restrições (por exemplo, custo, tamanho e regras de negócio) e de critérios de desempenho (por exemplo, velocidade, precisão). Os participantes são informados de que não se espera que as listas sejam exaustivas, mas espera-se que elas reflitam a percepção do sistema de cada um.

Como exemplo,[9] considere um extrato de documento de pré-reunião redigido por uma pessoa de marketing envolvida no projeto *CasaSegura*. Essa pessoa escreve a seguinte narrativa da *função de segurança residencial* que deve ser parte do *CasaSegura*:

> Nossa pesquisa indica que o mercado de sistemas de gestão residencial está crescendo a uma taxa de 40% ao ano. A primeira função do *CasaSegura* que levaremos ao mercado deve ser a função de segurança residencial. A maioria das pessoas está familiarizada com sistemas de "alarme", assim essa seria uma venda fácil.

8 Essa abordagem é algumas vezes chamada de *técnicas facilitadas de especificação de aplicação (Facilitated Application Specification Techniques — FAST)*.

9 O exemplo *CasaSegura* (com extensões e variações) é usado para ilustrar importantes métodos de engenharia de software em vários capítulos seguintes. Como exercício, valeria a pena conduzir sua própria reunião de coleta de requisitos e desenvolver um conjunto de listas para ela.

Document shared on www.docsity.com

> A função de segurança residencial protegeria contra e/ou reconheceria várias "situações" indesejáveis tais como entrada ilegal, fogo, inundação, níveis de monóxido de carbono e outras. Ela usará nossos sensores sem fio para detectar cada situação, poderá ser programada pelo proprietário e discará automaticamente para a agência de monitoração sempre que uma situação for detectada.

Na verdade, outros contribuiriam para essa narrativa durante a reunião de coleta de requisitos e muito mais informações ficariam disponíveis. Mas, mesmo com informações adicionais, a ambigüidade estaria presente, omissões provavelmente existiriam e erros poderiam ocorrer. Por enquanto, a "descrição funcional" precedente será suficiente.

AVISO

Se um sistema ou produto for servir a muitos usuários, esteja absolutamente certo de que os requisitos foram levantados de uma seção transversal representativa dos usuários. Se apenas um usuário definir todos os requisitos, o risco de aceitação é alto.

A equipe de coleta de requisitos é composta de representantes de marketing, da engenharia de software e hardware, e de fabricação. Um facilitador externo deve ser usado.

Cada pessoa da equipe desenvolve as listas descritas anteriormente. Os objetos descritos para o *CasaSegura* poderiam incluir o painel de controle, detectores de fumaça, sensores para janelas e portas, detectores de movimento, alarme, um evento (um sensor foi ativado), um mostrador, um PC, números de telefone, uma chamada telefônica e assim por diante. A lista de serviços poderia incluir *configurar* o sistema, *ligar* o alarme, *monitorar* os sensores, *discar* o telefone, *programar* o painel de controle, *ler* o mostrador (note que os serviços agem sobre os objetos). De modo semelhante, cada participante desenvolverá listas de restrições (por exemplo, o sistema deve reconhecer quando os sensores não estão funcionando, deve ser amigável ao uso, deve fazer interface diretamente com uma linha telefônica padrão) e os critérios de desempenho (por exemplo, um evento de sensoriamento deve ser reconhecido dentro de um segundo, um esquema de prioridade de eventos deve ser implementado).

> "Fatos não deixam de existir porque são ignorados."
>
> Aldous Huxley

Quando a reunião de coleta de requisitos começa, o primeiro tópico de discussão é a necessidade e a justificativa do novo produto — todos devem concordar que o produto é justificável. Uma vez estabelecida a concordância, cada participante apresenta suas listas para discussão. As listas podem ser penduradas nas paredes da sala usando grandes folhas de papel, grudadas com adesivo na parte de trás ou escritas em um quadro de parede. Alternativamente, as listas podem ser colocadas em um quadro de avisos eletrônico, em um site interno da Web ou em um ambiente de bate-papo eletrônico para revisão anterior à reunião. O ideal é que cada entrada em uma lista possa ser manipulada separadamente para que as listas possam ser combinadas, as entradas apagadas e adições possam ser feitas. Nesse estágio, críticas e debates são estritamente proibidos.

Depois que as listas individuais sobre um assunto forem apresentadas, uma lista combinada é criada pelo grupo. A lista combinada elimina entradas redundantes, adiciona qualquer idéia nova que tenha surgido durante a discussão, mas não apaga nada. Depois que as listas combinadas para todos os assuntos tiverem sido criadas, o facilitador coordena a discussão. A lista combinada é reduzida, ampliada ou reescrita para refletir adequadamente o produto/sistema a ser desenvolvido. O objetivo é desenvolver uma *lista de consenso* em cada assunto (objetos, serviços, restrições e desempenho). As listas são então postas de lado para ação posterior.

AVISO

Evite o impulso de atacar uma idéia do cliente como "muito cara" ou "não-prática". A idéia aqui é negociar uma lista que seja aceitável para todos. Para fazer isso, você deve estar com a mente aberta.

Uma vez completadas as listas de consenso, a equipe é subdividida em equipes menores; cada uma trabalha para desenvolver *miniespecificações* para uma ou mais entradas de cada uma das listas.[10] Cada miniespecificação é um detalhamento da palavra ou frase que consta de uma lista. Por exemplo, a miniespecificação para o objeto **Painel de Controle** do *CasaSegura* poderia ser:

> O **Painel de Controle** é uma unidade montada na parede que tem o tamanho aproximado de 22 x 12 centímetros. Ele tem conectividade sem fio a sensores e a um PC. A interação com o usuário ocorre por meio do teclado padrão de 12 teclas. Um mostrador LCD de 5x5 centímetros, aproximadamente, fornece o feedback do usuário. O software fornece *prompts* interativos, eco e funções similares.

10 Em vez de criar miniespecificações, muitas equipes de software decidem desenvolver cenários do usuário, chamados de *casos de uso*. Eles serão considerados em detalhe na Seção 7.5.

Cada subequipe apresenta, em seguida, sua miniespecificação a todos os participantes para discussão. Adições, supressões e mais detalhamentos são feitos. Em alguns casos, o desenvolvimento de miniespecificações descobrirá novos objetos, serviços, restrições ou requisitos de desempenho que serão adicionados às listas originais. Durante todas as discussões, a equipe pode levantar um assunto que não pode ser resolvido durante a reunião. Uma *lista de assuntos* é mantida de modo que essas idéias possam ser trabalhadas depois.

Depois que as miniespecificações são completadas, cada participante faz uma lista de critérios de validação para o produto/sistema e apresenta-a à equipe. Uma lista de consenso dos critérios de validação é criada. Finalmente, atribui-se a um ou mais participantes (ou pessoas externas) a tarefa de redigir o rascunho completo da especificação usando todas as entradas produzidas durante a reunião.

CASASEGURA

Condução de uma Reunião de Coleta de Requisitos

A cena: Uma sala de reunião. A primeira reunião de coleta de requisitos está em progresso.

Os personagens: Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software; Ed Robbins, membro da equipe de software; Doug Miller, gerente de engenharia de software; três membros de marketing; um representante de engenharia de produto e um facilitador.

A conversa:

Facilitador (apontando para um quadro): Então, esta é a lista atual de objetos e serviços para a função de segurança residencial.

Pessoa de Marketing: Isso praticamente cobre tudo do nosso ponto de vista.

Vinod: Alguém não mencionou que queria toda a funcionalidade do *CasaSegura* acessível via Internet? Isso incluiria a função de segurança residencial, não?

Pessoa de Marketing: É, está certo... vamos ter de acrescentar essa funcionalidade e os objetos adequados.

Facilitador: Isso também acrescenta algumas restrições?

Jamie: Sim, tanto técnicas quanto legais.

Representante da Produção: O que significa?

Jamie: Precisamos estar certos de que uma pessoa de fora não pode invadir o sistema, desarmá-lo e roubar o lugar ou pior. Pesada responsabilidade de nossa parte.

Doug: Muito certo.

Marketing: Mas nós ainda precisamos de conectividade via Internet... basta nos certificarmos de impedir a invasão de uma pessoa de fora.

Ed: Isso é mais fácil de falar do que de fazer e...

Facilitador (interrompendo): Eu não quero discutir esse ponto agora. Vamos anotá-lo como um item de ação e prosseguir.

(Doug, servindo como secretário da reunião faz a anotação adequada.)

Facilitador: Tenho um sentimento de que existe ainda mais a considerar.

(O grupo gasta os 45 minutos seguintes refinando e expandindo os detalhes da função de segurança residencial.)

PONTO CHAVE

A IFQ define requisitos de modo a maximizar a satisfação do cliente.

7.4.2 Implantação da Função de Qualidade

A *Implantação da Função de Qualidade* (IFQ) (*Quality Function Deployment*, QFD) é uma técnica que traduz as necessidades do cliente para requisitos técnicos do software. A IFQ "concentra-se em maximizar a satisfação do cliente a partir do processo de engenharia de software" [ZUL92]. Para conseguir isso, a IFQ enfatiza o entendimento do que tem valor para o cliente e depois implanta esses valores por meio do processo de engenharia. A IFQ identifica três tipos de requisitos [ZUL92]:

Requisitos normais. Esses requisitos refletem os objetivos e metas estabelecidos para um produto ou sistema durante as reuniões com o cliente. Se esses requisitos estiverem presentes, o cliente fica satisfeito. Exemplos de requisitos normais podem ser os tipos de mostradores gráficos requeridos, funções específicas do sistema e níveis de desempenho definidos.

Requisitos esperados. Esses requisitos estão implícitos no produto ou sistema e podem ser tão fundamentais que o cliente não se refere a eles explicitamente. Sua ausência seria causa de

Document shared on www.docsity.com

AVISO

Todos querem implementar muitos requisitos excitantes, mas tenha cuidado. É assim que a "febre de requisitos" se instala. Por outro lado, freqüentemente os requisitos excitantes levam a um produto que se sobressai!

insatisfação significativa. Exemplos de requisitos esperados são: facilidade de interação homem/máquina, correção e confiabilidade operacional global e facilidade de instalação do software.

Requisitos excitantes. Esses requisitos refletem características que vão além das expectativas do cliente e mostram ser muito satisfatórias quando presentes. Por exemplo, um software de processamento de texto é solicitado com as características padrão. O produto entregue dispõe de várias facilidades para leiaute de página que são muito agradáveis e inesperadas.

Na verdade, a IFQ cobre todo o processo de engenharia [PAR96]. No entanto, muitos conceitos da IFQ são aplicáveis à atividade de levantamento de requisitos. Apresentamos apenas um panorama desses conceitos (adaptados para software de computadores) nos parágrafos seguintes.

> "Freqüentemente as expectativas falham, e mais freqüentemente ainda onde eram mais promissoras."
>
> William Shakespeare

Veja na Web

Informações úteis sobre a IFQ podem ser obtidas em www.qfdi.org.

Em reuniões com o cliente, a *implantação de funções* é usada para determinar o valor de cada função necessária ao sistema. A *implantação da informação* identifica tanto os objetos de dados quanto os eventos que o sistema deve consumir e produzir. Eles são ligados às funções. Finalmente, a *implantação de tarefas* examina o comportamento do sistema ou produto dentro do contexto do seu ambiente. A *análise de valor* é conduzida para determinar a prioridade relativa dos requisitos determinados durante cada uma das três implantações.

A IFQ usa entrevistas com o cliente e observação, levantamentos e exame de dados históricos (por exemplo, relatórios de problemas) como dados brutos para a atividade de levantamento de requisitos. Esses dados são então traduzidos em uma tabela de requisitos — chamada de *tabela da voz do cliente* — que é revisada com o cliente. Diversos diagramas, matrizes e métodos de avaliação são então usados para levantar os requisitos esperados e para tentar derivar os requisitos excitantes [BOS91].

CASASEGURA

Desenvolvimento de um Cenário de Usuário Preliminar

A cena: Uma sala de reunião, continuando a primeira reunião de coleta de requisitos.

Os personagens: Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software; Ed Robbins, membro da equipe de software; Doug Miller, gerente de engenharia de software; três membros de marketing, um representante de engenharia de produção e um facilitador.

A conversa:

Facilitador: Estivemos conversado sobre a segurança de acesso à funcionalidade do *CasaSegura* que vai ficar acessível via Internet. Eu gostaria de tentar algo.

Vamos desenvolver um cenário de usuário para o acesso à função de segurança residencial.

Jamie: Como?

Facilitador: Podemos fazê-lo de diferentes modos, mas, por agora, eu gostaria de conservar as coisas realmente informais. Conte-nos (ele aponta para uma pessoa de marketing) como você imagina o acesso ao sistema.

Pessoa de Marketing: Hmm, bem, isso é o tipo de coisa que eu faria se estivesse longe de casa e tivesse de deixar alguém entrar nela, por exemplo, uma arrumadeira ou uma pessoa de manutenção que não tivesse o código de segurança.

Facilitador (sorrindo): Essa é razão pela qual você faria isso... diga-me como você realmente faria isso.

Pessoa de Marketing: Hmm... a primeira coisa que eu precisaria seria de um PC. Eu entraria no site que nós manteríamos para todos os usuários do *CasaSegura*. Forneceria a minha ID de usuário e...

Vinod (interrompendo): A página Web teria de ser segura, criptografada para garantir que estivéssemos seguros e...

Facilitador (interrompendo): Essa é uma boa informação, Vinod, mas é técnica. Vamos apenas nos concentrar sobre como o usuário final vai usar a possibilidade, certo?

Vinod: Sem problemas.

Pessoa de Marketing: Então, como eu estava dizendo, eu entraria em um site e forneceria minha ID de usuário e dois níveis de senha.

Jamie: E se eu esquecesse minha senha?

Facilitador (interrompendo): Boa idéia, Jamie, mas não vamos cuidar disso agora. Vamos fazer uma anotação sobre isso e chamá-la de "exceção". Estou certo de que haverá outras.

Pessoa de Marketing: Depois que eu introduzisse as senhas, uma tela representando todas as funções de *CasaSegura* apareceria. Eu selecionaria a função segurança residencial. O sistema poderia solicitar que eu verificasse quem sou, digamos perguntando o meu endereço ou o número de telefone ou alguma outra coisa. Depois, ele exibiria uma figura do painel de controle do sistema de segurança junto com uma lista de funções que eu poderia realizar — armar o sistema, desarmar o sistema, desarmar um ou mais sensores. Acho que ele poderia também me permitir reconfigurar zonas de segurança e outras coisas análogas, mas não estou certo disso.

(À medida que a pessoa de marketing continua a falar, Doug faz várias anotações. Elas formam a base do primeiro cenário de caso de uso informal. Alternativamente, a pessoa de marketing poderia ter sido solicitada a redigir o cenário, mas isso seria feito fora da reunião.)

7.4.3 Cenários de Usuários

À medida que os requisitos são coletados, uma visão geral das funções e características do sistema começam a se materializar. No entanto, é difícil avançar para atividades mais técnicas de engenharia de software até que a equipe de software entenda como essas funções e características serão usadas por diferentes classes de usuários finais. Para conseguir isso, desenvolvedores e usuários podem criar um conjunto de cenários que identifiquem uma linha de uso para o sistema a ser construído. Os cenários, freqüentemente chamados de *casos de uso* [JAC92], fornecem uma descrição de como o sistema será usado. Casos de uso serão discutidos em maiores detalhes na Seção 7.5.

7.4.4 Produtos de Trabalho do Levantamento

Os produtos de trabalho produzidos em conseqüência do levantamento de requisitos variam dependendo do tamanho do sistema ou do produto a ser construído. Para a maioria dos sistemas, os produtos de trabalho incluem:

? Que informações são produzidas como conseqüência da coleta de requisitos?

- Uma declaração da necessidade e da viabilidade.
- Uma afirmação limitada do escopo do sistema ou do produto.
- Uma lista de clientes, usuários e outros interessados que participaram do levantamento de requisitos.
- Uma descrição do ambiente técnico do sistema.
- Uma lista de requisitos (preferencialmente organizada por função) e as restrições de domínio que se aplicam a cada um deles.
- Um conjunto de cenários de uso que fornecem informações sobre o uso do sistema ou do produto sob diferentes condições de operação.
- Quaisquer protótipos desenvolvidos para definir melhor os requisitos.

Cada um desses produtos de trabalho é revisado por todas as pessoas que participaram do levantamento de requisitos.

7.5 DESENVOLVIMENTO DE CASOS DE USO

Em um livro que discute como redigir casos de uso efetivos, Alistair Cockburn [COC01] observa que "um caso de uso se refere a um contrato...[que] descreve o comportamento do sistema sob várias condições em que o sistema responde a uma solicitação de um dos seus interessados". Em essência, um *caso de uso* conta uma história estilizada sobre como um usuário final (alguém desempenhando um entre vários papéis possíveis) interage com o sistema sob um conjunto específico de circunstâncias. A história pode ser um texto narrativo, um delineamento das tarefas ou interações, uma descrição baseada em gabarito ou uma representação diagramática. Inde-

Document shared on www.docsity.com

pendentemente de sua forma, um caso de uso descreve o software ou o sistema do ponto de vista do usuário.

PONTO CHAVE

Os casos de uso são definidos do ponto de vista de um ator. O ator é um papel que a pessoa (usuário) ou dispositivo desempenha quando interage com o software.

O primeiro passo ao escrever um caso de uso é definir o conjunto de "atores" que estarão envolvidos na história. *Atores* são as diferentes pessoas (ou dispositivos) que usam o sistema ou produto dentro do contexto da função e do comportamento que devem ser descritos. Atores representam os papéis que pessoas (ou dispositivos) desempenham quando o sistema está em operação. Definido mais formalmente, um ator é qualquer coisa que se comunique com o sistema ou o produto e que seja externa ao sistema em si. Cada ator tem um ou mais objetivos quando usa o sistema.

É importante notar que um ator e um usuário final não são necessariamente a mesma coisa. O usuário típico pode desempenhar vários papéis diferentes quando usa um sistema, enquanto o ator representa uma classe de entidades externas (freqüentemente, mas nem sempre, pessoas) que desempenham apenas um papel no contexto do caso de uso. Como exemplo, considere um operador de máquina (um usuário) que interaja com o computador de controle de uma célula de fabricação que contém um certo número de robôs e máquinas de controle numérico. Após cuidadosa revisão dos requisitos, o software para o computador de controle requer quatro diferentes modos (papéis) de interação: modo de programação, modo de teste, modo de monitoração e modo de reparação. Assim, quatro atores podem ser definidos: programador, testador, monitor e reparador. Em alguns casos, o operador de máquina pode desempenhar todos esses papéis. Em outros, diferentes pessoas podem desempenhar o papel de cada ator.

Veja na Web

Um artigo excelente sobre casos de uso pode ser encontrado em www.rational.com/products/whitepapers/100622.jsp.

Como o levantamento de requisitos é uma atividade evolutiva, nem todos os atores são identificados durante a primeira iteração. É possível identificar atores principais [JAC92] durante a primeira iteração, e atores secundários quando se fica sabendo mais a respeito do sistema. Os *atores principais* interagem para conseguir a função desejada do sistema e derivar o benefício pretendido com o sistema. Eles trabalham direta e freqüentemente com o software. Os *atores secundários* dão suporte ao sistema, de modo que os atores principais possam fazer seu trabalho.

Uma vez identificados os atores, casos de uso podem ser desenvolvidos. Jacobson [JAC92] sugere algumas questões[11] que deveriam ser respondidas por um caso de uso:

? O que é necessário saber para desenvolver um caso de uso efetivo?

- Quem é(são) o(s) ator(es) principal(is) e o(s) ator(es) secundário(s)?
- Quais são as metas dos atores?
- Que pré-condições devem existir antes da história começar?
- Que tarefas ou funções principais são desempenhadas pelo ator?
- Que exceções deveriam ser consideradas quando a história é descrita?
- Que variações na interação dos atores são possíveis?
- Que informações do sistema o ator vai adquirir, produzir ou modificar?
- O ator terá de informar o sistema sobre alterações no ambiente externo?
- Que informações o ator deseja do sistema?
- O ator deseja ser informado sobre modificações inesperadas?

Recordando os requisitos básicos do *CasaSegura*, podemos definir três atores: o **proprietário** (um usuário), um **gerente de configuração** (provavelmente, a mesma pessoa que o **proprietário**, mas desempenhando um papel diferente), **sensores** (dispositivos ligados ao sistema) e o **subsistema de monitoração** (a estação central que monitora a função de segurança residencial do *CasaSegura*). Para a finalidade deste exemplo, consideramos apenas o ator **proprietário**. O proprietário interage com a função de segurança residencial de um certo número de modos diferentes usando ou o painel de controle do alarme ou um PC:

11 As questões de Jacobson foram estendidas para fornecer uma visão mais completa do conteúdo do caso de uso.

- Ele introduz uma senha para permitir todas as outras interações.
- Consulta o estado de uma zona de segurança.
- Consulta o estado de um sensor.
- Aperta o botão de pânico em uma emergência.
- Ativa/desativa o sistema de segurança.

Considerando a situação em que o proprietário usa o painel de controle, o caso de uso básico para o sistema de ativação é o seguinte:[12]

1. O proprietário observa o painel de controle do *CasaSegura* (Figura 7.2) para determinar se o sistema está disponível para a entrada. Se o sistema não estiver disponível, uma mensagem *não disponível* será exibida no mostrador LCD e o proprietário deverá fechar fisicamente as janelas/portas para que a mensagem *não disponível* desapareça. (Uma mensagem *não disponível* implica que um sensor está aberto, isto é, que uma porta ou janela está aberta.)
2. O proprietário usa o teclado para teclar uma senha de quatro dígitos. A senha é comparada com a senha válida armazenada no sistema. Se a senha não for correta, o painel de controle emitirá um bip e preparar-se-á para uma entrada adicional. Se a senha estiver correta, o painel de controle esperará a ação subseqüente.
3. O proprietário seleciona e tecla *dentro* ou *fora* (veja a Figura 7.2) para ativar o sistema. Dentro ativa somente os sensores periféricos (os sensores internos de detecção de movimento são desativados). Fora ativa todos os sensores.
4. Quando a ativação ocorre, uma luz de alarme vermelha pode ser observada pelo proprietário.

AVISO

Casos de uso são freqüentemente escritos informalmente. No entanto, usar o gabarito mostrado aqui garante que você abordará todos os tópicos-chave.

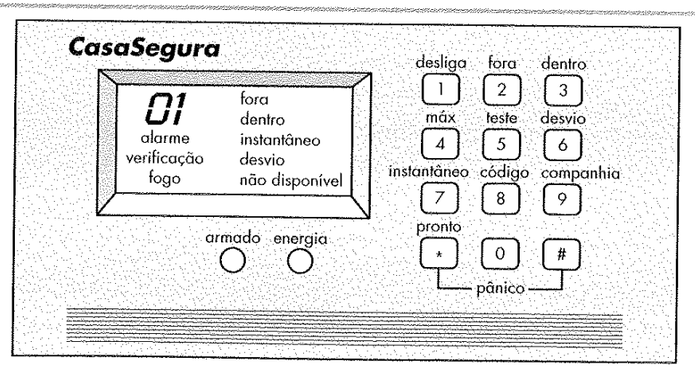
O caso de uso básico apresenta uma história de alto nível que descreve a interação entre o ator e o sistema.

Em muitas instâncias, os casos de uso são elaborados para fornecer consideravelmente mais detalhes sobre a interação. Por exemplo, Cockburn [COC01] sugere o seguinte gabarito para descrições detalhadas de casos de uso:

Caso de uso: *IniciarMonitoração*
Ator principal: Proprietário.

FIGURA 7.2

Painel de Controle do *CasaSegura*



12 Note que esse caso de uso difere da situação em que se tem acesso ao sistema via Internet. Nesse caso, a interação ocorre pelo painel de controle e não pela GUI fornecida quando um PC é usado.

Document shared on www.docsity.com

Meta no contexto: Iniciar o sistema para monitorar os sensores quando o proprietário sair de casa ou permanecer dentro dela.

Precondições: O sistema foi programado para uma senha e para reconhecer vários sensores.

Disparo: O proprietário decide "iniciar" o sistema, isto é, ligar as funções de alarme.

Cenário:

1. Proprietário: observa o painel de controle.
2. Proprietário: introduz a senha.
3. Proprietário: seleciona "dentro" ou "fora".
4. Proprietário: observa luz vermelha do alarme indicando que o *CasaSegura* foi ligado.

Exceções:

1. O painel de controle indica *não disponível*: o proprietário verifica todos os sensores para determinar quais estão abertos, fechando-os.
2. Senha incorreta (painel de controle soa uma vez): proprietário introduz novamente a senha correta.
3. Senha não reconhecida: o subsistema de monitoração e resposta deve ser contatado para reprogramar a senha.
4. *Dentro* é selecionado: o painel de controle soa duas vezes e a luz *dentro* acende; os sensores periféricos são ativados.
5. *Fora* é selecionado: o painel de controle soa três vezes e a luz *fora* acende; todos os sensores são ativados.

Prioridade: Essencial, deve ser implementada.

Quando disponível: Primeiro incremento.

Freqüência de uso: Muitas vezes por dia.

Canal para o ator: Via interface do painel de controle.

Atores secundários: Técnico de apoio, sensores.

Canais para atores secundários:

Técnico de apoio: linha telefônica.
Sensores: interfaces em hardware e sem fio.

Tópicos em aberto:

1. Deve haver um modo de ativar o sistema sem o uso de uma senha ou com uma senha abreviada?
2. O painel de controle deveria mostrar mensagens de texto adicionais?
3. Quanto tempo o proprietário tem para introduzir a senha a partir do momento que pressionou a primeira tecla?
4. Há um modo de desativar o sistema antes que ele seja realmente ativado?

PONTO CHAVE

Cada caso de uso fica acessível aos interessados e as prioridades relativas a cada um podem ser atribuídas.

Casos de uso para outras interações do **proprietário** seriam desenvolvidos de maneira similar. É importante notar que cada caso de uso deve ser revisado com cuidado. Se algum elemento da interação for ambíguo, é muito provável que uma revisão do caso de uso descubra o problema.

CASASEGURA

Desenvolvimento de um Diagrama de Caso de Uso de Alto Nível

A cena: Uma sala de reuniões continuando a reunião de coleta de requisitos.

Os personagens: Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software; Ed Robbins, membro da equipe de software; Doug Miller, gerente de engenharia de software; três membros de marketing, um representante de engenharia de produto e um facilitador.

A conversa:

Facilitador: Nós gastamos uma quantidade considerável de tempo falando sobre a funcionalidade de segurança residencial do *CasaSegura*. Durante o intervalo esbocei um diagrama de caso de uso para resumir os cenários importantes que fazem parte dessa função. Dêem uma olhada.

(Todos os presentes olham para a Figura 7.3.)

Jamie: Eu estou apenas começando a aprender a notação UML. Assim, a função de segurança residencial está representada pela caixa grande com as elipses dentro dela? E as elipses representam os casos de uso que nós escrevemos no texto?

Facilitador: Sim. E as figuras dos bonecos representam atores — as pessoas ou coisas que interagem com o sistema como descrito pelo caso de uso... ah, eu uso o quadrado rotulado para representar um ator que não é uma pessoa, neste caso, os sensores.

Doug: Isso é válido na UML?

Facilitador: Validade não é a questão. O ponto é comunicar a informação. Eu vejo o uso de uma figura de um boneco para representar um dispositivo como levando a erro. Assim, adaptei um pouco as coisas. Não acho que isso crie um problema.

Vinod: Tudo bem, então nós temos narrativas de casos de uso para cada uma das elipses. Precisamos desenvolver as narrativas mais detalhadas com base nos gabaritos, sobre as quais eu li?

Facilitador: Provavelmente, mas isso pode esperar até que tenhamos considerado outras funções do *CasaSegura*.

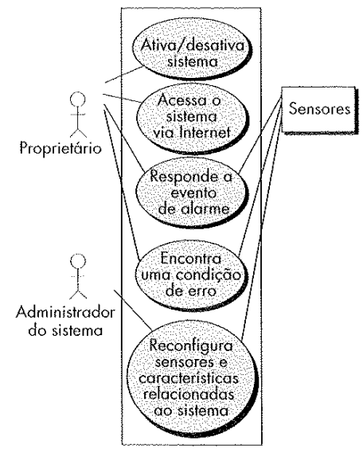
Pessoa de Marketing: Calma, eu fiquei olhando esse diagrama e subitamente percebi que nós omitimos alguma coisa.

Facilitador: Ah, é? Conte-me o que nós omitimos.

(A reunião continua.)

FIGURA 7.3

Diagrama de caso de uso para a função de segurança residencial do *CasaSegura*



Document shared on www.docsity.com

FERRAMENTAS DE SOFTWARE

Desenvolvimento de Caso de Uso

Objetivo: Auxiliar no desenvolvimento de casos de uso fornecendo gabaritos automáticos e mecanismos para avaliar a clareza e a consistência.

Mecânica: A mecânica das ferramentas varia. Em geral, as ferramentas de casos de uso fornecem gabaritos de preenchimento de espaços para criar casos de uso efetivos. A maioria das funcionalidades do caso de uso está embutida em um conjunto de funções mais amplas de engenharia de requisitos.

Ferramentas Representativas:[13]

Clear Requirement Workbench, desenvolvida pela LiveSpecs Software (www.livespecs.com), fornece suporte automatizado para a criação e avaliação de casos de uso bem como uma variedade de outras funções de engenharia de requisitos.

A grande maioria de ferramentas de modelagem de análise baseadas na UML fornece tanto apoio textual quanto gráfico para o desenvolvimento e modelagem de casos de uso.

Objects by Design, uma fonte de ferramentas UML (www.objectsbydesign.com/tools/umtools_byCompany.html) fornece *links* abrangentes para ferramentas desse tipo.

Uma variedade de gabaritos de casos de uso e um banco de dados para apoiá-los podem ser encontrados em UseCase.org (www.usecases.org).

7.6 Construção do Modelo de Análise

O objetivo do modelo de análise é fornecer uma descrição dos domínios informacional, funcional e comportamental necessários a um sistema baseado em computador. O modelo se modifica dinamicamente à medida que os engenheiros de software aprendem mais sobre o sistema a ser construído e os interessados compreendem mais sobre o que eles realmente precisam. Por essa razão, o modelo de análise é um instantâneo dos requisitos em um determinado instante. Espera-se que ele se modifique.

À medida que o modelo de análise evolui, certos elementos tornam-se relativamente estáveis, fornecendo uma base sólida para as tarefas posteriores de projeto. No entanto, outros elementos do modelo podem ser mais voláteis, indicando que o cliente ainda não entende totalmente os requisitos do sistema.

O modelo de análise e os métodos usados para construí-lo são apresentados em detalhes no Capítulo 8. Nas seções seguintes, apresentamos um breve panorama.

7.6.1 Elementos do Modelo de Análise

Há muitas maneiras diferentes de cuidar dos requisitos de um sistema baseado em computador. Algumas pessoas de software alegam que é melhor selecionar um modo de representação (por exemplo, casos de uso) e aplicá-lo com exclusão de todos os outros modos. Outros profissionais acreditam que vale a pena usar um certo número de diferentes modos de representação para descrever o modelo de análise. Diferentes modos de representação forçam a equipe de software a considerar requisitos de diferentes pontos de vista — uma abordagem que tem maior probabilidade de descobrir omissões, inconsistências e ambigüidades.

Os elementos específicos do modelo de análise são ditados pelo método de modelagem de análise (Capítulo 8) que será usado. No entanto, um conjunto de elementos genéricos é comum à maioria dos modelos de análise:

Elementos baseados em cenários. O sistema é descrito do ponto de vista do usuário usando uma abordagem com base em cenário. Por exemplo, casos de uso básicos (Seção 7.5) e seus respectivos diagramas de caso de uso (Figura 7.3) evoluem para casos de uso mais elaborados baseados

13 As ferramentas mencionadas aqui não representam uma recomendação, mas, em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas de seus respectivos desenvolvedores.

AVISO

É sempre uma boa idéia envolver os interessados. Um dos melhores modos de fazer isso é pedir a cada interessado que escreva casos de uso que descrevam como o software será usado.

em gabarito. Os elementos com base em cenários do modelo de análise são, freqüentemente, a primeira parte do modelo de análise a ser desenvolvida. Como tal, eles servem como entrada para a criação de outros elementos de modelagem.

Uma abordagem ligeiramente diferente da modelagem baseada em cenário representa as atividades ou funções que foram definidas como parte da tarefa de levantamento de requisitos. Essas funções existem dentro de um contexto de processamento. Isto é, a seqüência de atividades (os termos *funções* ou *operações* também podem ser usados) que descreve o processamento dentro de um contexto limitado é definida como parte do modelo de análise. Como a maioria dos elementos do modelo de análise (e outros modelos de engenharia de software), atividades (funções) podem ser representadas em muitos níveis diferentes de abstração. Modelos nessa categoria podem ser definidos iterativamente. Cada iteração fornece detalhes adicionais de processamento. Como exemplo, a Figura 7.4 mostra um diagrama de atividade UML para o levantamento de requisitos.[14] Três níveis de detalhe são mostrados.

AVISO

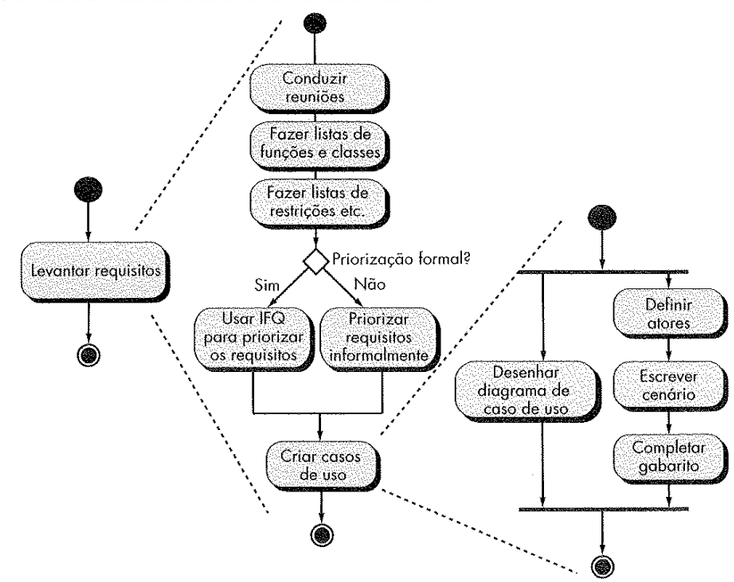
Um modo de isolar classes é procurar substantivos descritivos em um texto de caso de uso. Pelo menos alguns dos substantivos serão candidatos a classes. Mais sobre isso no Capítulo 8.

Elementos baseados em classe. Cada cenário de uso implica em um conjunto de "objetos" que são manipulados à medida que um ator interage com o sistema. Esses objetos são categorizados em classes — uma coleção de coisas que têm atributos semelhantes e comportamentos comuns. Por exemplo, um diagrama de classe pode ser usado para descrever uma classe **Sensor** para a função de segurança do *CasaSegura* (Figura 7.5). Note que o diagrama lista os atributos de sensores (por exemplo, **nome/id**, **tipo**) e as operações [por exemplo, *identificar(), armar()*] que podem ser aplicadas para modificar esses atributos. Além dos diagramas de classe, outros elementos de modelagem de análise descrevem o modo pelo qual as classes colaboram umas com as outras e os relacionamentos e interações entre classes. Isso é discutido em mais detalhes no Capítulo 8.

Elementos comportamentais. O comportamento de um sistema baseado em computador pode ter um profundo efeito sobre o projeto que é escolhido e a abordagem de implementação que é aplicada. Assim, o modelo de análise deve fornecer elementos de modelagem que descrevem comportamento.

FIGURA 7.4

Diagramas de atividade para o levantamento de requisitos

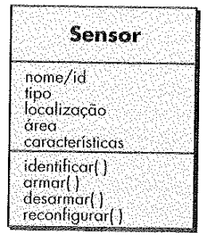


14 O diagrama de atividade é similar a um fluxograma — um diagrama gráfico para representar seqüências de fluxo de controle e de lógica (Capítulo 11).

Document shared on www.docsity.com

FIGURA 7.5

Diagrama de classes para Sensor



PONTO CHAVE

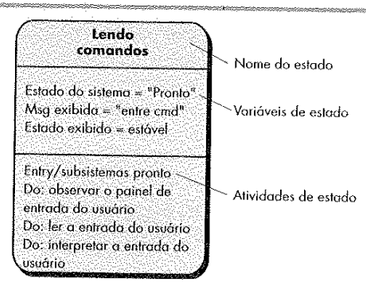
O estado é um modo de comportamento externamente observável. Estímulos externos causam transição entre estados.

O *diagrama de estado* (Capítulo 8) é um método para representar o comportamento de um sistema pela representação de seus estados e dos eventos que causam a modificação do estado do sistema. Um *estado* é qualquer modo de comportamento observável. Além disso, o diagrama de estado indica que ações (por exemplo, ativação de processo) são realizadas em conseqüência de um determinado evento.

Para ilustrar o diagrama de estado, considere o estado *lendo comandos* para uma fotocopiadora de escritório. A notação de diagrama de estado UML é mostrada na Figura 7.6. Um retângulo com cantos arredondados representa um estado. O retângulo é dividido em três áreas: (1) o *nome do estado* (por exemplo, Lendo comandos), (2) *variáveis de estado* que indicam como o estado se manifesta para o mundo exterior, e (3) *atividades de estado* que indicam como o estado é iniciado **(entry/)** e as ações **(do:)** que são invocadas dentro do estado.

FIGURA 7.6

Notação do diagrama de estado UML



CASASEGURA

Modelagem Comportamental Inicial

A cena: Uma sala de reuniões, continuando a reunião de requisitos.

Os personagens: Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software; Ed Robbins, membro da equipe de software; Doug Miller, gerente de engenharia de software; três membros de marketing; um representante de engenharia do produto; e um facilitador.

A conversa:

Facilitador: Nós estamos quase acabando de falar sobre a funcionalidade de segurança residencial do *CasaSegura*. Mas, antes disso, eu quero discutir o comportamento da função.

Pessoa de marketing: Eu não entendo o que você quer dizer por comportamento.

Ed (rindo): É quando você dá ao produto um "limite" se ele se comporta mal.

Facilitador: Não exatamente. Deixe-me explicar.

(O facilitador explica as coisas básicas sobre modelagem comportamental para a equipe de coleta de requisitos.)

Pessoa de marketing: Isso parece um pouco técnico. Não estou certo de poder ajudar nisso.

Facilitador: Lógico que você pode. Que comportamento você observa do ponto de vista do usuário?

Pessoa de marketing: Hmm, bem, o sistema vai estar *monitorando* os sensores. *Lendo comandos* do proprietário. *Mostrando* o seu estado.

Facilitador: Viu, você pode fazer.

Jamie: Ele vai também *observar* o PC para determinar se há qualquer entrada vinda dele, por exemplo, acesso baseado na Internet ou informação de configuração.

Vinod: Sim, de fato, *configurar o sistema* é um estado em si.

Doug: Vocês estão enrolando. Vamos dar a isso um pouco mais de raciocínio... Há um modo de diagramar isso?

Facilitador: Há, mas vamos deixar isso para depois da reunião.

Elementos orientados a fluxo. A informação é transformada à medida que ela flui por um sistema baseado em computador. O sistema aceita entrada de diversas formas; aplica funções para transformá-la e produz saída de diversas formas. A entrada pode ser um sinal de controle transmitido por um transdutor, uma série de números digitados por um operador humano, um pacote de informações transmitido em uma ligação de rede ou um arquivo de dados volumoso recuperado de armazenamento secundário. A(s) transformação(ões) pode(m) compreender uma única comparação lógica, um algoritmo numérico complexo ou uma abordagem de inferência de regras de um sistema especialista. A saída pode acender um único LED ou produzir um relatório de 200 páginas. Com efeito, podemos criar um *modelo de fluxo* para qualquer sistema baseado em computador, independentemente do tamanho e da complexidade. Uma discussão mais detalhada sobre a modelagem de fluxo é apresentada no Capítulo 8.

7.6.2 Padrões de Análise

Qualquer pessoa que tenha feito engenharia de requisitos de mais do que uns poucos projetos de software começa a notar que certas coisas se repetem em todos os projetos de um domínio de aplicação específico.[15] Essas repetições podem ser chamadas de *padrões de análise* [FOW97] e representam algo (por exemplo, uma classe, função ou comportamento) dentro do domínio de aplicação que pode ser reusado quando se modela muitas aplicações.

Geyer-Schulz e Hahsler [GEY01] sugerem dois benefícios que podem ser associados ao uso de padrões de análise:

> Primeiro, padrões de análise aceleram o desenvolvimento de modelos de análise abstratos que captam os requisitos principais do problema concreto fornecendo modelos de análise reusáveis com exemplos bem como uma descrição das vantagens e limitações. Segundo, os padrões de análise facilitam a transformação do modelo de análise em um modelo de projeto sugerindo padrões de projeto e soluções confiáveis para problemas comuns.

Veja na Web

Uma biblioteca útil de padrões e muitos outros recursos de padrões podem ser encontrados em hillside.net/patterns.

Padrões de análise são integrados ao modelo de análise referenciando o nome do padrão. Eles também são armazenados em um repositório de modo que os engenheiros de requisitos possam usar facilidades de busca para encontrá-los e reusá-los.

Informações sobre um padrão de análise são apresentadas em um gabarito padrão que tem a forma [GEY01]:[16]

? Existe um gabarito recomendado para descrever padrões?

Nome do padrão: Um descritor que capta a essência do padrão. O descritor é usado no modelo de análise quando é feita referência ao padrão.

Intenção: Descreve o que o padrão realiza ou representa e/ou que problema ele trata dentro do contexto de um domínio de aplicação.

Motivação: Um cenário que ilustra como o padrão pode ser usado para tratar o problema.

15 Em alguns casos, coisas se repetem independentemente do domínio de aplicação. Por exemplo, as características e funções das interfaces do usuário são comuns e independem do domínio de aplicação em consideração.

16 Uma variedade de gabaritos de padrões tem sido proposta na literatura. Leitores interessados devem ver [FOW97], [BSU96] e [GAM95], entre muitas outras fontes.

Document shared on www.docsity.com

Forças e contexto: Uma descrição de tópicos externos (forças) que podem afetar como o padrão é usado e também os tópicos externos que serão resolvidos quando o padrão for aplicado. Tópicos externos podem incluir assuntos relacionados a negócios, restrições técnicas externas e assuntos relacionados a pessoas.

Solução: Uma descrição de como o padrão é aplicado para resolver o problema com ênfase nos tópicos estrutural e comportamental.

Conseqüências: Refere-se ao que acontece quando o padrão é aplicado e que compromissos existem durante sua aplicação.

Projeto: Discute como se pode chegar ao padrão de análise por meio do uso de padrões de projeto conhecidos.

Usos conhecidos: Exemplos de usos em sistemas reais.

Padrões relacionados: Um ou mais padrões de análise que são relacionados ao padrão em pauta, porque o padrão de análise (1) é comumente usado com o padrão em pauta, (2) é estruturalmente semelhante ao padrão em pauta, (3) é uma variante do padrão em pauta.

Exemplos de padrões de análise e discussão adicional sobre esse tópico são apresentados no Capítulo 8.

INFO

Padrões

Vemos padrões em praticamente tudo o que encontramos no dia-a-dia.

Considere filmes de ação-aventura — mais especificamente de ação-aventura de detetive com conotação cômica. Podemos definir padrões para *Herói&Amigo*, *CapitãoQueControlaHerói*, *BandidoComCoração* e muitos outros.

Por exemplo, o *CapitãoQueControlaHerói* é invariavelmente o mais velho, usa gravata (o herói não usa), grita com o *Herói&Amigo* constantemente, usualmente fornece alívio cômico ou pode ser usado em um papel mais malevolente para colocar bloqueios burocráticos ou de interesse próprio no caminho do *Herói&Amigo*. Um padrão dramático está sendo estabelecido.

Como um exemplo um pouco mais técnico, considere um telefone móvel. Os seguintes padrões são óbvios: *FazerChamada*, *ProcurarNúmero* e *ObterMensagens*, entre outros. Cada um desses padrões pode ser descrito uma vez e depois reusado no software para qualquer telefone móvel.

7.7 Negociação de Requisitos

Em um contexto ideal de engenharia de requisitos, as tarefas de concepção, levantamento e elaboração determinam os requisitos do cliente em detalhes suficientes para prosseguir em direção aos passos subseqüentes da engenharia de software. Infelizmente, isso raramente acontece. Na realidade, o cliente e o desenvolvedor entram em um processo de *negociação,* em que o cliente pode ser solicitado a ponderar a funcionalidade, o desempenho e outras características do produto ou sistema em face do custo e do prazo para chegar ao mercado. A intenção dessa negociação é desenvolver um plano de projeto que satisfaça às necessidades do cliente, ao mesmo tempo em que reflete as restrições do mundo real (por exemplo, prazo, pessoal, orçamento) propostas para a equipe de software.

> "Conciliação é a arte de dividir um bolo de tal modo que todos acreditam ter o maior pedaço."
>
> Ludwing Erhard

Veja na Web

Um breve artigo sobre a negociação de requisitos de software pode ser encontrado em sunset.usc.edu/~negyed/publications/Software_Requirements_Negotiation-Some_lessons_Learned.html.

As melhores negociações buscam um resultado "ganha-ganha"[17], isto é, o cliente ganha por obter o sistema ou produto que satisfaça à maioria das necessidades dos clientes, e a equipe de software ganha por trabalhar com orçamentos e prazos realistas e realizáveis.

Boehm [BOE98] define um conjunto de atividades de negociação no começo de cada iteração do processo de software. Em vez de uma simples atividade de comunicação com o cliente, as seguintes atividades são definidas:

1. Identificação dos interessados-chave do sistema ou subsistema.
2. Determinação das "condições de ganho" dos interessados.
3. Negociação das condições de ganho dos interessados para reconciliá-las em um conjunto de condições ganha-ganha para todos os envolvidos (inclusive a equipe de software).

A conclusão bem-sucedida desses passos iniciais atinge um resultado "ganha-ganha", o qual se torna o critério-chave para prosseguir para as atividades subseqüentes de engenharia de software.

INFO

A Arte da Negociação

Aprender como negociar efetivamente pode ser bem útil para toda a sua vida pessoal e técnica. Vale a pena considerar as seguintes diretrizes:

1. *Reconheça que não é uma competição.* Para ter sucesso, ambas as partes devem ter a sensação de que venceram ou conseguiram alguma coisa. Ambas terão de ceder.
2. *Trace uma estratégia.* Decida o que você gostaria de conseguir; o que a outra parte quer conseguir, e como você fará para que ambos ocorram.
3. *Ouça atentamente.* Não trabalhe na formulação da sua resposta enquanto a outra parte estiver falando. Ouça. É provável que você obtenha conhecimento que irá ajudá-lo a melhor negociar a sua posição.
4. *Focalize os interesses das outras partes.* Não assuma posições radicais se deseja evitar conflitos.
5. *Não deixe a coisa ficar pessoal.* Enfoque o problema que precisa ser resolvido.
6. *Seja criativo.* Não tenha medo de pensar grande se estiver em um impasse.
7. *Esteja pronto a se comprometer.* Uma vez chegado a um acordo, não enrole: comprometa-se com ele e siga adiante.

CASASEGURA

O Início de uma Negociação

A cena: O escritório de Lisa Perez, depois da primeira reunião de coleta de requisitos.

Os personagens: Doug Miller, gerente de engenharia de software e Lisa Perez, gerente de marketing.

A conversa:

Lisa: Então, eu soube que a primeira reunião foi realmente boa.

Doug: Realmente, foi. Você enviou um bom pessoal para a reunião... eles realmente contribuíram.

Lisa (sorrindo): É, eles na verdade me contaram que conseguiram e que isso não foi uma atividade exaustiva.

Doug (rindo): Eu cuidarei de pôr de lado a minha tendência técnica na próxima visita... Olhe, Lisa, eu acho que nós podemos ter um problema com a obtenção de toda a funcionalidade para a função de segurança residencial nas datas que sua gerência está falando. Eu sei que ainda é cedo, mas estive fazendo um pouco do rascunho de planejamento e...

Lisa: Nós temos que ter isso naquela data, Doug. De qual funcionalidade você está falando?

Doug: Eu imagino que nós possamos obter toda a funcionalidade de segurança residencial nessa data final, mas nós temos de adiar o acesso pela Internet até a segunda versão.

17 Dúzias de livros têm sido escritos sobre habilidades de negociação (por exemplo, [LEW00], [FAR97], [DON96]). Essa é uma das coisas mais importantes que um engenheiro ou gerente de software novo (ou velho) pode aprender. Leia alguns deles.

Document shared on www.docsity.com

Lisa: Doug, é o acesso pela Internet que dá o apelo mágico ao *CasaSegura*. Nós vamos construir toda a campanha de marketing em torno disso. Nós temos de ter isso.

Doug: Eu entendo a sua situação, de fato entendo. O problema é que para lhe dar o acesso pela Internet vamos precisar de um site totalmente seguro, pronto e funcionando. Isso leva tempo e pessoal. Nós também teremos de construir um monte de funcionalidades adicionais na primeira versão... Eu não acho que nós possamos conseguir isso com os recursos que temos.

Lisa (franzindo a testa): Está bem, mas você tem de arranjar um jeito de conseguir. É essencial para as funções de segurança residencial e para as outras funções também ... as outras funções podem esperar até a versão seguinte... eu concordo com isso.

Lisa e Doug parecem estar em um impasse, no entanto, eles precisam negociar uma solução para esse problema. Eles podem, ambos, "ganhar" com isso? Desempenhando o papel de mediador, o que você sugeriria?

7.8 Validação de Requisitos

? Quando reviso requisitos, quais questões devo fazer?

À medida que cada elemento do modelo de análise é criado, ele é examinado quanto a consistência, omissões e ambigüidade. Os requisitos representados pelo modelo são priorizados pelo cliente e agrupados em pacotes de requisitos que serão implementados como incrementos de software e entregues ao cliente. Uma revisão do modelo de análise trata das seguintes questões:

- Cada requisito está consistente com o objetivo global do sistema/produto?
- Todos os requisitos foram especificados no nível de abstração adequado? Isto é, algum requisito fornece um nível de detalhe técnico inadequado neste estágio?
- O requisito é realmente necessário ou representa uma característica adicional que pode não ser essencial para o objetivo do sistema?
- Cada requisito é limitado e não ambíguo?
- Cada requisito tem atribuição? Isto é, uma fonte (geralmente um indivíduo específico) está associada a cada requisito?
- Algum requisito conflita com outros requisitos?
- Cada requisito é realizável no ambiente técnico que vai alojar o sistema ou o produto?
- Cada requisito pode ser testado quando estiver implementado?
- O modelo de requisitos reflete adequadamente a informação, a função e o comportamento do sistema a ser construído?
- O modelo de requisitos foi "particionado" de modo a expor cada vez mais informações detalhadas sobre o sistema?
- Padrões de requisitos foram usados para simplificar o modelo de requisitos? Todos os padrões foram adequadamente validados? Todos os padrões são consistentes com os requisitos do cliente?

Essas e outras questões devem ser formuladas e respondidas para garantir que o modelo de requisitos reflita adequadamente as necessidades dos clientes e forneça uma base sólida para o projeto.

7.9 Resumo

É necessário entender os requisitos antes que o projeto e a construção de um sistema baseado em computador possam começar. Para conseguir isso, um conjunto de tarefas de engenharia de requisitos é conduzido. A engenharia de requisitos ocorre durante as atividades de comunicação com o cliente e modelagem que definimos para o processo genérico de software. Sete funções distintas

de engenharia de requisitos — concepção, levantamento, elaboração, negociação, especificação, validação e gestão — são conduzidas pelos membros da equipe de software.

Na concepção do projeto, o desenvolvedor e o cliente (bem como os outros interessados) estabelecem os requisitos básicos do problema, definem as restrições que afetam o projeto e tratam das principais características e funções que devem estar presentes para que o sistema alcance os seus objetivos. Essa informação é refinada e expandida durante o levantamento — uma atividade de coleta de requisitos que faz uso de reuniões facilitadas, IFQ e desenvolvimento de cenários de usuário.

A elaboração expande ainda mais os requisitos em um modelo de análise — uma coleção de elementos do modelo baseados em cenário, atividade, classe, comportamentais e orientados ao fluxo. Uma variedade de notações de modelagem pode ser usada para criar esses elementos. O modelo pode referenciar padrões de análise — características do domínio do problema que têm sido recorrentes ao longo de diferentes aplicações.

À medida que os requisitos são identificados e o modelo de análise é criado, a equipe de software e os outros interessados no projeto negociam a prioridade, a disponibilidade e o custo relativos a cada requisito. O objetivo dessa negociação é desenvolver um plano de projeto realístico. Além disso, cada um dos requisitos e todo o modelo de análise são validados em face da necessidade de garantir que o sistema correto seja construído.

Referências Bibliográficas

[BOE98] Boehm, B., e Egyed, A., "Software Requirements Negotiation: Some Lessons Learned", *Proc. Intl. Conf. Software Engineering*, ACM/IEEE, 1998, p. 503-06.

[BOS91] Bossert, J. L., *Quality Function Deployment: A Practitioner's Approach*, ASQC Press, 1991.

[BUS96] Buschmann, F., *et al.*, *Pattern-Oriented Software Architecture: A System of Pattern*, Wiley, 1996.

[COC01] Cockburn, A., *Writing Efective Use-Cases*, Addison-Wesley, 2001.

[CRI92] Christel, M. G. e Kang, K.C., "Issues in Requirements Elicitation", Software Engineering Institute, CMU/SEI-92-TR-12 7, set. 1992.

[DON96] Donaldson, M. C. e Donaldson, M., *Negotiating for Dummies*, IDG Books Worldwide, 1996.

[FAR97] Farber, D. C., *Common Sense Negotiation: The Art of Winning Gracefully*, Bay Press, 1997.

[FOW97] Fowler, M., *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1997.

[GAM95] Gamma, E., *et al.*, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[GAU89] Gause, D. C. e Weinberg, G. M., *Exploring Requirements: Quality Before Design*, Dorset House, 1989.

[GEY01] Geyer-Schulz, A., e Hahsler, M., *Software Engineering with Analysis Patterns*, Technical Report 01/2001, Institui für Informationsverarbeitung und-wirtschaft, Wirschaftsuniversität Wien, nov. 2001, baixado de http://wwwai.wu-wien.ac.at/~hahsler/research/virlib_working2001/virlib/.

[JAC92] Jacobson, I., *Object-Oriented Software Engineering*, Addison-Wesley, 1992.

[LEW00] Lewicki, R., Saunders, D. e Minton, J., *Essentials of Negotiation*, McGraw-Hill, 2000.

[PAR96] Pardee, W., *To Satisfy and Delight Your Customer*, Dorset House, 1996.

[SOM97] Somerville, I. e Sawyer, P., *Requirements Engineering*, Wiley, 1997.

[THA97] Thayer, R. H., e Dorfman, M., *Software Requirements Engineering*, 2. ed., IEEE Computer Society Press, 1997.

[YOU01] Young, R., *Effective Requirements Practices*, Addison-Wesley, 2001.

[ZAH90] Zahniser, R. A., "Building Software in Groups", *American Programmer*, v. 3, n. 7-8, jul./ago. 1990.

[ZUL92] Zultner, R., "Quality Function Deployment for Software: Satisfying Customers", *American Programmer*, fev. 1992, p. 28-41.

Problemas e Pontos a Considerar

7.1. Por que muitos desenvolvedores de software não dão atenção suficiente para a engenharia de requisitos? Há circunstâncias em que você pode desprezá-la?

7.2. O que a "análise de viabilidade" implica quando é discutida no contexto da função de concepção?

7.3. A você foi atribuída a responsabilidade de fazer o levantamento de requisitos de um cliente que lhe diz estar muito ocupado para encontrá-lo. O que você deveria fazer?

Document shared on www.docsity.com

7.4. Discuta alguns dos problemas que ocorrem quando os requisitos devem ser levantados de três ou quatro clientes diferentes.

7.5. Por que dizemos que o modelo de análise representa um instantâneo de um sistema no tempo?

7.6. Vamos assumir que você convenceu o cliente (você é um vendedor muito bom) a concordar com todas as solicitações que fez como desenvolvedor. Isso faz de você um mestre negociador? Por quê?

7.7. Desenvolva no mínimo três "questões livres de contexto" que você poderia formular a um interessado durante a concepção.

7.8. Ao longo deste capítulo nos referimos ao "cliente". Descreva o "cliente" de desenvolvedores de sistemas de informação, de construtores de produtos baseados em computador, de construtores de sistemas. Tome cuidado: pode haver mais neste problema do que você imagina.

7.9. Desenvolva um "kit" de coleta facilitada de requisitos. O kit deve incluir um conjunto de diretrizes para conduzir uma reunião de coleta de requisitos e materiais que podem ser usados para facilitar a criação de listas e quaisquer outros itens que possam ajudar na definição de requisitos.

7.10. Seu instrutor dividirá a classe em grupos de quatro ou seis estudantes. Metade do grupo vai desempenhar o papel do departamento de marketing e a outra metade vai desempenhar o papel do engenheiro de software. Seu trabalho é definir requisitos para a função de segurança descrita para o *CasaSegura* neste capítulo. Conduza uma reunião de coleta de requisitos usando as diretrizes apresentadas neste capítulo.

7.11. Desenvolva um caso de uso completo para uma das seguintes atividades:

a. Fazer uma retirada em um caixa eletrônico.

b. Usar seu cartão de crédito para uma refeição em um restaurante.

c. Comprar uma ação usando uma conta de corretagem *on-line*.

d. Pesquisar livros (de um tópico específico) usando uma livraria *on-line*.

e. Uma atividade especificada pelo seu instrutor.

7.12. O que as "exceções" de casos de uso representam?

7.13. Discuta brevemente cada um dos elementos de um modelo de análise. Indique o que cada um contribui para o modelo, como cada um é singular, e qual informação geral é apresentada por cada um.

7.14. Descreva um padrão de análise com as suas próprias palavras.

7.15. Usando o gabarito apresentado na Seção 7.6.2, sugira um ou mais padrões de análise para uma aplicação sugerida pelo seu instrutor.

7.16. O que "ganha-ganha" significa no contexto de negociação durante a atividade de engenharia de requisitos?

7.17. O que você acha que acontece quando a validação de requisitos descobre um erro? Quem está envolvido na correção do erro?

Leituras e Fontes de Informação Adicionais

Por ser essencial para a criação com sucesso de qualquer sistema complexo baseado em computador, a engenharia de requisitos é discutida em um amplo conjunto de livros. Hull e seus colegas (*Requirements Engineering*, Springer-Verlag, 2002), Bray (*An Introduction to Requirements Engineering*, Addison-Wesley, 2002), Arlow (*Requirements Engineering*, Addison-Wesley, 2001), Gilb (*Requirements Engineering*, Addison-Wesley, 2000), Graham (*Requirements Engineering and Rapid Development*, Addison-Wesley, 1999) e Sommerville e Kotonya (*Requirement Engineering: Processes and Techniques*, Wiley,1998) são alguns dos muitos livros dedicados ao assunto. Dan Berry (http://se.uwarteloo.ca/~dberry/bib.html) tem publicado uma ampla variedade de artigos com pensamentos provocativos sobre tópicos de engenharia de requisitos.

Lauesen (*Software Requirements: Styles and Techniques*, Addison-Wesley, 2002) apresenta um levantamento abrangente de métodos e notações de análise de requisitos. Weigers (*Software Requirements*, Microsoft Press, 1999) e Leffingwell e seus colegas (*Managing Software Requirements: A Unified Approach*, Addison-Wesley, 2000) apresentam uma coleção útil das melhores práticas de requisitos e sugerem diretrizes pragmáticas para a maioria dos tópicos do processo de engenharia de requisitos.

Robertson e Robertson (*Mastering the Requirements Process*, Addison-Wesley, 1999) apresentam um estudo de caso muito detalhado que ajuda a explicar todos os tópicos da análise de requisitos de software e o modelo de análise. Kovitz (*Practical Software Requirements: A Manual of Content and Style*, Manning Publications, 1998) discute uma abordagem passo-a-passo para a análise de requisitos e um guia de estilo para aqueles que precisam desenvolver especificações de requisitos. Jackson (*Software Requirements Analysis and Specification: A Lexicon of Practices, Principles and Prejudices*, Addison-Wesley, 1995) apresenta uma visão interessante do assunto de A a Z (literalmente). Ploesch (*Assertions, Scenarios and Prototypes*, Springer-Verlag, 2003) discute técnicas avançadas para o desenvolvimento de requisitos de software.

Windle e Abreo (*Software Requirements Using the Unified Process*, Prentice-Hall, 2002) discutem a engenharia de requisitos no contexto do Processo Unificado e da notação UML. Alexander e Steven (*Writing Better Requirements*, Addison-Wesley, 2002) apresentam um conjunto resumido de diretrizes para escrever requisitos claramente, representando-os como cenários e revisando o resultado final.

A modelagem de casos de uso é freqüentemente o motor da criação de todos os outros aspectos do modelo de análise. O assunto é discutido em profundidade por Bittner e Spence (*Use-Case Modeling*, Addison-Wesley, 2002), Cockburn [COC01], Armour e Miler (*Advanced Use-Case Modeling: Software Systems*, Addison-Wesley, 2000), Kulak e seus colegas (*Use Cases: Requirements in Context*, Addison-Wesley, 2000) e Schneider e Winters (*Applying Use Cases*, Addison-Wesley, 1998).

Uma grande variedade de fontes de informação sobre engenharia e análise de requisitos está disponível na Internet. Uma lista atualizada de referências da World Wide Web que são relevantes para análise e engenharia de requisitos pode ser encontrada no site deste livro: **http://www.mhhe.com/pressman.**

Document shared on www.docsity.com

CAPÍTULO

MODELAGEM DE ANÁLISE

CONCEITOS-CHAVE

análise de domínio 147
análise estruturada 147
classes..... 165
diagramas de fluxo de dados..... 160
modelagem de análise
baseada em classes .. 165
baseada em cenário .. 153
comportamental 177
CRC..... 171
dados 148
espec de controle 163
orientada a fluxo ... 159
objetos de dados..... 148
regras práticas 146

No nível técnico, a engenharia de software começa com uma série de tarefas de modelagem que levam à especificação completa dos requisitos e à representação abrangente do projeto para o software a ser construído. O *modelo de análise,* na verdade um conjunto de modelos, é a primeira representação técnica do sistema.

Em um livro pioneiro sobre o assunto, Tom DeMarco [DEM79] descreve a análise estruturada do seguinte modo:

> Revendo os problemas reconhecidos e as falhas da fase de análise, sugiro que façamos as seguintes adições ao nosso conjunto de metas para a fase de análise:
>
> - Os produtos da análise devem ser altamente manuteníveis. Isso se aplica particularmente ao documento-alvo [especificações de requisitos do software].
> - Problemas de tamanho devem ser enfrentados usando um método efetivo de particionamento. A especificação da novela vitoriana está fora.
> - Gráficos precisam ser usados sempre que possível.
> - Precisamos diferenciar as considerações lógicas [essenciais] e físicas [de implementação]...
>
> No mínimo, precisamos de...
>
> - Algo que nos ajude a particionar nossos requisitos e documentar esse particionamento antes da especificação...
> - Algum modo de fazer o rastreio e avaliar as interfaces...
> - Novas ferramentas para descrever lógica e política, algo melhor do que texto narrativo.

Embora DeMarco tenha escrito sobre os atributos da modelagem de análise há mais de um quarto de século, seus comentários ainda se aplicam aos métodos e notação modernos de modelagem de análise.

PANORAMA

O que é? A palavra escrita é um veículo magnífico de comunicação, mas não é necessariamente o melhor modo de representar os requisitos de software para computador. A modelagem da análise usa uma combinação de formas textuais e diagramáticas para mostrar os requisitos de dados, função e comportamento, de modo que seja relativamente fácil de entender e, mais importante, mais direto de revisar quanto à correção, completeza e consistência.

Quem faz? Um engenheiro de software (algumas vezes chamado de analista) constrói um modelo usando requisitos extraídos do cliente.

Por que é importante? Para validar requisitos de software, você precisa examiná-los de diferentes pontos de vista. A modelagem da análise representa os requisitos em várias "dimensões", aumentando conseqüentemente a probabilidade de que sejam encontrados erros, apareçam inconsistências e que omissões sejam descobertas.

Quais são os passos? Os requisitos informacionais, funcionais e comportamentais são modelados usando vários formatos diagramáticos diferentes. A modelagem baseada em cenário representa o sistema sob o ponto de vista do usuário. A modelagem orientada a fluxo fornece indicação de como os objetos de dados são transformados pelas funções de processamento. A modelagem baseada em classes define objetos, atributos e relacionamentos. A modelagem comportamental mostra os estados do sistema e de suas classes, e o impacto que os eventos têm nesses estados. Uma vez criados os modelos preliminares, eles são refinados e analisados para avaliar sua clareza, completeza e consistência. O modelo final de análise é então validado por todos os interessados.

Qual é o produto do trabalho? Uma grande variedade de formas diagramáticas podem ser escolhidas para o modelo de análise. Cada uma dessas representações fornece uma visão de um ou mais elementos do modelo.

Como tenho certeza de que fiz corretamente? Os produtos do trabalho da modelagem da análise devem ser revisados quanto à correção, completeza e consistência. Eles precisam refletir as necessidades de todos os interessados e estabelecer um fundamento com base na qual o projeto pode ser conduzido.

8.1 ANÁLISE DE REQUISITOS

PONTO CHAVE

O modelo de análise e especificação de requisitos fornece meios para avaliar a qualidade quando o software é construído.

Análise de requisitos resulta na especificação das características operacionais do software; indica a interface do software com outros elementos do sistema e estabelece restrições a que o software deve satisfazer. A análise de requisitos permite ao engenheiro de software (algumas vezes chamado de *analista* ou *modelador* nesse papel) elaborar requisitos básicos do software estabelecidos durante tarefas anteriores de engenharia de requisitos e construir modelos que descrevam cenários de usuário, atividades funcionais, classes de problemas e seus relacionamentos, comportamento do sistema e das classes, e fluxo dos dados à medida que são transformados.

A análise de requisitos fornece ao projetista de software uma representação da informação, função e comportamento, que podem ser traduzidos para os projetos arquitetural, de interfaces e em nível de componentes. Por fim, o modelo de análise e a especificação de requisitos propiciam ao desenvolvedor e ao cliente os meios de avaliar a qualidade quando o software é construído.

Ao longo da modelagem de análise, o principal foco do engenheiro de software é em *o que*, e não em *como*. Que objetos o sistema manipula, que funções ele deve executar, que comportamentos exibe, que interfaces são definidas e que restrições se aplicam?[1]

Em capítulos anteriores, notamos que a especificação completa dos requisitos pode não ser possível nesse estágio. O cliente pode não estar seguro do que precisamente é necessário. O desenvolvedor pode não estar seguro de que uma abordagem específica vai realizar apropriadamente funções e desempenho. Essas realidades depõem a favor de uma abordagem iterativa para análise e modelagem de requisitos. O analista deve modelar o que é conhecido e usar tais modelos como base para o projeto do incremento de software.[2]

8.1.1 Objetivos Gerais e Filosofia

O modelo de análise deve atingir três objetivos principais: (1) descrever o que o cliente exige; (2) estabelecer a base para a criação de um projeto de software; e (3) definir um conjunto de requisitos que possam ser validados quando o software for construído. O modelo de análise vence o espaço entre uma descrição em nível de sistema (veja o Capítulo 6), que descreve a funcionalidade global do sistema como ela é conseguida pela aplicação de software, hardware, dados, pessoas e outros elementos do sistema; e um projeto de software (Capítulo 9), que descreve a arquitetura, interface do usuário e estrutura em nível de componente da aplicação de software. Esse relacionamento é apresentado na Figura 8.1.

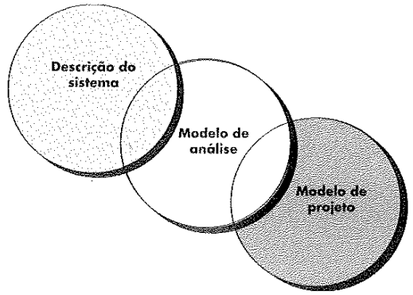
1 Deve-se notar que, à medida que os clientes se tornam mais sofisticados tecnologicamente, há uma tendência de especificação do *como* junto com *o que*. No entanto, o principal foco deve permanecer no *o que*.

2 Como alternativa, a equipe de software pode preferir criar um protótipo (veja o Capítulo 3) em um esforço de melhorar a compreensão sobre os requisitos do sistema.

Document shared on www.docsity.com

FIGURA 8.1

O modelo de análise como uma ponte entre a descrição do sistema e o modelo de projeto



> "Problemas que merecem ser resolvidos com energia mostram seu mérito respondendo com a mesma energia."
>
> **Piet Hein**

É importante notar que alguns dos elementos do modelo de análise estão presentes (em um nível mais alto de abstração) na descrição do sistema, e que as tarefas de engenharia de requisitos realmente começam como parte da engenharia de sistemas. Além disso, todos os elementos do modelo de análise são diretamente rastreáveis para partes do modelo de projeto. Nem sempre é possível uma clara divisão das tarefas de análise e projeto entre essas duas importantes atividades de modelagem. Algum projeto invariavelmente ocorre como parte da análise e alguma análise será conduzida durante o projeto.

8.1.2 Regras Práticas de Análise

Arlow e Neustadt [ARL02] sugerem um número de regras práticas valiosas que deveriam ser seguidas quando da criação de um modelo de análise:

- *O modelo deve focalizar os requisitos que são visíveis no problema ou domínio do negócio. O nível de abstração deve ser relativamente alto.* "Não se atole em detalhes" [ARL02] que tentam explicar como o sistema funcionará.
- *Cada elemento do modelo de análise deve contribuir para um entendimento global dos requisitos do software e fornecer uma visão aprofundada do domínio da informação, função e comportamento do sistema.*
- *Adie a consideração de modelos de infra-estrutura e outros não funcionais até o projeto.* Por exemplo, um banco de dados pode ser necessário, mas as classes necessárias para implementá-lo, as funções exigidas para dar acesso a ele e o comportamento que será exibido à medida que é usado devem ser considerados somente depois de a análise do domínio do problema ter sido completada.
- *Minimize o acoplamento ao longo de todo o sistema.* É importante representar os relacionamentos entre classes e funções. No entanto, se o nível de "interconexão" é extremamente alto, esforços devem ser feitos para reduzi-lo.
- *Certifique-se de que o modelo de análise tem valor para todos os interessados.* Cada clientela tem o seu próprio uso para o modelo. Por exemplo, interessados no negócio devem usar o modelo para validar os requisitos; projetistas devem usar o modelo como base para o projeto. O pessoal de Garantia de Qualidade (QA) deve usar o modelo para ajudar a planejar os testes de aceitação.
- *Mantenha o modelo tão simples quanto puder.* Não acrescente diagramas quando eles não fornecerem informação nova. Não use formulários notacionais completos, quando uma lista simples servir.

Veja na Web

Muitos recursos úteis para a análise de domínio podem ser encontrados em www.iturls.com/English/SoftwareEngineering/SE_mod5.asp.

8.1.3 Análise de Domínio

Em nossa discussão de engenharia de requisitos (veja o Capítulo 7), observamos que padrões de análise freqüentemente se repetem ao longo de muitas aplicações em um domínio específico de negócios. Se esses padrões são definidos e categorizados de modo que permitam a um engenheiro de software ou analista reconhecê-lo e reutilizá-los, a criação do modelo de análise é acelerada. Mais importante, a probabilidade de aplicar padrões de projeto e componentes executáveis de software reutilizáveis cresce consideravelmente. Isso melhora o prazo de colocação no mercado e reduz os custos de desenvolvimento.

Contudo, como os padrões de análise são inicialmente reconhecidos? Quem os define, os categoriza e os apronta para uso em projetos subseqüentes? As respostas a essas questões estão na *análise de domínio*. Firesmith [FIR93] a descreve do seguinte modo:

> A análise de domínio de software é a identificação, análise e especificação dos requisitos comuns de um domínio de aplicação específico, tipicamente para reuso em vários projetos dentro daquele domínio de aplicação... [Análise de domínio orientada a objetos é] a identificação, análise e especificação de capacidades comuns, reusáveis dentro de um domínio de aplicação específico, em termos de objetos, classes, submontagens e arcabouço comuns.

O "domínio de aplicação específico" pode ir de aviônica a serviços bancários, de videogames multimídia a software embutido em dispositivos médicos. A meta da análise de domínio é direta: encontrar ou criar as classes de análise e/ou as funções e características que são amplamente aplicáveis, de modo que possam ser reutilizadas.[3]

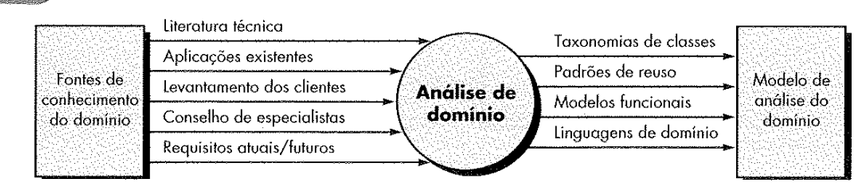
> "A grande arte de aprender é entender um pouco de cada vez."
>
> **John Locke**

Veja na Web

Uma discussão interessante sobre engenharia e análise de domínio pode ser encontrada em www.sei.cmu.edu/str/descriptions/deda.html.

De certo modo, o papel de um analista de domínio é semelhante ao papel de um mestre ferramenteiro em um ambiente de intensa fabricação. O trabalho do ferramenteiro é projetar e construir ferramentas que podem ser usadas por várias pessoas, fazendo trabalho similar, mas não necessariamente o mesmo. O papel do analista de domínio[4] é descobrir e definir padrões de análise reutilizáveis, classes de análise e informação relacionada que possam ser usadas por várias pessoas trabalhando em aplicações semelhantes, mas não necessariamente as mesmas. A Figura 8.2 [ARA89] ilustra entradas e saídas-chave para o processo de análise de domínio. Fontes de conhecimento do domínio são consultadas em uma tentativa de identificar objetos que podem ser reusados ao longo do domínio.

FIGURA 8.2 Entradas e saídas da análise de domínio



8.2 ABORDAGENS DE MODELAGEM DE ANÁLISE

Uma visão de modelagem de análise, chamada de *análise estruturada*, considera os dados e os processos que transformam os dados entidades separadas. Objetos de dados são modelados para

3 Uma visão complementar de análise de domínio "envolve a modelagem do domínio para que os engenheiros de software e outros interessados possam melhor aprender sobre ele... nem todas as classes do domínio necessariamente resultam no desenvolvimento de classes reusáveis" [LET03].

4 O fato de um analista de domínio estar trabalhando não implica que um engenheiro de software não precise entender o domínio da aplicação. Todos os membros de uma equipe de software devem ter algum conhecimento do domínio em que o software será colocado.

Document shared on www.docsity.com

que definam seus atributos e relacionamentos. Processos que manipulam objetos de dados são modelados para que mostrem como eles transformam os dados à medida que os objetos de dados fluem pelo sistema.

Uma segunda abordagem para modelagem de análise, chamada de *análise orientada a objetos*, focaliza a definição de classes e o modo pelo qual elas colaboram umas com as outras para atender aos requisitos do cliente. UML e Processo Unificado (veja o Capítulo 3) são predominantemente orientados a objetos.

> "[A]análise é frustrante, cheia de relacionamentos interpessoais complexos, indefinida e difícil. Em uma palavra, é fascinante. Uma vez que você seja fisgado, os antigos prazeres fáceis da construção de sistemas nunca mais serão suficientes para satisfazê-lo."
>
> **Tom DeMarco**

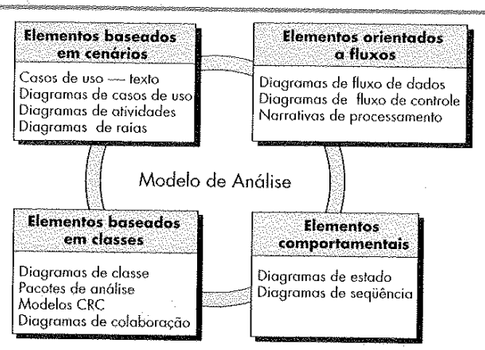
Apesar de o modelo de análise que propomos neste livro combinar características de ambas as abordagens, equipes de software freqüentemente escolhem uma e excluem todas as representações da outra. A questão não é qual a melhor, mas que combinação de representações fornecerá aos interessados o melhor modelo de requisitos de software e a conexão mais efetiva para o projeto de software.

Modelagem de análise conduz à derivação de cada um dos elementos de modelagem, mostrados na Figura 8.3. No entanto, o conteúdo específico de cada elemento (isto é, os diagramas usados para construir o elemento e o modelo) pode diferir de projeto a projeto. Como mencionamos várias vezes neste livro, a equipe de software deve trabalhar para mantê-los simples. Somente aqueles elementos de modelagem que adicionam valor ao modelo devem ser usados.

> "Por que devemos construir modelos? Por que não construirmos apenas o sistema propriamente dito? A resposta é que podemos construir modelos de tal modo que destaquem ou enfatizem certas características críticas de um sistema, enquanto simultaneamente retiram a ênfase de outras características."
>
> **Ed Yourdon**

FIGURA 8.3

Elementos do modelo de análise



8.3 CONCEITOS DE MODELAGEM DE DADOS

Veja na Web

Informação útil sobre modelagem de dados pode ser encontrada em www.datamodel.org.

Modelagem de análise freqüentemente começa com a *modelagem de dados*. O engenheiro de software ou analista define todos os objetos de dados processados no sistema, os relacionamentos entre os objetos de dados e outra informação pertinente aos relacionamentos.

8.3.1 Objetos de Dados

O *objeto de dados* é uma representação de quase toda informação composta que deve ser compreendida pelo software. Por *informação composta* queremos nos referir a algo que tem várias

? Como o objeto de dados se manifesta no contexto de uma aplicação?

PONTO CHAVE

O objeto de dados é uma representação de qualquer informação composta, processada pelo software.

PONTO CHAVE

Os atributos nomeiam um objeto de dados, descrevem suas características e, em alguns casos, fazem referência a outro objeto.

Veja na Web

O conceito chamado de "normalização" é importante para aqueles que pretendem fazer modelagem completa de dados. Uma introdução útil pode ser encontrada em www.datamodel.org.

propriedades ou atributos diferentes. Assim, "largura" (um simples valor) não seria um objeto de dados válido, mas **dimensões** (incorporando altura, largura e profundidade) poderia ser definido como objeto.

Um objeto de dados pode ser uma entidade externa (por exemplo, algo que produza ou consuma informação), uma coisa (por exemplo, um relatório ou um mostrador), uma ocorrência (por exemplo, uma chamada telefônica) ou um evento (por exemplo, um alarme), uma função (por exemplo, vendedor), uma unidade organizacional (por exemplo, departamento de contabilidade), um lugar (por exemplo, um depósito) ou uma estrutura (por exemplo, um arquivo). Uma pessoa ou um automóvel pode ser visto como objeto de dados, no sentido de que qualquer um deles pode ser definido em termos de conjunto de atributos. A descrição do objeto de dados incorpora o objeto de dados e todos os seus atributos.

Um objeto de dados encapsula apenas dados — não há referência dentro de um objeto de dados a operações que agem sobre os dados.[5] Ele pode ser representado por uma tabela, como mostra a Figura 8.4. Os títulos das colunas da tabela refletem os atributos do objeto. Neste caso, um automóvel é definido em termos de **marca**, **modelo**, **número da placa**, **tipo de carroceria**, **cor** e **proprietário**. O corpo da tabela representa instâncias específicas do objeto de dados. Por exemplo, Chevy Corvette é um exemplo do objeto de dados automóvel.

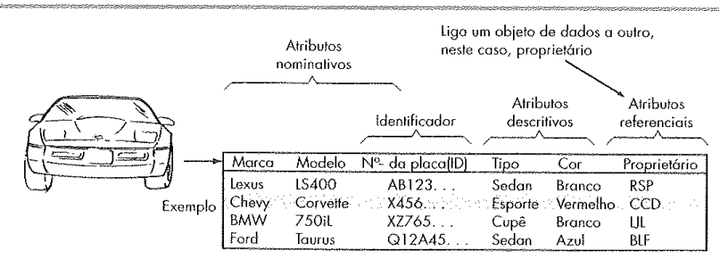
8.3.2 Atributos de Dados

Atributos de dados definem as propriedades de um objeto de dados e assumem uma de três diferentes características. Podem ser usados para (1) nomear um exemplo do objeto de dados, (2) descrever o exemplo ou (3) fazer referência a outro exemplo em outra tabela. Além disso, um ou mais dos atributos podem ser definidos como *identificador* — isto é, o atributo identificador torna-se uma "chave" quando desejamos encontrar um exemplo do objeto de dados. Em alguns casos, valores do(s) identificador(es) são únicos, apesar disso não ser exigido. Com referência ao objeto de dados **automóvel**, um identificador razoável poderia ser o número de identificação.

O conjunto de atributos adequado a certo objeto de dados é determinado entendendo-se o contexto do problema. Os atributos de **automóvel** poderiam servir bem para uma aplicação usada pelo Departamento de Veículos Automotores, mas eles seriam inúteis para uma empresa de automóveis que necessita de software para controle de fabricação. Nesse último caso, os atributos para automóvel poderiam também incluir **número de identificação**, **tipo de carroceria** e **cor**, mas muitos atributos adicionais (por exemplo, **código interno**, **tipo de conjunto de direção**, **indicador do pacote de opções do acabamento interno** e **tipo de transmissão**) teriam de ser adicionados para tornar **automóvel** um objeto significativo no contexto do controle de fabricação.

FIGURA 8.4

Representação tabular de objetos de dados



5 Essa distinção separa o objeto de dados da classe ou objeto definido como parte da abordagem orientada a objetos.

Document shared on www.docsity.com

INFO

Objetos de Dados e Classes OO — São a Mesma Coisa?

Uma questão comum ocorre quando objetos de dados são discutidos: objeto de dados é a mesma coisa que classe orientada a objetos? A resposta é não.

O objeto de dados define um item de dados composto; ele incorpora uma coleção de itens de dados individuais (atributos) e dá a ela um nome (o nome do objeto de dados). Uma classe OO encapsula atributos de dados, mas também incorpora as operações que manipulam os dados envolvidos por aqueles atributos. Além disso, a definição de classes implica uma ampla infra-estrutura que é parte da abordagem de engenharia de software orientada a objetos. Classes comunicam-se umas com as outras via mensagens; elas podem ser organizadas em hierarquias; e fornecem características de herança para os objetos que são uma instância da classe.

PONTO CHAVE

Relacionamentos indicam a maneira pela qual objetos de dados são "conectados" uns com os outros.

8.3.3 Relacionamentos

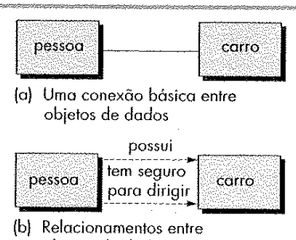
Objetos de dados são conectados uns com os outros de diferentes modos. Considere dois objetos de dados, **pessoa** e **carro**, que podem ser representados por meio da notação simples ilustrada na Figura 8.5a. Uma conexão é estabelecida entre **pessoa** e **carro** porque os dois objetos são relacionados. Mas quais são os relacionamentos? Para determinarmos a resposta, precisamos entender o papel de pessoas (proprietárias, neste caso) e carros, no contexto do software a ser construído. Podemos definir um conjunto de pares objeto/relacionamento que definem os relacionamentos relevantes. Por exemplo,

- Uma pessoa *possui* um carro.
- Uma pessoa *tem seguro para dirigir* um carro.

Os relacionamentos *possui* e *tem seguro para dirigir* definem as conexões relevantes entre **pessoa** e **carro**. A Figura 8.5b ilustra graficamente esses pares objeto/relacionamento. As setas da Figura 8.5b fornecem informação importante sobre a direcionalidade do relacionamento e freqüentemente reduzem ambigüidade ou má interpretação.

FIGURA 8.5

Relacionamentos entre objetos de dados



(a) Uma conexão básica entre objetos de dados

(b) Relacionamentos entre objetos de dados

8.3.4 Cardinalidade e Modalidade

Os elementos da modelagem de dados — objetos de dados, atributos e relacionamentos — fornecem a base para a compreensão do domínio de informação de um problema. No entanto, também deve ser entendida a informação adicional relacionada a esses elementos básicos.

Definimos um conjunto de objetos e representamos os pares objeto/relacionamento que os ligam. Mas um simples par que declara **objetoX** *está relacionado a* **objetoY** não fornece informação suficiente para os fins da engenharia de software. Devemos entender quantas ocorrências do **objetoX** são relacionadas a quantas ocorrências do **objetoY**. Isso leva a um conceito de modelagem de dados chamado de *cardinalidade*.

INFO

Diagramas Entidade/Relacionamento

O par objeto/relacionamento é a pedra fundamental do modelo de dados. Esses pares podem ser representados graficamente por meio do diagrama entidade/relacionamento (DER).[6] O DER foi originalmente proposto por Peter Chen [CHE77] para o projeto de sistemas de base de dados relacional e foi estendido por outros. Um conjunto de componentes primordiais é identificado para o DER: objetos de dados, atributos, relacionamentos e indicadores de vários tipos. A finalidade principal do DER é representar objetos de dados e seus relacionamentos.

Uma notação rudimentar do DER já foi introduzida. Objetos de dados são representados por um retângulo rotulado; relacionamentos são indicados por uma linha rotulada conectando objetos. Em algumas variantes do DER, a linha de conexão contém um losango, que é rotulado com um relacionamento. Conexões entre objetos de dados e relacionamentos são estabelecidas por diversos símbolos especiais que indicam cardinalidade e modalidade.

Para mais informação sobre modelagem de dados e diagrama entidade/realcionamento, o leitor poderá consultar [THA00].

? Como resolvo uma situação em que um objeto de dados está relacionado a muitas ocorrências de um outro objeto de dados?

O modelo de dados deve ser capaz de representar o número de ocorrências dos objetos em um dado relacionamento. Tillmann [TIL93] define cardinalidade de um par objeto/relacionamento do seguinte modo:

"Cardinalidade é a especificação do número de ocorrências de um [objeto] que pode ser relacionado ao número de ocorrências de outro [objeto]". Por exemplo, um objeto pode se relacionar somente com um outro objeto (um relacionamento 1:1), um objeto pode se relacionar com muitos objetos (um relacionamento 1:N); um certo número de ocorrências de um objeto pode se relacionar com um outro número de ocorrências de um outro objeto (um relacionamento M:N).[7] Cardinali-

FERRAMENTAS DE SOFTWARE

Modelagem de Dados

Objetivo: Ferramentas de modelagem de dados fornecem ao engenheiro de software habilidade para representar objetos de dados, suas características e seus relacionamentos. Usadas principalmente para aplicações de grandes bancos de dados e outros projetos de sistemas de informação, ferramentas de modelagem de dados fornecem um meio automatizado para criação de diagramas entidade/relacionamento abrangentes, dicionários de objetos de dados e modelos relacionados.

Mecânica: Ferramentas nessa categoria possibilitam ao usuário descrever objetos de dados e seus relacionamentos. Em alguns casos, as ferramentas usam a notação DER. Em outros, as ferramentas modelam relacionamentos por meio de alguns outros mecanismos. Ferramentas nessa categoria possibilitam a criação de um modelo de banco de dados pela geração do esquema de banco de dados comum para Sistemas de Gestão de Banco de Dados (SGBD; em inglês, DBMS).

Ferramentas Representativas[8]

AllFusion ERWin, desenvolvida por Computer Associates (www3.ca.com), auxilia no projeto de objetos de dados, estrutura adequada e elementos-chave para bancos de dados.

ER/Studio, desenvolvida por Embarcadero Software (www.embarcadero.com), suporta a modelagem entidade/relacionamento.

Oracle Designer, desenvolvida por Oracle Systems (www.oracle.com), modela processos de negócio, entidades e relacionamentos de dados que são transformados em projetos dos quais aplicações completas e bancos de dados são gerados.

MetaScope, desenvolvida por Madrone Systems (www.madronesystems.com), é uma ferramenta de modelagem de dados de baixo custo que suporta a representação gráfica dos dados.

ModelSphere, desenvolvida por Magna Solutions GmbH (www.magnasolutions.com), suporta uma variedade de ferramentas de modelagem relacional.

Visible Analyst, desenvolvida por Visible Systems (www.visible.com), suporta uma variedade de funções de modelagem de análise incluindo modelagem de dados.

6 Embora o DER ainda seja usado em algumas aplicações de projeto de banco de dados, a notação UML é agora mais comumente utilizada para o projeto de dados.

7 Por exemplo, um tio pode ter muitos sobrinhos e um sobrinho pode ter muitos tios.

8 As ferramentas mencionadas aqui não representam uma recomendação mas, em vez disso uma exemplificação de ferramentas dessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas de seus respectivos desenvolvedores.

Document shared on www.docsity.com

dade também define "o número máximo de objetos que podem participar de um relacionamento" [TIL93]. Todavia, não indica se um objeto de dados particular deve ou não participar de um relacionamento. Para especificar essa informação, o modelo de dados acrescenta modalidade ao par objeto/relacionamento.

A *modalidade* de uma relação é 0 se não houver necessidade explícita de o relacionamento ocorrer ou se o relacionamento for opcional. A modalidade é 1 se a ocorrência do relacionamento for obrigatória.

> "Para que um sistema de informação possa ser útil, confiável, adaptável e econômico, deve ser baseado, inicialmente, em uma sólida modelagem de dados e, somente depois, em análise do processo... porque a estrutura de dados é inerentemente sobre a verdade enquanto o processo é sobre técnica."
>
> Duncan Dwellle

8.4 Análise Orientada a Objetos

Qualquer discussão sobre análise orientada a objetos deve começar focalizando o termo *orientado a objetos*. O que é um ponto de vista orientado a objetos? Por que um método é considerado orientado a objetos? O que é um objeto? À medida que OO ganhou inúmeros adeptos durante as décadas de 1980 e 1990, têm havido muitas opiniões diferentes (por exemplo, [BER93], [TAY90], [STR88], [BOO86]) sobre as respostas corretas a essas questões. Hoje, uma visão coerente de OO emergiu. O objetivo da OO é definir todas as classes relevantes ao problema a ser resolvido — as operações e os atributos associados a elas, as relações entre elas e o comportamento que elas exibem. Para tanto, algumas tarefas devem ser realizadas:

1. Os requisitos básicos do usuário precisam ser discutidos entre o cliente e o engenheiro de software (veja o Capítulo 7).
2. As classes precisam ser identificadas (ou seja, atributos e métodos são definidos).
3. Uma hierarquia de classes precisa ser especificada.
4. As relações de objeto para objeto (conexões entre objetos) devem ser representadas.
5. O comportamento do objeto precisa ser modelado.
6. As tarefas de 1 a 5 são reaplicadas iterativamente até que o modelo seja completado.

Em vez de examinarmos o problema usando um modelo mais convencional de entrada-processamento-saída (fluxo de informação) ou um modelo derivado exclusivamente de estruturas de informação hierárquicas, a Análise Orientada a Objetos (AOO) cria um modelo orientado a classes que se apóia no entendimento dos conceitos de OO.

INFO

Conceitos Orientados a Objetos

Conceitos Orientados a Objetos (OO) tornaram-se bem estabelecidos no mundo da engenharia de software. A seguir são apresentadas descrições resumidas de conceitos OO importantes, freqüentemente encontrados durante a modelagem de análise. Conceitos OO adicionais, mais diretamente relacionados com projeto de software, são apresentados no Capítulo 10.

Atributos — uma coleção de valores de dados que descrevem uma classe.

Classe — encapsula dados e abstrações procedurais necessárias para descrever o conteúdo e o comportamento de alguma entidade do mundo real. Dito de outro modo, classe é uma descrição generalizada (por exemplo, um gabarito, padrão ou planta) que descreve uma coleção de objetos semelhantes.

Objetos — instâncias de uma classe específica. Objetos herdam os atributos e operações da classe.

Operações — também chamadas de *métodos* e *serviços*, fornecem uma representação de um dos comportamentos da classe.

Subclasse — especialização da superclasse. Uma subclasse pode herdar tanto atributos quanto operações de uma superclasse.

Superclasse — também chamada de *classe-base*, é uma generalização de um conjunto de classes relacionadas a ela.

8.5 Modelagem Baseada em Cenário

Embora o sucesso de um sistema ou produto baseado em computador seja medido de diferentes modos, a satisfação do usuário está no topo da lista. Se engenheiros de software entendem como os usuários finais (e outros atores) querem interagir com um sistema, a equipe de software estará melhor habilitada para caracterizar apropriadamente os requisitos e construir modelos significativos de análise e projeto. Assim, a modelagem de análise com UML começa com a criação de cenários na forma de casos de uso, diagramas de atividade e diagramas de raias.

8.5.1 Escrita de Casos de Uso

Um caso de uso capta as interações que ocorrem entre produtores e consumidores de informação e o sistema em si. Nesta seção, examinamos como os casos de uso são desenvolvidos como parte da atividade de modelagem de análise.[9]

O conceito de caso de uso (veja o Capítulo 7) é relativamente fácil de entender — descreve um cenário de uso específico em linguagem direta do ponto de vista de um ator definido.[10] Porém, precisamos saber (1) sobre o que escrever, (2) quanto devemos escrever sobre isso, (3) quão detalhada deve ser a nossa descrição, (4) como organizar a descrição. Essas são as questões que precisam ser respondidas se os casos de uso tiverem de mostrar valor como ferramenta de modelagem de análise.

AVISO

Em algumas situações, casos de uso tornam-se os mecanismos dominantes da engenharia de requisitos. No entanto, isso não significa que você deve descartar os conceitos e as técnicas discutidos no Capítulo 7.

> "[Casos de uso] são simplesmente um apoio para definir o que existe fora do sistema (atores) e o que deve ser realizado pelo sistema (casos de uso)."
>
> Ivar Jacobson

Sobre o que escrever? As duas primeiras tarefas de engenharia de requisitos[11] — concepção e extração — fornecem-nos a informação de que precisamos para iniciar a escrita de casos de uso. Reuniões de coleta de requisitos, IFQ e outros mecanismos de engenharia de requisitos são usados para identificar os interessados, definir o escopo do problema, especificar os objetivos operacionais gerais, esboçar todos os requisitos funcionais conhecidos e descrever as coisas (objetos) que vão ser manipuladas pelo sistema.

Para iniciar o desenvolvimento de um conjunto de casos de uso, as funções ou atividades realizadas por um ator específico são listadas. Essas podem ser obtidas de uma lista de funções necessárias para o sistema, por meio de conversas com os clientes ou usuários finais, ou por uma avaliação dos diagramas de atividade (veja a Seção 8.5.2) desenvolvidos como parte da modelagem de análise.

9 Os casos de uso são parte especialmente importante da modelagem de análise para interfaces de usuários. Análise de interface é discutida em detalhes no Capítulo 12.

10 Um ator não é uma pessoa específica, mas, em vez disso, um papel que uma pessoa (ou um dispositivo) desempenha em um contexto específico. Um ator "pede ao sistema para disponibilizar um dos seus serviços" [COC01].

11 Essas tarefas de engenharia de requisitos são discutidas em detalhe no Capítulo 7.

Document shared on www.docsity.com

CASASEGURA

Desenvolvimento de Outro Cenário Preliminar de Usuário

A cena: Uma sala de reunião, durante o segundo encontro para coleta de requisitos.

Os personagens: Jamie Lazar, membro da equipe de engenharia; Ed Robbins, membro da equipe de engenharia; Doug Miller, gerente de engenharia de software; três membros de marketing; um representante de engenharia do produto; e um facilitador.

A conversa:

Facilitador: É hora de começarmos a falar sobre a função de vigilância do *CasaSegura*. Vamos desenvolver um cenário de usuário para o acesso à função de segurança residencial.

Jamie: Quem fará o papel de ator nessa situação?

Facilitador: Acho que Meredith (uma pessoa de marketing) tem trabalhado nessa funcionalidade. Por que você não faz o papel?

Meredith: Você quer fazer do mesmo modo que fizemos da última vez, certo?

Facilitador: Certo... do mesmo modo.

Meredith: Bem, obviamente a razão para vigilância é permitir ao proprietário verificar o exterior da casa enquanto estiver longe, gravar e rever o vídeo que foi captado... essa espécie de coisa.

Ed: O vídeo será digital e estará armazenado em disco?

Facilitador: Boa pergunta, mas vamos adiar os tópicos de implementação por enquanto. Meredith?

Meredith: Está bem, então basicamente há duas partes para a função de vigilância... a primeira configura o sistema, inclusive inserindo uma planta baixa — nós precisamos de ferramentas para ajudar o proprietário a fazer isso —, e a segunda parte é a função real de vigilância propriamente dita. Como a inserção é parte da atividade de configuração, enfocarei a função de vigilância.

Facilitador (sorrindo): Você tirou as palavras da minha boca.

Meredith: Vejamos... eu quero ter acesso à função de vigilância via PC ou via Internet. Minha idéia é que o acesso pela Internet seja usado mais freqüentemente. De todo modo, quero poder mostrar visões da câmera em um PC e controlar o *zoom* e o *pan** de uma câmera específica; específico a câmera selecionando-a na planta baixa. Quero seletivamente gravar e rever a saída da câmera. Também quero poder bloquear o acesso a uma ou mais câmeras com uma senha específica. E quero a opção de visualizar pequenas janelas que mostrem vistas de todas as câmeras e, então, escolher uma que eu desejo aumentar.

Jamie: Essas são as chamadas visões múltiplas.

Meredith: Está certo, então quero visões múltiplas de todas as câmeras. Também quero que a interface da função de vigilância tenha a mesma aparência de todas as outras interfaces do *CasaSegura*. Ou seja, que ela seja intuitiva, o que significa que não quero ter que ler um manual para usá-la.

Facilitador: Bom trabalho, agora vamos cuidar dessa função com um pouco mais de detalhe...

A função de vigilância residencial do *CasaSegura* (subsistema) discutida no quadro identifica as seguintes funções (uma lista resumida), executadas pelo ator **proprietário**:

- Acessar a câmera de vigilância via Internet.
- Selecionar câmera para visualização.
- Solicitar visão múltipla para todas as câmeras.
- Mostrar as visões da câmera em uma janela de PC.
- Controlar pan e zoom de uma câmera específica.
- Registrar seletivamente a saída de câmera.
- Reproduzir saída de câmera.

À medida que conversas posteriores com o interessado (que desempenha o papel de proprietário) progridem, a equipe de coleta de requisitos desenvolve casos de uso para cada uma das funções anotadas. Em geral, os casos de uso são escritos primeiro em uma narrativa de modo informal. Se mais formalidade for necessária, o mesmo caso de uso é reescrito com um formato estruturado semelhante àquele proposto no Capítulo 7 e reproduzido nesta seção em um quadro.

? Como examino alternativas de cursos de ação quando desenvolvo um caso de uso?

* N. de R.T.: *Zoom* é o controle de aproximação pela lente da câmera a um detalhe de objeto, ou distanciamento. *Pan* é um movimento da câmera no tripé para mudar a direção para a qual ela está apontada.

Para ilustrar, considere a função "acessar a câmera de vigilância — exibir as visões da câmera (ACV-EVC)". O interessado que desempenha o papel do ator **proprietário** pode escrever a seguinte narrativa:

Caso de uso: Acessar a câmera de vigilância — exibir visões da câmera (ACV-EVC)

Ator: proprietário

Se eu estiver em um local remoto, posso usar qualquer PC com um software de navegação adequado para ter acesso ao site de *Produtos CasaSegura*. Introduzo minha ID de usuário e dois níveis de senha e, depois de validado, tenho acesso a toda funcionalidade do meu sistema *CasaSegura* instalado. Para ter acesso a uma visão de câmera específica, seleciono "vigilância" dentre os botões exibidos com as funções principais. Então seleciono "escolher uma câmera", e a planta baixa da casa é exibida. Depois seleciono a câmera em que estou interessado. Como alternativa, posso utilizar o recurso visão múltipla e visualizar todas as câmeras simultaneamente, selecionando "todas as câmeras" como minha escolha de visão. Uma vez escolhida uma câmera, seleciono "visão", e a visão de um quadro por segundo aparece em uma janela de visão que está identificada pela ID da câmera. Se eu quiser trocar de câmeras, seleciono "escolher uma câmera", a janela de visão original desaparece, e a planta baixa da casa é mostrada outra vez. Então, seleciono a câmera que me interessa. Uma nova janela de visão aparece.

Uma variante da narrativa do caso de uso apresenta a interação como uma seqüência ordenada de ações do usuário. Cada ação é representada como uma sentença declarativa. Revisando a função ACV-EVC, poderíamos escrever:

Caso de uso: Acessar a câmera de vigilância — exibir visões da câmera (ACS-DCV)

Ator: proprietário

1. O proprietário obtém acesso ao site de *Produtos CasaSegura*.
2. O proprietário introduz sua ID de usuário.
3. O proprietário introduz duas senhas (cada uma com no mínimo oito caracteres).
4. O sistema mostra todos os botões das principais funções.
5. O proprietário seleciona "vigilância" dentre os botões das principais funções.
6. O proprietário seleciona "escolher uma câmera".
7. O sistema mostra a planta baixa da casa.
8. O proprietário seleciona um ícone de câmera na planta baixa.
9. O proprietário seleciona o botão "visão".
10. O sistema mostra uma janela de visão que é identificada pela ID da câmera.
11. O sistema mostra a saída de vídeo na janela de visão com um quadro por segundo.

É importante notar que essa apresentação seqüencial não considera qualquer alternativa de interações (a narrativa é mais fluente e representa algumas alternativas). Os casos de uso desse tipo são algumas vezes referidos como *cenários principais* [SCH98].

> "Casos de uso podem ser usados em muitos processos [software]. Nosso favorito é um processo iterativo e pautado por risco."
>
> Geri Schneider e Jason Winters

Obviamente, uma descrição de interações alternativas é essencial para completo entendimento da função que está sendo descrita. No entanto, cada passo do cenário principal é avaliado formulando-se as seguintes questões [SCH98]:

- O ator pode realizar alguma outra ação neste ponto?
- É possível que o ator encontre alguma condição de erro neste ponto? Em caso afirmativo, qual seria ela?

Document shared on www.docsity.com

- É possível que o ator encontre algum outro comportamento neste ponto (por exemplo, comportamento que é invocado por algum evento fora do controle do ator)? Em caso afirmativo, qual seria ele?

Respostas a essas questões resultam na criação de um conjunto de cenários secundários que são parte do caso de uso original, mas representam comportamento alternativo.

Por exemplo, considere os passos 6 e 7 do cenário principal anteriormente apresentado:

6. O proprietário seleciona "escolher uma câmera".
7. O sistema mostra a planta baixa da casa.

O ator pode realizar alguma outra ação neste ponto? A resposta é sim. Referindo-se à narrativa fluente, o ator pode escolher a visão múltipla de todas as câmeras simultaneamente. Assim, um cenário secundário poderia ser: "Ver visões múltiplas condensadas de todas as câmeras."

É possível que o ator encontre alguma condição de erro neste ponto? Qualquer número de condições de erro pode ocorrer quando se opera um sistema baseado em computador. Nesse contexto, consideramos apenas condições de erro que são prováveis como resultado direto da ação descrita no passo 6 ou no passo 7. Novamente a resposta à questão é sim. Uma planta baixa com ícones das câmeras pode nunca ter sido configurada. Selecionar "escolher uma câmera" resulta em uma condição de erro "Nenhuma planta baixa configurada para essa casa".[12] Essa condição de erro torna-se um cenário secundário.

É possível que o ator encontre algum outro comportamento neste ponto? Novamente a resposta é sim. À medida que os passos 6 e 7 ocorrem, o sistema pode encontrar uma condição de alarme. Isso poderia resultar na exibição pelo sistema de uma notificação especial de alarme (tipo, localização, ação do sistema) e fornecer ao ator um número de opções relevantes conforme a natureza do alarme. Como o cenário secundário pode ocorrer para virtualmente todas as interações, ele não será parte do caso de uso **AVC-EVC**. Em vez disso, um caso de uso separado — "Condição de alarme encontrada" — seria desenvolvido e referido por meio de outros casos de uso quando necessário.

Segundo o models de caso de uso formal mostrado no quadro a seguir, os cenários secundários são representados como exceções da seqüência básica descrita para **ACV-EVC.**

CASASEGURA

Gabarito do Caso de Uso de Vigilância

Caso de uso: Ter acesso à câmera de vigilância — exibir as visões da câmera (ACV-EVC).

Ator principal: Proprietário.

Objetivo no contexto: Ver saída de câmera, colocada na casa, de qualquer localização remota via Internet.

Pré-condições: O sistema deve estar completamente configurado; devem ser obtidas ID de usuário e senhas adequadas.

Disparo: O proprietário decide dar uma olhada dentro da casa estando fora.

Cenário:

1. O proprietário obtém acesso ao site *Produtos CasaSegura*.
2. O proprietário introduz sua ID de usuário.
3. O proprietário introduz duas senhas (cada uma com no mínimo oito caracteres).
4. O sistema mostra todos os botões das principais funções.
5. O proprietário seleciona "vigilância" dentre os botões das principais funções.
6. O proprietário seleciona "escolher uma câmera".
7. O sistema mostra a planta baixa da casa.
8. O proprietário seleciona um ícone de câmera na planta baixa.
9. O proprietário seleciona o botão "visão".
10. O sistema mostra a janela de visão que é identificada pela ID da câmera.
11. O sistema mostra saída de vídeo na janela de visão com um quadro por segundo.

12 Nesse caso, um outro ator, o **administrador do sistema**, teria de configurar a planta baixa, instalar e iniciar (por exemplo, associar uma ID de equipamento) todas as câmeras e testá-las verificando se cada uma está acessível por meio do sistema e da planta baixa.

Exceções:

1. ID ou senhas estão incorretas ou não reconhecidas — veja caso de uso: "validar ID e senhas".
2. Função de Vigilância não configurada para este sistema — sistema exibe mensagem de erro apropriada; veja caso de uso "configurar função de vigilância".
3. Proprietário seleciona "visão múltipla condensada de todas as câmeras" — veja caso de uso "ver visão múltipla condensada de todas as câmeras".
4. Uma planta baixa não está disponível ou não foi configurada — exibir mensagem de erro apropriada e veja caso de uso "configurar planta baixa".
5. Uma condição de alarme é encontrada — veja caso de uso "condição de alarme encontrada".

Prioridade: Prioridade moderada, a ser implementada depois das funções básicas.

Quando disponível: Terceiro incremento.

Freqüência de uso: Não freqüente.

Canal para o ator: Navegador via PC e conexão com o site *CasaSegura*.

Atores secundários: Administrador do sistema, câmeras.

Canais para os atores secundários:

1. Administrador do sistema: sistema baseado em PC.
2. Câmeras: conectividade sem fio.

Tópicos em aberto:

1. Que mecanismos protegem o uso não autorizado desse recurso por empregados da empresa?
2. A segurança é suficiente? Invasores nessa característica representariam a principal invasão de privacidade.
3. Será aceitável a resposta do sistema via Internet, dada a largura de banda necessária para as visões de câmeras?
4. Vamos desenvolver a possibilidade de fornecer vídeo a uma velocidade maior em quadros por segundo, quando conexões de banda mais larga estiverem disponíveis?

Veja na Web

Quando você termina de escrever casos de uso? Para uma discussão valiosa sobre esse tópico, veja ootips.org/use-cases-done.htmlootips.org/use-cases-done.html.

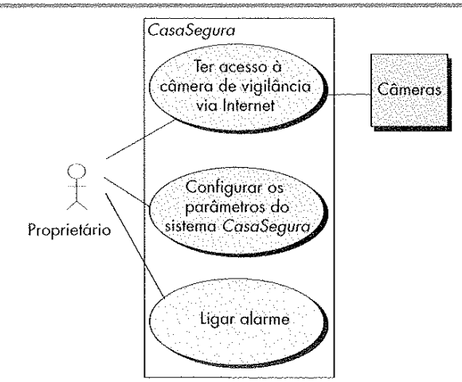
Em muitos casos, não há necessidade de criar uma representação gráfica de um cenário de uso. Entretanto, representação diagramática pode facilitar o entendimento, particularmente quando o cenário é complexo. Como observamos no Capítulo 7, UML fornece habilidade diagramática de caso de uso. A Figura 8.6 descreve um diagrama de caso de uso preliminar para o produto *CasaSegura*. Cada caso de uso é representado por uma elipse. Somente o caso de uso **ACV-EVC** está discutido em detalhe nesta seção.

8.5.2 Desenvolver um Diagrama de Atividade

O diagrama de atividade UML (discutido brevemente no Capítulo 7) complementa o caso de uso fornecendo uma representação gráfica do fluxo de interação em um cenário específico. Analogamente ao fluxograma, um diagrama de atividade usa retângulos arredondados para descrever uma função específica do sistema, setas para representar fluxo através do sistema, losangos de decisão para representar decisões de derivação (cada seta originária de um losango é rotulada), e linhas horizontais sólidas para indicar que estão ocorrendo atividades paralelas. Um diagrama de atividade

FIGURA 8.6

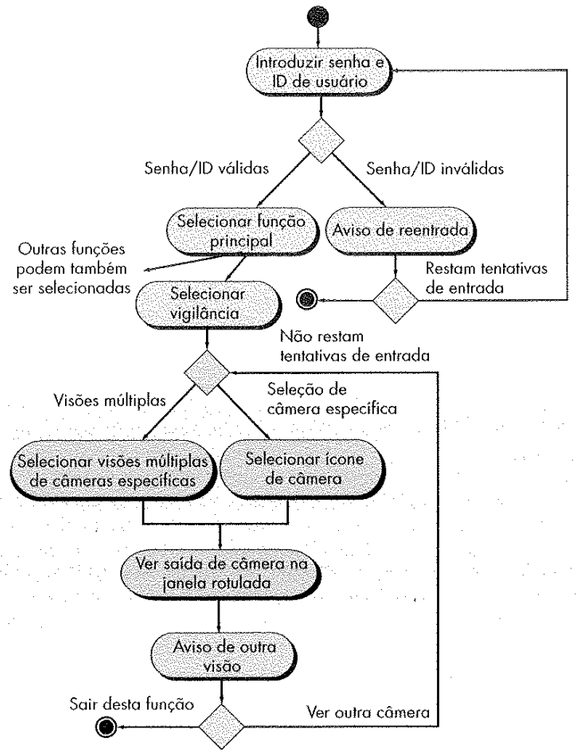
Diagrama de caso de uso preliminar para o sistema *CasaSegura*



Document shared on www.docsity.com

FIGURA 8.7

Diagrama de atividade para ter acesso à câmera de vigilância – função exibir visões de câmera



PONTO CHAVE

Um diagrama de atividade UML representa as ações e decisões que ocorrem à medida que alguma função é realizada.

PONTO CHAVE

Um diagrama de raias UML representa o fluxo de ações e decisões e indica quais atores as realizam.

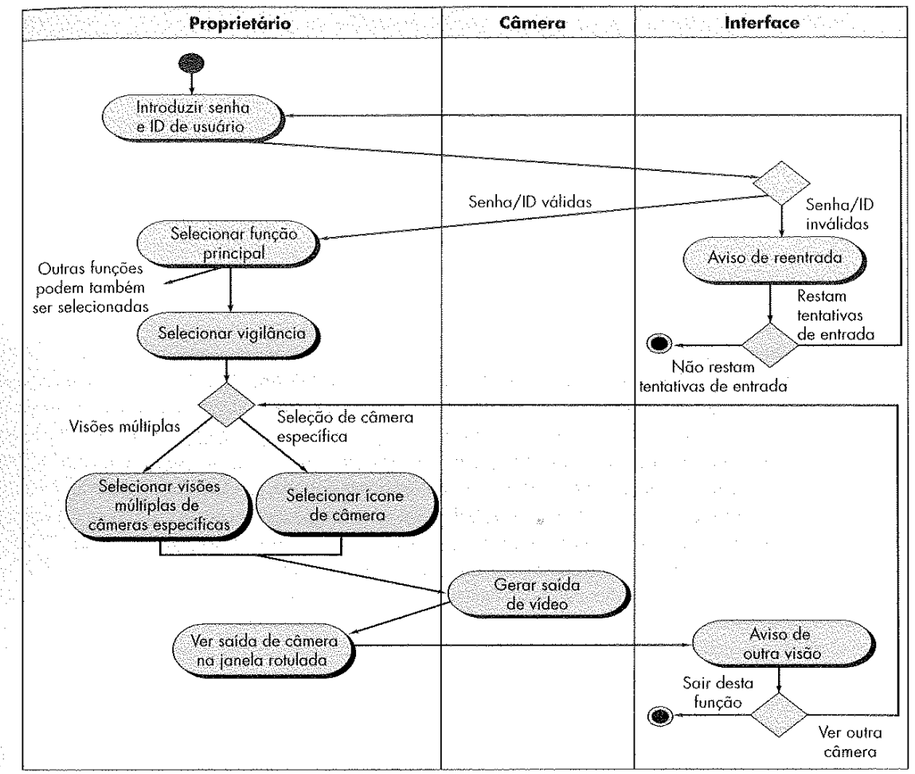
para a função **ACV-EVC** é mostrado na Figura 8.7. Deve-se observar que o diagrama de atividade acrescenta detalhes adicionais, não diretamente mencionados (mas implícitos) no caso de uso. Por exemplo, um usuário pode somente tentar introduzir a **ID do usuário** e **senha**, um número limitado de vezes. Isso é representado pelo losango de decisão abaixo de *Aviso de reentrada.*

8.5.3 Diagramas de Raias

O *diagrama de raias* UML é uma variação útil do diagrama de atividade que permite, ao modelador, representar o fluxo de atividades descrito pelo caso de uso e, ao mesmo tempo, indicar que ator (se houver vários atores envolvidos em uma função específica) ou classe de análise é responsável pela ação descrita por um retângulo de atividade. Responsabilidades são representadas por segmentos paralelos que dividem o diagrama verticalmente, como as raias de uma piscina de natação.

Três classes de análise — **Proprietário**, **Interface** e **Câmera** — têm responsabilidades diretas ou indiretas no contexto do diagrama de atividade representado na Figura 8.7. De acordo com a Figura 8.8, o diagrama de atividade é rearranjado de modo que as atividades associadas com uma particular classe de análise caiam dentro da raia daquela classe. Por exemplo, a classe **Interface** representa a interface do usuário como vista pelo proprietário. O diagrama de atividade tem dois avisos que são de responsabilidade da interface — *aviso para reentrar* e *aviso para outra visão.* Esses avisos e as decisões associadas a eles caem dentro da raia da **Interface**. No entanto, setas partindo daquela raia voltam para a raia do **Proprietário**, onde as ações do proprietário ocorrem.

FIGURA 8.8 Diagrama de raias para ter acesso à câmera de vigilância — função exibir visões de câmera



8.6 MODELAGEM ORIENTADA A FLUXO

Modelagem orientada a fluxo de dados continua a ser uma das notações de análise mais amplamente usadas atualmente.[13] Embora o *diagrama de fluxo de dados* (DFD) e diagramas e informações relacionados não sejam parte formal da UML, eles podem ser usados para complementar os diagramas UML e fornecer visão adicional dos requisitos e fluxo do sistema.

O DFD tem uma visão entrada-processo-saída de um sistema. Ou seja, objetos de dados entram no software, são transformados por elementos de processamento e os objetos de dados resultantes saem do software. Objetos de dados são representados por setas rotuladas e transformações são representadas por círculos (também chamados de *bolhas*). O DFD é apresentado de modo hierárquico, isto é, o primeiro modelo de fluxo de dados (algumas vezes chamado de DFD nível 0 ou diagrama de contexto) representa o sistema como um todo. Diagramas de fluxo de dados subseqüentes refinam o diagrama de contexto, fornecendo detalhamento progressivo em cada nível subseqüente.

13 Modelagem de fluxo de dados é uma atividade de modelagem central em *análise estruturada.*

Document shared on www.docsity.com

> "O objetivo do diagrama de fluxo de dados é fornecer uma ponte semântica entre os usuários e os desenvolvedores de sistemas."
>
> **Kenneth Kozar**

AVISO

Alguns vão sugerir que o DFD é da "velha escola" e que não tem lugar na prática moderna. Essa é uma visão que exclui um modo potencialmente útil de representação no nível de análise. Se ele puder ajudar, use DFD.

8.6.1 Criação de um Modelo de Fluxo de Dados

O diagrama de fluxo de dados permite ao engenheiro de software desenvolver modelos do domínio informacional e do domínio funcional ao mesmo tempo. À medida que o DFD é refinado em maior nível de detalhe, o analista realiza uma decomposição funcional implícita do sistema, cumprindo assim o quarto princípio operacional de análise para função. Ao mesmo tempo, o refinamento do DFD resulta em um refinamento correspondente dos dados à medida em que se movem pelos processos que constituem a aplicação.

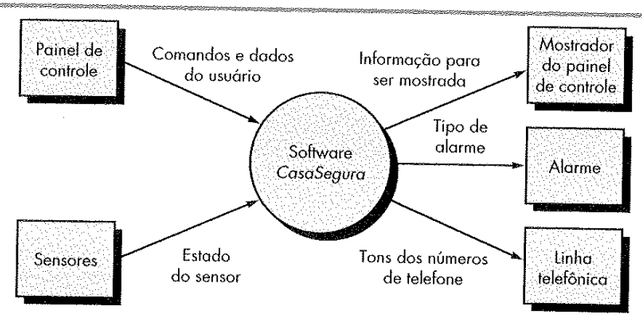
Algumas diretrizes simples podem ajudar imensamente durante a derivação do diagrama de fluxo de dados: (1) o diagrama de fluxo de dados de nível 0 deve mostrar o software/sistema como uma única bolha; (2) a entrada e a saída principal devem ser cuidadosamente registradas; (3) o refinamento deve começar pelo isolamento dos processos, objetos de dados e depósitos de dados candidatos a ser representados no nível seguinte; (4) todas as setas e bolhas devem ser rotuladas com nomes significativos; (5) a continuidade do fluxo de informação deve ser mantida de nível para nível;[14] e (6) uma bolha de cada vez deve ser refinada. Há uma tendência natural de supercomplicar o diagrama de fluxo de dados. Isso ocorre quando o analista tenta mostrar excesso de detalhes prematuramente ou representar aspectos procedurais do software em vez do fluxo da informação.

Para ilustrarmos o uso de DFD e da notação relacionada, novamente consideramos a função de segurança do *CasaSegura*. Um DFD de nível de contexto da função de segurança é mostrado na Figura 8.9. As entidades externas principais (caixas) produzem informação para uso do sistema e consomem informação gerada pelo sistema. As setas rotuladas representam objetos de dados ou hierarquias de tipos de objetos de dados. Por exemplo, **comandos e dados do usuário** incluem todos os comandos de configuração, todos os comandos de ativação/desativação, todas as variedades de interações e todos os dados que dão entrada para qualificar ou expandir um comando.

O DFD de nível 0 é agora expandido para um modelo de nível 1. No entanto, como devemos proceder? Uma abordagem simples, mas efetiva, é realizar uma "análise gramatical" [ABB83] da narrativa que descreve a bolha no nível de contexto. Isto é, isolamos todos os substantivos (e frases substantivas) e verbos (e frases verbais) na narrativa[15] de *processamento* da *CasaSegura*, derivada durante a primeira reunião de coleta de requisitos. Para ilustrar, considere o seguinte texto de

FIGURA 8.9

DFD em nível de contexto da função de segurança do *CasaSegura*



14 Ou seja, os objetos de dados que fluem para dentro do sistema ou qualquer transformação em um nível precisam ser os mesmos objetos de dados (ou suas partes constituintes) que fluem para dentro da transformação em um nível mais refinado.

15 Uma narrativa de processamento é semelhante, em estilo, ao caso de uso, mas um tanto diferente em finalidade. A narrativa de processamento fornece uma descrição geral da função a ser desenvolvida. Não é um cenário escrito do ponto de vista de um ator.

PONTO CHAVE

A continuidade do fluxo de informação deve ser mantida à medida que cada nível do DFD é refinado. Isso significa que a entrada e saída para um nível devem ser as mesmas que a entrada e saída para o nível refinado.

AVISO

A análise gramatical não é a toda prova, mas pode proporcionar-lhe um excelente passo inicial, se você está lutando para definir objetos de dados e as transformações que neles operam.

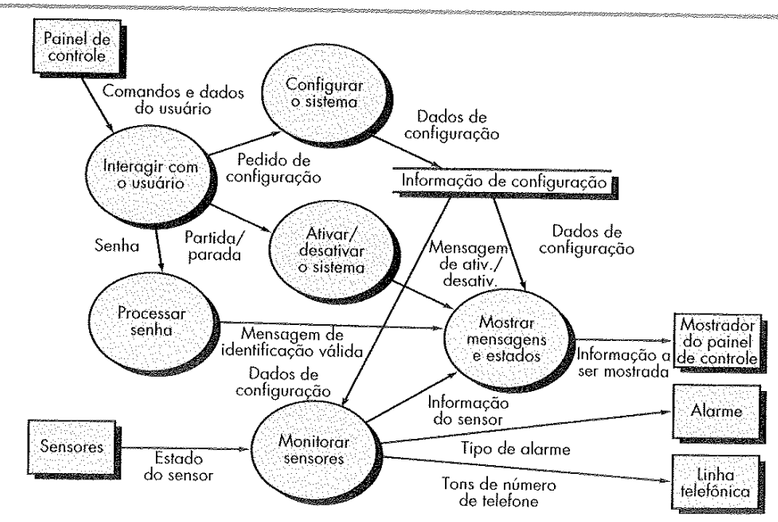
narrativa de processamento, com a primeira ocorrência de todos os substantivos sublinhada e a primeira ocorrência de todos os verbos em itálico.[16]

> A função de segurança CasaSegura *permite* que o proprietário *configure* o sistema de segurança quando é *instalado, monitora* todos os sensores *conectados* ao sistema de segurança e *interage* com o proprietário por intermédio da Internet, de um PC ou do painel de controle.
>
> Durante a instalação, o PC do CasaSegura é usado para *programar* e *configurar* o sistema. A cada sensor é atribuído um número e tipo, uma senha mestra é programada para *armar* e *desarmar* o sistema e números de telefone são *introduzidos* para serem *discados* quando um evento de sensor ocorrer.
>
> Quando um evento de sensor é *reconhecido*, o software *aciona* um alarme sonoro conectado ao sistema. Após um tempo de espera, especificado pelo proprietário durante as atividades de configuração do sistema, o sistema disca o número do telefone de um serviço de monitoramento, *fornece* informação sobre o local e *informa* a natureza do evento que foi detectado. O número de telefone será *rediscado* a cada 20 segundos até que a conexão telefônica seja *conseguida*.
>
> O proprietário *recebe* informação de segurança via painel de controle, PC ou navegador, coletivamente chamados de interface. A interface *exibe* mensagens de aviso e informação do estado do sistema no painel de controle, PC ou janela do navegador. A interação do proprietário assume a seguinte forma....

Observando a "análise gramatical", um padrão começa a emergir. Os verbos são processos *CasaSegura;* eles podem, em última análise, ser representados como bolhas em um DFD subseqüente. Os substantivos ou são entidades externas (caixas), ou objetos de dados ou de controle (setas), ou depósitos de dados (linhas duplas). Note ainda que substantivos e verbos podem ser acoplados uns aos outros. Por exemplo, cada sensor tem um número e um tipo a ele atribuídos, assim, **número** e **tipo** são atributos do objeto de dados **sensor**. Conseqüentemente, pela realização de uma análise gramatical na narrativa de processamento de uma bolha, em qualquer nível de DFD, podemos gerar muita informação útil sobre como prosseguir com o refinamento para o nível seguinte. Usando essa informação, um DFD de nível 1 é mostrado na Figura 8.10. O processo no nível de contexto, apresentado na Figura 8.9, foi expandido em seis processos derivados do exame da análise gramatical.

FIGURA 8.10

DFD nível 1 para a função de segurança de *CasaSegura*

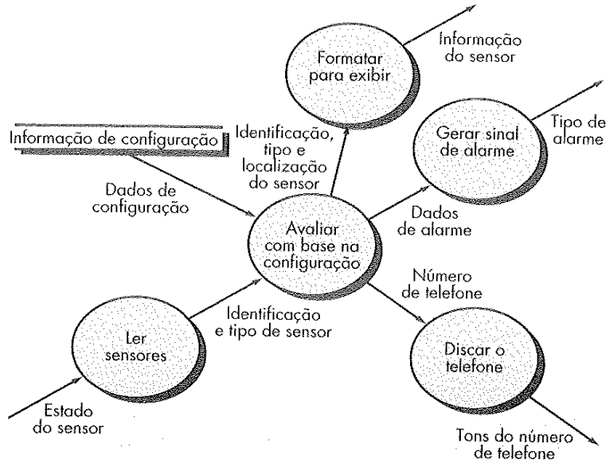


16 Devemos chamar a atenção para o fato de que substantivos e verbos que são sinônimos ou não têm relação direta com o processo de modelagem são omitidos. Observe também que usaremos uma análise gramatical similar quando abordarmos a modelagem baseada em classes na Seção 8.7.

Document shared on www.docsity.com

FIGURA 8.11

DFD nível 2 que refina o processo monitorar sensores



AVISO

Certifique-se de que a narrativa de processamento que você pretende analisar esteja toda escrita no mesmo nível de abstração.

Analogamente, o fluxo de informação entre processos no nível 1 foi derivado da análise. Além disso, a continuidade do fluxo de informação é mantida entre os níveis 0 e 1.

Os processos representados no DFD de nível 1 podem ser refinados em níveis inferiores. Por exemplo, o processo *monitorar sensores* pode ser refinado em um DFD de nível 2, como é mostrado na Figura 8.11. Note novamente que a continuidade do fluxo de informação foi mantida entre os níveis.

O refinamento dos DFDs continua até que cada bolha realize uma simples função, ou seja, até que o processo representado pela bolha desempenhe uma função que seria facilmente implementada como um componente de programa. No Capítulo 9, discutiremos um conceito chamado *coesão*, que pode ser usado para avaliar a simplicidade de uma dada função. Por enquanto, tentamos refinar os DFDs até que cada bolha tenha um "único objetivo".

8.6.2 Criação de um Modelo de Fluxo de Controle

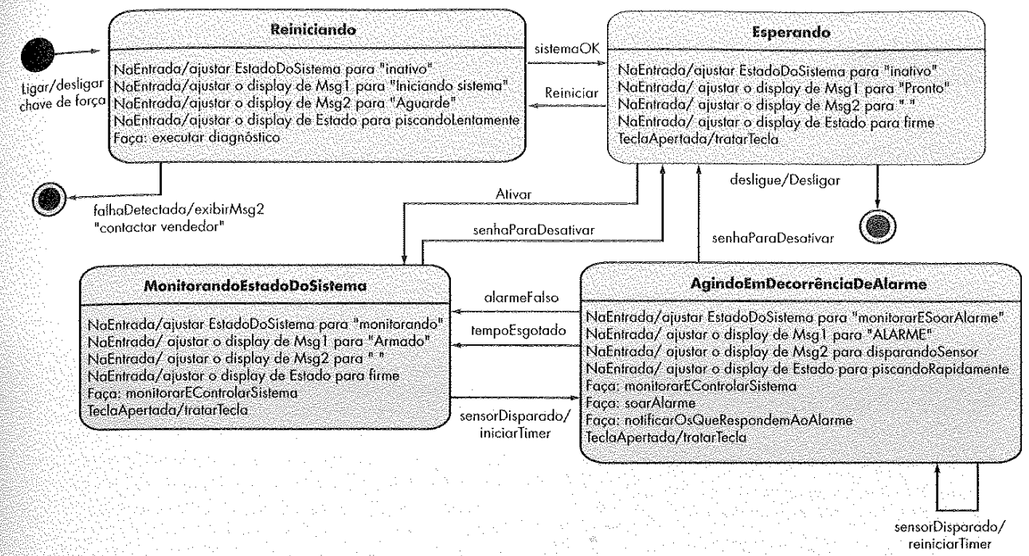
Para muitos tipos de aplicação de processamento de dados, o modelo de dados e o diagrama de fluxo de dados são tudo o que é necessário para obter significativo conhecimento dos requisitos do software. Como já mencionamos anteriormente, no entanto, uma grande classe de aplicações é "orientada" por eventos em vez de dados; produzem informação de controle, em vez de relatórios ou imagens, e processam informação com grande preocupação de tempo e desempenho. Tais aplicações requerem o uso *de modelagem de fluxo de controle*, além da modelagem de fluxo de dados.

Já mencionamos anteriormente que um evento ou item de controle é implementado como um valor booleano (por exemplo, verdadeiro ou falso, ligado ou desligado, 1 ou 0) ou uma lista separada de condições (vazio, emperrado, cheio). Para selecionar potenciais candidatos a eventos, as seguintes diretrizes são sugeridas:

? Como seleciono eventos potenciais para um diagrama de fluxo de controle, diagrama de estado ou CSPEC?

- Liste todos os sensores que são "lidos" pelo software.
- Liste todas as condições de interrupção.
- Liste todas as "chaves" que são ajustadas por um operador.
- Liste todas as condições de dados.
- Voltando à análise substantivo/verbo aplicada à narrativa de processamento, reveja todos os "itens de controle" como possíveis para controlar o fluxo de entradas/saídas.

FIGURA 8.12 Diagrama de estado para a função de segurança *CasaSegura*



- Descreva o comportamento de um sistema pela identificação de seus estados; identifique como cada estado é atingido; e defina as transições entre estados.
- Concentre-se em possíveis omissões — um erro muito comum na especificação de controle; por exemplo, pergunte: "Há qualquer outro modo de chegar a esse estado ou sair dele?".

8.6.3 A Especificação de Controle

A *especificação de controle* (*control specification*, CSPEC) representa o comportamento do sistema (no nível em que ela foi referida) de dois modos diferentes.[17] A CSPEC contém um diagrama de transição de estado, que é uma especificação de comportamento seqüencial. Ela pode também conter uma tabela de ativação de programa — uma especificação de comportamento combinatória. A Figura 8.12 mostra um diagrama de estado preliminar[18] para o modelo de fluxo de controle de nível 1 de *CasaSegura*. O diagrama indica como o sistema responde a eventos à medida que passa pelos quatro estados definidos nesse nível. Ao revisar o diagrama de estado, um engenheiro de software pode determinar o comportamento do sistema e, mais importante, pode verificar se há "furos" no comportamento especificado.

Por exemplo, o diagrama de estado (veja a Figura 8.12) indica que as transições que se originam no estado *Esperando* podem ocorrer se o sistema é reiniciado, ativado ou desligado. Se o sistema é ativado (isto é, o sistema de alarme é ligado), uma transição para o estado *MonitorandoEstadodoSistema* ocorre, as mensagens exibidas são trocadas como mostrado e o processo **monitorarEControlarSistema** é chamado. Duas transições ocorrem do estado *MonitorandoEstadoDoSistema* – (1) quando o sistema é desativado, uma transição ocorre de volta para o estado *Esperando*; (2) quando um sensor é disparado, uma transição para o estado *AgindoEmDecorrenciaDeAlarme* ocorre. Todas as transições e os conteúdos de todos os estados são considerados durante a revisão.

17 Posteriormente, neste capítulo, será apresentada notação adicional de modelagem de comportamento.

18 A notação de diagrama de estado usada aqui concorda com a notação UML. Um "diagrama de transição de estado" está disponível em análise estruturada, mas o formato UML é superior em conteúdo de informação e representação.

Document shared on www.docsity.com

A CSPEC descreve o comportamento do sistema, mas não nos dá informação sobre o trabalho interno dos processos ativados como resultado desse comportamento. A notação de modelagem que fornece essa informação é discutida na Seção 8.6.4.

CASASEGURA

Modelagem de Fluxo de Dados

A cena: Sala de Jamie, depois de concluída a última reunião de coleta de requisitos.

Os personagens: Jamie, Vinod e Ed — todos membros da equipe de software *CasaSegura*.

A conversa:

(Jamie esboçou os modelos mostrados da Figura 8.9 a 8.12 e os está apresentando para Ed e Vinod.)

Jamie: Eu fiz um curso de engenharia de software na faculdade e nos ensinaram essa matéria. O professor dizia que isso é um pouco antigo, mas você quer saber? Me ajuda a esclarecer as coisas.

Ed: Está bom. Mas, eu não vi qualquer classe ou objeto.

Jamie: Não... isso é justamente um modelo de fluxo com um pouco de comportamento no meio dele.

Vinod: Então esses DFDs representam uma visão E-P-S do software, certo?

Ed: E-P-S?

Vinod: Entrada-Processamento-Saída. Os DFDs são na verdade bastante intuitivos... se você olhá-los por um instante, eles mostram como os objetos de dados fluem através do sistema e se transformam à medida que fluem.

Ed: Veja como nós convertemos todas as bolhas em componentes executáveis... pelo menos no nível mais baixo de DFD.

Jamie: Essa é a parte tranqüila, você pode. De fato há um modo de traduzir os DFDs para uma arquitetura de projeto.

Ed: Verdade?

Jamie: Sim, mas primeiro temos que desenvolver um modelo completo de análise, e este não é.

Vinod: Bem, esse é um primeiro passo, mas vamos ter de cuidar de elementos baseados em classe e também de aspectos de comportamento, embora este diagrama de estado faça algo disso.

Ed: Nós temos muito trabalho a fazer e não temos muito tempo para fazê-lo.

(Doug — o gerente de engenharia de software — entra na sala.)

Doug: Então os próximos dias vão ser gastos desenvolvendo o modelo de análise, não é?

Jamie (parecendo orgulhosa): Nós já começamos.

Doug: Bom, nós temos muito trabalho a fazer e não temos muito tempo para fazê-lo.

(Os três engenheiros de software se entreolham e sorriem.)

8.6.4 A Especificação de Processo

PONTO CHAVE

A PSPEC é uma "miniespecificação" para cada transformação no nível mais baixo de refinamento de um DFD.

A *especificação de processo* (*process specification*, PSPEC) é usada para descrever todos os processos do modelo de fluxo que aparecem no nível de refinamento final. O conteúdo da especificação de processo pode incluir texto narrativo, uma descrição em linguagem de projeto de programa[19] (*program design language*, PDL) do algoritmo do processo, equações matemáticas, tabelas, diagramas ou gráficos. Fornecendo uma PSPEC para acompanhar cada bolha do modelo de fluxo, o engenheiro de software cria uma "miniespecificação" que pode servir como diretriz para o projeto do componente de software que vai implementar o processo.

Para ilustrar o uso da PSPEC, considere a transformação *processar senha* representada no modelo de fluxo de *CasaSegura* (veja a Figura 8.10). A PSPEC para essa função poderia tomar a forma:

> **PSPEC: processar senha (para o painel de controle).** A transformação *processar senha* realiza todas as validações de senha para o painel de controle da função de segurança *CasaSegura*. *Processar senha* recebe uma senha de quatro dígitos da função *interagir como usuário*. Essa é comparada com a senha mestra armazenada no sistema. Se a senha mestra coincide [mensagem de id válida = verdadeiro], é passada para a função *mostrar mensagem e status*. Se a senha mestra não coincide, os quatro dígitos são comparados com uma tabela de senhas secundárias (que podem ser distribuídas a hóspedes e/ou empregados que precisam ter acesso à casa quando o proprietário não está presente). Se a senha coincide com uma entrada da tabela [mensagem id válida = verdadeiro], é passada para a função *mostrar mensagem e status*. Se não há coincidência [mensagem id válida = falso], é passada para a função *mostrar mensagem e status*.

19 A linguagem de projeto de programa (PDL, *Program Design Language*) mistura sintaxe de linguagem de programação com texto narrativo para fornecer um projeto procedural detalhado. PDL será discutida no Capítulo 11.

Se detalhes algorítmicos adicionais são desejados nesse estágio, uma representação em linguagem de projeto de programa pode ser incluída como parte da PSPEC. No entanto, muitos acreditam que a versão PDL deve ser adiada até o início do projeto de componentes.

FERRAMENTAS DE SOFTWARE

Análise Estruturada

Objetivo: Ferramentas de análise estruturada permitem ao engenheiro de software criar modelos de dados, modelos de fluxo e modelos comportamentais, de modo que favoreça a consistência e a verificação de continuidade e facilite a edição e a extensão. Os modelos criados com essas ferramentas fornecem ao engenheiro de software discernimento da representação de análise e ajuda a eliminar erros antes que eles se propaguem no projeto, ou pior, na implementação em si.

Mecânica: Ferramentas dessa categoria usam um "dicionário de dados" como banco de dados central para a descrição de todos os objetos de dados. Uma vez definidas as entradas no dicionário, diagramas entidade/relacionamento podem ser criados e hierarquias de objetos podem ser desenvolvidas. Características da diagramação de fluxo de dados permitem a fácil criação desse modelo gráfico e também fornecem características para a criação de PSPECs e CSPECs. Ferramentas de análise também habilitam o engenheiro de software a criar modelos comportamentais usando diagramas de estado como a notação operativa.

Ferramentas Representativas[20]

AxiomSys, desenvolvida por STG, Inc. (www.stgcase.com), fornece um conjunto completo de ferramentas para análise estruturada incluindo extensões de Hatley-Pirbhai para modelagem de sistemas de tempo real.

MacA&D, *WinA&D* desenvolvida por Excel Software (www.excelsoftware.com), fornece um conjunto de ferramentas simples e baratas para análise e projeto para máquinas Mac e Windows.

MetaCASE Workbench, desenvolvida por MetaCase Consulting (www.metacase.com), é uma metaferramenta usada para definir métodos de análise e projeto (incluindo análise estruturada): seus conceitos, regras, notações e geradores.

System Architect, desenvolvido por Popkin Software (www.popkin.com), fornece ferramentas na ampla faixa de análise e projeto, incluindo ferramentas para modelagem de dados e análise estruturada.

8.7 MODELAGEM BASEADA EM CLASSE

Para onde vamos quando o assunto é desenvolvimento de elementos baseados em classe de um modelo de análise — classes e objetos, atributos, operações, pacotes, modelos CRC e diagramas de colaboração? A seção seguinte apresenta uma série de diretrizes informais que vão ajudar na sua identificação e representação.

8.7.1 Identificação de Classes de Análise

Se você olha em volta de uma sala, há um conjunto de objetos físicos que podem ser facilmente identificados, classificados e definidos (em termos de atributos e operações). Mas quando você "olha em volta" do espaço do problema de uma aplicação de software, os objetos podem ser mais difíceis de compreender.

20 As ferramentas mencionadas aqui não representam uma recomendação mas, em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

Document shared on www.docsity.com

> "O problema realmente difícil é descobrir em primeiro lugar quais são os objetos [classes] corretos."
>
> **Carl Argila**

Podemos começar a identificar classes examinando o enunciado do problema ou (usando a terminologia aplicada anteriormentes neste capítulo) executando uma "análise gramatical" das narrativas de casos de uso ou de processamento do sistema desenvolvidas para o sistema a ser construído. As classes são determinadas sublinhando-se cada substantivo ou cláusula substantiva e colocando-as em uma tabela simples. Sinônimos devem ser anotados. Se a classe é necessária para implementar uma solução, então ela é parte do espaço de solução, caso contrário, se uma classe é necessária apenas para descrever uma solução, ela é parte do espaço do problema. O que devemos procurar quando todos os substantivos tiverem sido isolados? *Classes de análise* manifestam-se por um dos seguintes modos:

? Como fazer classes de análise se manifestarem por si mesmas como elementos do espaço solução?

- *Entidades externas* (por exemplo, outros sistemas, dispositivos e pessoas) que produzem ou consomem informação a ser usada por um sistema baseado em computador.
- *Coisas* (por exemplo, relatórios, figuras, cartas, sinais) que são parte do domínio de informação do problema.
- *Ocorrências ou eventos* (por exemplo, uma transferência de propriedade ou a finalização de uma série de movimentos de robô) que ocorrem dentro do contexto da operação do sistema.
- *Papéis* (por exemplo, gerente, engenheiro, vendedor) desempenhados por pessoas que interagem com o sistema.
- *Unidades organizacionais* (por exemplo, divisão, grupo, equipe) que são relevantes para uma aplicação.
- *Lugares* (por exemplo, piso de fabricação ou plataforma de carregamento) que estabelecem o contexto do problema ou a função global do sistema.
- *Estruturas* (por exemplo, sensores, veículos de quatro rodas, ou computadores) que definem uma classe de objetos ou, no extremo, classes relacionadas de objetos.

Essa categorização é apenas uma das muitas que têm sido propostas na literatura.[21] Por exemplo, Budd [BUD96] sugere uma taxinomia de classes que inclui produtores (fontes) e consumidores (ralos) de dados, gerentes de dados, classes de visão ou observação e classes auxiliares.

É importante notar também o que as classes e os objetos não são. Em geral, uma classe não deve ter nunca um "nome procedural imperativo" [CAS89]. Por exemplo, se os desenvolvedores de software para um sistema de imagens médicas definirem um objeto com o nome **InverterImagem** ou mesmo **InversãoDeImagem**, eles estariam cometendo um erro sutil. A **Imagem** obtida com o software poderia, sem dúvida, ser uma classe (é uma coisa que é parte do domínio da informação). Inversão da imagem é uma operação aplicada à classe. É provável que *inversão()* fosse definida como uma operação da classe **<> Imagem**, mas não seria definida como uma classe separada para conotar "inversão da imagem". Como Cashman [CAS89] declara: "o objetivo da orientação a objetos é encapsular, mas manter ainda separados os dados e as operações sobre os dados".

Para ilustrarmos como classes de análise poderiam ser definidas durante os estágios iniciais da modelagem, voltemos ao exemplo da função de segurança *CasaSegura*. Na Seção 8.6.1, realizamos uma "análise gramatical" em uma narrativa de processamento[22] para a função de segurança. Extraindo os nomes, podemos propor um número de classes potenciais:

21 Outra importante categorização — definindo entidade, fronteira e classes controladoras — é discutida na Seção 8.7.4.

22 É importante notar que essa técnica deve também ser usada para todo caso de uso desenvolvido como parte da atividade de coleta (extração) de requisitos. Ou seja, os casos de uso podem ser analisados gramaticalmente para extrair as potenciais classes de análise.

Classe Potencial	Classificação geral
proprietário	papel ou entidade externa
sensor	entidade externa
painel de controle	entidade externa
instalação	ocorrência
sistema (nome alternativo para sistema de segurança)	coisa
número, tipo	não-objeto, atributos de sensor
senha mestra	coisa
número de telefone	coisa
evento de sensor	ocorrência
alarme sonoro	entidade externa
serviço de monitoração	unidade organizacional ou entidade externa

A lista continuaria até que todos os substantivos na narrativa de processamento tivessem sido considerados. Note que chamamos cada entrada na lista de um objeto potencial. Precisamos considerar cada item mais profundamente antes que uma decisão final seja feita.

> "As classes lutam, algumas triunfam, outras são eliminadas."
>
> **Mao Zedong**

? Como determino se uma classe potencial poderia de fato se tornar uma classe de análise?

Coad e Yourdon [COA91] sugerem seis características de seleção que deveriam ser usadas quando um analista considera cada objeto em potencial para inclusão no modelo de análise:

1. *Informação retida*. A classe potencial será útil durante a análise apenas se a informação sobre ela tiver de ser lembrada para que o sistema possa funcionar.
2. *Serviços necessários*. A classe potencial deve ter um conjunto de operações identificáveis que podem mudar o valor de seus atributos de algum modo.
3. *Atributos múltiplos*. Durante a análise de requisitos, o foco deve ficar na informação "principal"; uma classe com um único atributo pode, na realidade, ser útil durante o projeto, mas é provavelmente melhor representada como atributo de outra classe durante a atividade de análise.
4. *Atributos comuns*. Um conjunto de atributos pode ser definido para a classe potencial e esses atributos se aplicam a todas as instâncias da classe.
5. *Operações comuns*. Um conjunto de operações pode ser definido para a classe potencial e essas operações se aplicam a todas as instâncias da classe.
6. *Requisitos essenciais*. Entidades externas que aparecem no espaço do problema e produzem ou consomem informação essencial para a operação de qualquer solução do sistema serão, quase sempre, definidas como classes no modelo de requisitos.

Para ser considerada uma classe legítima para inclusão no modelo de requisitos, uma classe potencial deve satisfazer a todas (ou a quase todas) essas características. A decisão de inclusão de classes potenciais no modelo de análise é um tanto subjetiva, e uma avaliação posterior pode causar o descarte ou a reinclusão de uma classe. No entanto, o primeiro passo da modelagem baseada em classe é a definição de classes, e decisões (mesmo subjetivas) devem ser tomadas. Com isso em mente, aplicamos as características de seleção à lista de classes potenciais *CasaSegura*:

Document shared on www.docsity.com

Classe potencial	Número da característica que se aplica
proprietário	rejeitado: 1 e 2 falham, embora 6 se aplica
sensor	aceito: todas se aplicam
painel de controle	aceito: todas se aplicam
instalação	rejeitada
sistema (nome alternativo para sistema de segurança)	aceito: todas se aplicam
número, tipo	rejeitado: 3 falhas, atributos de sensor
senha mestra	rejeitado: 3 falhas
número de telefone	rejeitado: 3 falhas
evento de sensor	aceito: todas se aplicam
alarme sonoro	aceito: 2, 3, 4, 5, 6 se aplicam
serviço de monitoração	rejeitado: 1 e 2 falham, embora 6 se aplica

Deve-se notar que (1) a lista precedente não é completa, classes adicionais teriam de ser acrescentadas para completar o modelo; (2) algumas das classes potenciais rejeitadas vão se tornar atributos das que foram aceitas (por exemplo, **número** e **tipo** são atributos de **Sensor**, e **senha mestra** e **número de telefone** podem se tornar atributos de **Sistema**); (3) diferentes enunciados do problema poderiam causar a implementação de diferentes decisões "aceitar ou rejeitar" (por exemplo, se cada proprietário tivesse uma senha individual ou fosse identificado pela impressão vocal, a classe **Proprietário** satisfaria as características 1 e 2 e teria sido aceita).

PONTO CHAVE

Atributos são o conjunto de objetos de dados que definem completamente uma classe no contexto do problema.

8.7.2 Especificação de Atributos

Atributos descrevem uma classe que foi selecionada para inclusão no modelo de análise. Em essência, são os atributos que definem a classe — que esclarecem o que a classe representa no contexto do espaço do problema.

Para desenvolver um conjunto de atributos significativo para uma classe de análise, um engenheiro de software pode novamente estudar o caso de uso e selecionar as "coisas" que razoavelmente "pertencem" à classe. Além disso, as seguintes questões devem ser respondidas para cada classe: "Que itens de dados (compostos e/ou elementares) descrevem plenamente essa classe no contexto do problema em mãos?".

Para ilustrarmos, consideremos a classe **Sistema** definida para *CasaSegura*. Mencionamos anteriormente neste livro que o proprietário pode configurar a função de segurança para refletir a informação de sensores, informação de reação a alarme, informação de ativação/desativação, informação de identificação e assim por diante. Podemos representar esses itens de dados compostos do seguinte modo:

informação de identificação = ID sistema + número de telefone de verificação + estado do sistema

informação de reação ao alarme = tempo de retardo + número de telefone

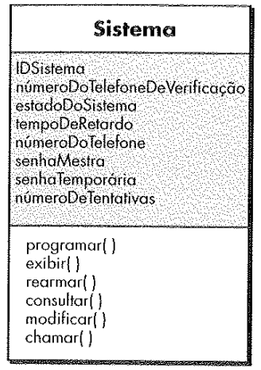
informação de ativação/desativação = senha mestra + número permitido de tentativas + senha temporária

Alguns itens de dados à direita do sinal de igual poderiam ser mais refinados até o nível elementar, mas, para nossa finalidade, eles constituem uma lista de atributos razoável para a classe **Sistema** (parte sombreada da Figura 8.13).

Sensores fazem parte do sistema global *CasaSegura* e, no entanto, não são listados como itens de dados ou como atributos na Figura 8.13. **Sensor** já foi definido como uma classe, e múltiplos objetos **Sensor** serão associados com a classe **Sistema**. Em geral, evitamos definir um item como atributo, se mais de um dos itens estão associados com as classes.

FIGURA 8.13

Diagrama de classe para a classe Sistema



AVISO

Quando você definir operações para uma classe de análise, concentre-se no comportamento orientado ao problema em vez de nos comportamentos necessários para implementação.

8.7.3 Definição das Operações

As operações definem o comportamento de um objeto. Apesar de existirem muitos tipos diferentes de operação, elas podem ser divididas em geral em quatro amplas categorias: (1) operações que de algum modo manipulam dados (somar, excluir, reformatar, selecionar), (2) operações que realizam um cálculo, (3) operações que pesquisam o estado de um objeto, e (4) operações que monitoram um objeto quanto à ocorrência de um evento de controle. Essas funções são realizadas pela operação sobre os atributos e/ou associações (Seção 8.7.5). Assim sendo, uma operação deve ter "conhecimento" da natureza dos atributos e associações da classe.

Como primeira iteração para derivar um conjunto de operações para uma classe de análise, o analista pode novamente estudar a narrativa de processamento (ou caso de uso) e selecionar aquelas operações que razoavelmente pertencem à classe. Para conseguir isso, a análise gramatical é novamente estudada e os verbos são isolados. Alguns desses verbos serão operações legítimas e podem ser facilmente conectados a uma classe específica. Por exemplo, para a narrativa de processamento de *CasaSegura* apresentada anteriormente neste capítulo, vemos que "ao sensor é atribuído um número e tipo" ou que "uma senha mestra é programada para armar e desarmar o sistema". Essas duas frases indicam algumas coisas:

- Que uma operação *atribuir()* é relevante para a classe **Sensor**.
- Que uma operação *programar()* será aplicada à classe **Sistema**.
- Que *armar()* e *desarmar()* são operações que se aplicam à classe **Sistema**.

Diante de mais investigações, é provável que a operação *programar()* seja dividida em algumas suboperações mais específicas necessárias para configurar o sistema. Por exemplo, *programar()* implica especificar números de telefone, configurar características do sistema (por exemplo, criando tabelas de sensores, introduzindo características do alarme) e introduzir senha(s); por enquanto, especificamos *programar()* como uma única operação.

CASASEGURA

Modelos de Classe

A cena: A sala de Ed, no começo da modelagem de análise.

Os personagens: Jamie, Vinod e Ed — todos membros da equipe de software *CasaSegura*.

A conversa:

(Ed tem estado trabalhando para extrair classes do gabarito de caso de uso de **Ter acesso à câmera de vigilância — exibir visões da câmera** [apresentado em quadro anterior neste capítulo] e está apresentando as classes que ele extraiu para seus colegas.)

Document shared on www.docsity.com

Ed: Quando o proprietário deseja selecionar uma câmera, ele deve selecioná-la pela planta baixa. Eu defini uma classe **PlantaBaixa**. Aqui está o diagrama.

(Eles olham a Figura 8.14.)

Jamie: Então, **PlantaBaixa** é uma classe que é colocada com as paredes que compõem os segmentos de paredes, portas e janelas, e também câmeras; isto é o que aquelas linhas rotuladas significam, não é?

Ed: Certo, elas são chamadas de "associações". Uma classe está associada a outra de acordo com as associações que eu mostrei. [Associações são discutidas na Seção 8.7.5.]

Vinod: Então, a planta baixa real é constituída de paredes e contém câmeras e sensores que estão colocados nessas paredes. Como a planta baixa sabe onde colocar esses objetos?

Ed: Ela não sabe, mas as outras classes sabem. Veja os atributos, por exemplo, de **SegmentoDeParede**, que é usado para construir uma parede. O segmento de parede tem coordenadas de início e fim e a operação *desenhar()* faz o resto.

Jamie: E o mesmo acontece com as janelas e portas. Veja como câmera tem alguns atributos extra.

Ed: Está bem, eu preciso deles para fornecer as informações de pan e zoom.

Vinod: Eu tenho uma questão. Por que a câmera tem uma ID e as outras coisas não têm?

Ed: Nós teremos que identificar cada câmera para fins de exibição.

Jamie: Isso faz sentido para mim, mas tenho algumas outras questões.

(Jamie faz perguntas que resultam em pequenas modificações.)

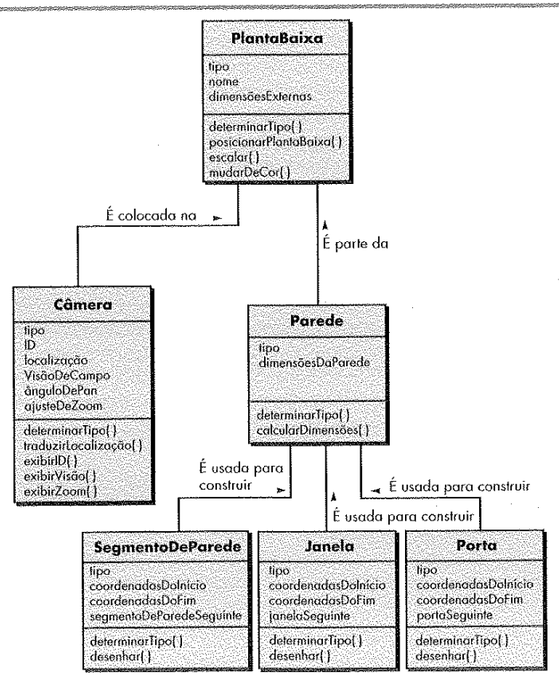
Vinod: Você tem cartões CRC para cada uma das classes? Se tem, nós temos que verificá-los, só para ter certeza de que nada foi omitido.

Ed: Eu não estou certo de como fazê-lo.

Vinod: Isso não é difícil e realmente vale a pena. Vou lhes mostrar.

FIGURA 8.14

Diagrama de classe de PlantaBaixa (veja a discussão no quadro anterior)



8.7.4 Modelagem Classe-Responsabilidade-Colaboração (CRC)

A modelagem *classe-responsabilidade-colaboração* (*class-responsibility-collaborator*, CRC) [WIR90] fornece um meio simples para identificar e organizar as classes relevantes aos requisitos do sistema ou produto. Ambler [AMB95] descreve a modelagem CRC do seguinte modo:

> Um modelo CRC é na realidade uma coleção de cartões-padrão de indexação que representam as classes. Os cartões são divididos em três seções. No alto do cartão, você escreve o nome da classe. No corpo do cartão você lista as responsabilidades da classe à esquerda e os colaboradores à direita.

Na verdade, o modelo CRC pode fazer uso de cartões de indexação reais ou virtuais. O objetivo é desenvolver uma representação organizada de classes. *Responsabilidades* são os atributos e operações relevantes à classe. Dito simplesmente, uma responsabilidade é "qualquer coisa que a classe sabe ou faz" [AMB95]. *Colaboradores* são as classes necessárias para dar à classe a informação necessária para completar uma responsabilidade. Em geral, uma colaboração implica tanto solicitação de informação como solicitação de alguma ação.

> "Uma finalidade dos cartões CRC é falhar cedo, freqüentemente e de maneira não dispendiosa. É muito mais barato rasgar um monte de cartões do que reorganizar uma grande quantidade de código-fonte."
>
> C. Horstmann

Um cartão CRC simples para a classe **PlantaBaixa** é ilustrado na Figura 8.15. A lista de responsabilidades mostrada no cartão CRC é preliminar e sujeita a adições ou modificações. As classes **Parede** e **Câmera** são posicionadas próximo das responsabilidades que vão requerer sua colaboração.

Classes. As diretrizes básicas para identificação de classes e objetos foram apresentadas anteriormente neste capítulo. A taxinomia de tipos de classe apresentada na Seção 8.7.1 pode ser estendida considerando as seguintes categorias:

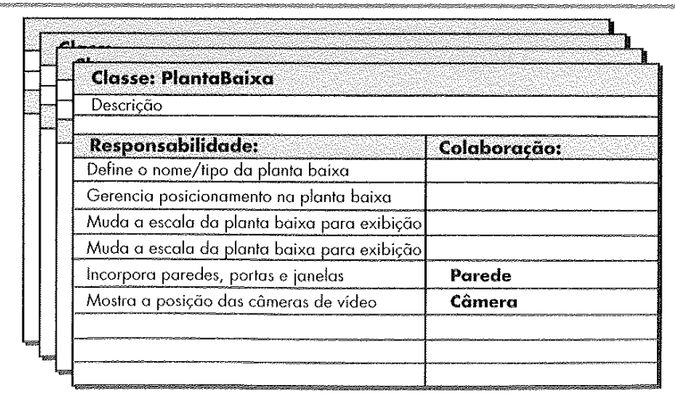
Veja na Web

Uma excelente discussão desses tipos de classe pode ser encontrada em www.theumlcafe.com/a0079.htm.

- *Classes de entidade*, também chamadas de *classes de negócio ou do modelo*, são extraídas diretamente do enunciado do problema (por exemplo, **PlantaBaixa** e **Sensor**). Essas classes representam tipicamente coisas que devem ser armazenadas em um banco de dados e persistem ao longo da duração da aplicação (a menos que sejam especificamente removidas).
- *Classes de fronteira* são usadas para criar a interface (por exemplo, telas interativas ou relatórios impressos) que o usuário vê e com a qual interage enquanto o software é usado. Classes de entidade contêm informações importantes para os usuários, mas elas próprias não são exibidas. As classes de fronteira são projetadas com a responsabilidade de gerir o

FIGURA 8.15

Um modelo de cartão CRC



Document shared on www.docsity.com

modo pelo qual os objetos da entidade são representados para os usuários. Por exemplo, uma classe de fronteira chamada de **JanelaDeCâmera** teria a responsabilidade de exibir a saída da câmera de vigilância para o sistema *CasaSegura*.

- *Classes controladoras* gerenciam a "unidade de trabalho" [UML03] do começo ao fim. Classes controladoras podem ser projetadas para gerir (1) a criação ou atualização de objetos da entidade; (2) a instanciação de objetos-fronteira à medida que eles obtêm informação dos objetos da entidade; (3) a comunicação complexa entre conjuntos de objetos; e (4) a validação dos dados comunicados entre objetos ou entre o usuário e a aplicação. Em geral, classes controladoras não são consideradas até que o projeto tenha começado.

> "Objetos podem ser classificados cientificamente em três categorias principais: aqueles que não funcionam, aqueles que quebram e aqueles que são perdidos."
>
> **Russel Baker**

Responsabilidades. Diretrizes básicas para a identificação de responsabilidades (atributos e operações) também foram apresentadas nas seções 8.7.2 e 8.7.3. Wirfs-Brock e seus colegas [WIR90] sugerem cinco diretrizes para atribuir responsabilidades a classes:

1. **A inteligência do sistema deve ser distribuída pelas classes para melhor atender às necessidades do problema.** Cada aplicação engloba um certo grau de inteligência; ou seja, o que o sistema sabe e o que pode fazer. Essa inteligência pode ser distribuída pelas classes de diferentes maneiras. Classes "burras" (aquelas que têm poucas responsabilidades) podem ser modeladas para agir como servidoras de algumas classes "espertas" (aquelas que têm muitas responsabilidades). Apesar de essa abordagem tornar o fluxo de controle direto no sistema, tem algumas desvantagens: (a) concentra toda a inteligência em umas poucas classes, tornando as modificações mais difíceis e (b) tende a exigir mais classes, conseqüentemente, mais esforço de desenvolvimento.

   Se a inteligência do sistema é mais uniformemente distribuída entre as classes, em uma aplicação, cada objeto tem conhecimento disso e faz apenas algumas coisas (que são geralmente bem focalizadas), e a conexidade do sistema é aumentada. Isso melhora a manutenibilidade do software e reduz o impacto de efeitos colaterais devido a modificações.

   Para determinar se a inteligência do sistema está distribuída adequadamente, as responsabilidades anotadas em cada cartão de indexação do modelo CRC devem ser avaliadas, para determinar se alguma classe tem uma lista de responsabilidades extraordinariamente longa. Isso indica uma concentração de inteligência.[23] Além disso, as responsabilidades de cada classe devem exibir o mesmo nível de abstração.
2. **Cada responsabilidade deve ser enunciada tão genericamente quanto possível.** Essa diretriz implica que as responsabilidades gerais (tanto atributos quanto operações) devem ficar no alto da hierarquia de classes (porque, sendo genéricas, serão aplicadas a todas as subclasses).
3. **A informação e o comportamento relacionado devem residir dentro da mesma classe.** Isso realiza o princípio OO que chamamos de *encapsulamento*. Os dados e os processos que manipulam os dados devem ser empacotados como uma unidade coesiva.
4. **A informação sobre uma coisa deve ser localizada em uma única classe, não distribuída entre várias classes.** Uma única classe deve assumir a responsabilidade de guardar e manipular um tipo específico de informação. Essa responsabilidade não deve, em geral, ser compartilhada por várias classes. Se a informação é distribuída, fica mais difícil manter e mais complicado testar o software.
5. **Responsabilidades devem ser compartilhadas por classes relacionadas, quando adequado.** Há muitos casos nos quais diversos objetos relacionados devem exibir o mesmo comportamento ao mesmo tempo. Como exemplo, considere um videogame que deve

? Que diretrizes podem ser aplicadas para alocar responsabilidades a classes?

23 Em tais casos, pode ser necessário subdividir a classe em várias classes ou subsistemas completos a fim de distribuir a inteligência mais efetivamente.

exibir os seguintes objetos: **Jogador**, **CorpoDoJogador**, **BraçosDoJogador**, **PernasDoJogador** e **CabeçaDoJogador**. Cada um desses objetos tem seus próprios atributos (**posição, orientação, cor, velocidade**) e todos precisam ser atualizados e mostrados, à medida que o usuário manipula um *joystick*. As responsabilidades *atualize()* e *mostre()* devem, conseqüentemente, ser compartilhadas por cada um dos objetos mencionados. **Jogador** sabe quando alguma coisa foi modificada e *atualize()* é necessária. Ele colabora com os outros objetos para atingir uma nova posição ou orientação, mas cada objeto controla sua própria exibição.

Colaborações. Classes satisfazem às suas responsabilidades em um de dois modos: (1) uma classe pode usar suas próprias operações para manipular seus próprios atributos, satisfazendo assim uma responsabilidade específica, ou (2) uma classe pode colaborar com outras classes.

Wirks-Brock e seus colegas [WIR90] definem colaborações do seguinte modo:

> Colaborações representam solicitações de um cliente para um servidor, para satisfazer às responsabilidades do cliente. Uma colaboração é a realização do contrato entre o cliente e o servidor... Dizemos que um objeto colabora com outro objeto se, para satisfazer a uma responsabilidade, ele precisa mandar qualquer mensagem a outro objeto. Uma única colaboração flui em uma direção — representando a solicitação do cliente para o servidor. Do ponto de vista do cliente, cada uma de suas colaborações está associada com uma responsabilidade específica, implementada pelo servidor.

PONTO CHAVE

Se uma classe não pode cumprir todas as suas obrigações por si mesma, então uma colaboração é necessária.

As colaborações identificam relações entre classes. Quando, em um conjunto de classes, todas colaboram para satisfazer a alguma solicitação, elas podem ser organizadas em um subsistema (assunto de projeto).

Colaborações são identificadas determinando se uma classe pode satisfazer a cada responsabilidade por si mesma. Se não pode, então precisa interagir com outra classe. Daí a colaboração.

Como exemplo, considere a função de segurança *CasaSegura*. Como parte do procedimento de ativação, o objeto **PainelDeControle** precisa determinar se algum sensor está aberto. Uma responsabilidade denominada *determina-estado-sensor()* é definida. Se há sensores abertos, **PainelDeControle** precisa ajustar um atributo de **status** para "não pronto". A informação de sensor pode ser adquirida de cada objeto **Sensor**. Conseqüentemente, a responsabilidade *determina-estado-sensor()* pode ser satisfeita apenas se **PainelDeControle** trabalha em colaboração com **Sensor**.

Para ajudar na identificação dos colaboradores, o analista pode examinar três diferentes relacionamentos genéricos entre classes [WIR90]: (1) o relacionamento *é-parte-de*, (2) o relacionamento *tem-conhecimento-de* e (3) o relacionamento *depende-de*. Cada um dos três relacionamentos genéricos é considerado resumidamente nos parágrafos que se seguem.

Todas as classes que fazem parte de uma classe agregada são conectadas a essa por meio de um relacionamento *é-parte-de*. Considere as classes definidas para o videogame mencionado anteriormente, a classe **CorpoDoJogador** *é-parte-de* **Jogador**, assim como **BraçosDoJogador**, **PernasDoJogador** e **CabeçaDoJogador**. Em UML, esses relacionamentos são representados como a agregação mostrada na Figura 8.16.

Quando uma classe precisa adquirir informação de outra classe, o relacionamento *tem-conhecimento-de* é estabelecido. A responsabilidade *determina-estado-sensor()*, mencionada anteriormente, é um exemplo de relacionamento *tem-conhecimento-de*.

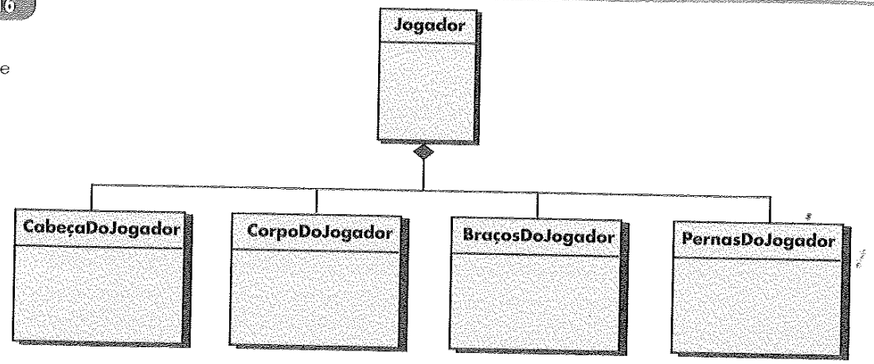
O relacionamento *depende-de* significa que duas classes têm uma dependência que não é conseguida por *tem-conhecimento-de* ou *é-parte-de*. Por exemplo, **CabeçaDoJogador** precisa sempre estar conectada a **CorpoDoJogador** (a menos que o videogame seja particularmente violento), no entanto, cada objeto poderia existir sem conhecimento direto do outro. Um atributo do objeto **CabeçaDoJogador** chamado de **posição-centro** é determinado pela posição do centro de **CorpoDoJogador**. Essa informação é obtida por intermédio de um terceiro objeto, **Jogador**, que a adquire de **CorpoDoJogador**. Assim, **CabeçaDoJogador** *depende-de* **CorpoDoJogador**.

Em todos os casos, o nome da classe colaboradora é registrado no cartão de indexação do modelo CRC junto da responsabilidade que provocou a colaboração. Assim, o cartão de indexação contém uma lista de responsabilidades e das correspondentes colaborações que permitem satisfazer às responsabilidades (veja a Figura 8.15).

Document shared on www.docsity.com

FIGURA 8.16

Uma classe composta de agregação



Quando um modelo completo CRC tiver sido desenvolvido, os representantes do cliente e das organizações de engenharia de software podem revisar o modelo usando a seguinte abordagem [AMB95]:

1. A todos os participantes da revisão (do modelo CRC) são dados subconjuntos das fichas de indexação do modelo CRC. As fichas que colaboram devem ser separadas (nenhum revisor deve ter duas fichas que colaboram).
2. Todos os cenários de casos de uso (e correspondentes diagramas de casos de uso) devem ser organizados em categorias.
3. O líder da revisão lê o caso de uso pausadamente. À medida que ele chega a um objeto especificado, passa a vez para a pessoa que está com a correspondente ficha de indexação.

   Por exemplo, um caso de uso de *CasaSegura* contém a seguinte narrativa:

   > O proprietário observa o painel de controle de *CasaSegura* para determinar se o sistema está pronto para operar. Se não está, o proprietário precisa fechar fisicamente as janelas/portas de modo que o indicador de pronto fique ligado. [Um indicador não-pronto implica que um sensor está aberto, isto é, que uma porta ou janela está aberta.]

   Quando o líder da revisão chega a "painel de controle", na narrativa do caso de uso, a vez é passada para a pessoa que está com a ficha de indexação **PainelDeControle**. A frase "implica que um sensor está aberto" exige que a ficha de indexação contenha uma responsabilidade que valide essa implicação (a responsabilidade *determinar-estado-sensor()* realiza isso). Junto da responsabilidade, na ficha de indexação, está o colaborador **Sensor**. A vez é então passada para a classe **Sensor**.
4. Quando a vez é passada, quem está de posse da ficha da classe deve descrever as responsabilidades anotadas na ficha. O grupo determina se uma (ou mais) das responsabilidades satisfaz à exigência do caso de uso.
5. Se as responsabilidades e colaborações anotadas nas fichas de indexação não podem acomodar o caso de uso, são feitas modificações nas fichas. Isso pode incluir a definição de novas classes (e correspondentes fichas de indexação CRC), ou a especificação de responsabilidades, ou colaborações novas ou revisadas nas fichas existentes.

Esse modo de agir continua até que o caso de uso seja finalizado. Quando todos os casos de uso tiverem sido revisados, a modelagem de análise continua.

CASASEGURA

Modelos CRC

A cena: A sala de Ed, enquanto a modelagem de análise continua.

Os personagens: Vinod e Ed — membros da equipe de engenharia de software de *CasaSegura*.

A conversa:

(Vinod decidiu explicar a Ed como desenvolver cartões CRC mostrando-lhe um exemplo.)

Vinod: Enquanto você estava trabalhando na vigilância e Jamie ficou preso na segurança, eu estive trabalhado na função de gestão da casa.

Ed: Em que estado isso está? Marketing continuou mudando suas idéias.

Vinod: Aqui está o primeiro esboço de caso de uso para toda a função... nós temos que refiná-lo um pouco, mas isso deve dar a você uma visão geral.

Caso de Uso: Função de gestão da casa para o *CasaSegura*.

Narrativa: Desejamos usar a interface de gestão da casa em um PC ou em uma conexão Internet para controlar dispositivos eletrônicos que têm controladores de interface sem fio. O sistema deveria me permitir ligar e desligar luzes específicas, controlar eletrodomésticos que estão conectados a uma interface sem fio, ajustar meu sistema de aquecimento e ar-condicionado a temperaturas que eu defina. Para fazer isso, quero selecionar os dispositivos por uma planta baixa da casa. Cada dispositivo deve ser identificado na planta baixa. Como característica opcional, quero controlar todos os dispositivos audiovisuais — áudio, televisão, DVD, gravadores digitais etc.

Com uma única seleção, quero ser capaz de ajustar a casa toda para várias situações: *em casa, fora dela, viagem de um dia* e *viagem prolongada*. Todas essas situações terão ajustes aplicados a todos os dispositivos. Nos estados *viagem de um dia* e *viagem prolongada*, o sistema deve ligar e desligar as luzes em intervalos aleatórios (para dar a impressão de que alguém está em casa) e controlar o sistema de aquecimento e ar-condicionado. Eu deveria poder refazer esses ajustes, via Internet, com uma senha de proteção adequada.

Ed: O pessoal de hardware já está com todas as interfaces sem fio planejadas?

Vinod (sorrindo): Eles estão trabalhando nisso, dizem que não é muito difícil. De qualquer modo, extraí um monte de classes para a gestão da casa e nós podemos usar uma como um exemplo. Vamos usar a classe **GestãoDeInterfaceDaCasa**.

Ed: Está bem, então as responsabilidades são os atributos e operações da classe, e as colaborações são as classes para as quais as responsabilidades apontam.

Vinod: Achei que você não entendia de CRC.

Ed: Um pouco talvez, mas vamos em frente.

Vinod: Então, aqui está a minha definição de classe para **GestãoDeInterfaceDaCasa**.

Atributos:

painelDeOpções — fornece informações sobre os botões que possibilitam ao usuário selecionar a funcionalidade.

painelDeSituação — fornece informações sobre os botões que possibilitam ao usuário selecionar a situação.

PlantaBaixa — o mesmo que objeto de vigilância, mas este exibe os dispositivos.

íconesDeDispositivos — informação sobre ícones que representam luzes, eletrodomésticos, HVAC etc.

painéisDeDispositivo — simulação de eletrodoméstico ou painel de controle de dispositivo; permite o controle.

Operações:

exibirControlador(), selecionarControlador(), exibirSituação(), selecionarSituação(), terAcessoAPlantaBaixa(), selecionarIconeDeDispositivo(), exibirPainelDeDispositivo(), terAcessoAPainelDeDispositivo(), ...

Classe: GestãoDeInterfaceDaCasa

Responsabilidade	**Colaborador**
exibirControlador()	**PainelDeOpções**(classe)
selecionarControlador()	**PainelDeOpções**(classe)
exibirSituação()	**PainelDeSituação**(classe)
selecionarSituação()	**PainelDeSituação**(classe)
terAcessoAPlantaBaixa()	**PlantaBaixa**(classe). . .
•	
•	
•	

Ed: Então, quando a operação *terAcessoAPlantaBaixa()* é invocada, ela colabora com o objeto **PlantaBaixa** de modo igual ao que desenvolvemos para vigilância. Espere, eu tenho uma descrição disso aqui. (Eles olham para a Figura 8.14).

Vinod: Exatamente. E se nós quiséssemos revisar todo o modelo de classe, poderíamos começar com esse cartão indexado, depois ir para a cartão indexado do colaborador, de lá para um dos colaboradores do colaborador e assim por diante.

Ed: Bom modo de encontrar omissões ou erros.

Vinod: Está certo.

Document shared on www.docsity.com

8.7.5 Associações e Dependências

PONTO CHAVE

Uma associação define um relacionamento entre classes. Multiplicidade define quantos de uma classe estão relacionados com quantos de outra classe.

Em muitas instâncias, duas classes de análise estão relacionadas entre si de algum modo, como dois objetos de dados podem estar relacionados entre si (Seção 8.3.3). Em UML esses relacionamentos são chamados de *associações*. Retornando à Figura 8.14, a classe **PlantaBaixa** é definida pela identificação de um conjunto de associações entre **PlantaBaixa** e duas outras classes, **Câmera** e **Parede**. A classe **Parede** está associada a três classes que permitem que uma parede seja construída, **SegmentoDeParede**, **Janela** e **Porta**.

Em alguns casos, uma associação pode ser melhor definida pela indicação da *multiplicidade* (o termo *cardinalidade* foi usado anteriomente neste capítulo). Retornando à Figura 8.14, um objeto **Parede** é construído de um ou mais objetos **SegmentoDeParede**. Além disso, o objeto **Parede** pode conter 0 ou mais objetos **Janela** e 0 ou mais objetos **Porta**. Essas restrições de multiplicidade são ilustradas na Figura 8.17, onde "um ou mais" é representado usando 1..* e "0 ou mais" por 0..*. Em UML, o asterisco indica um extremo superior não limitado para o intervalo.[24]

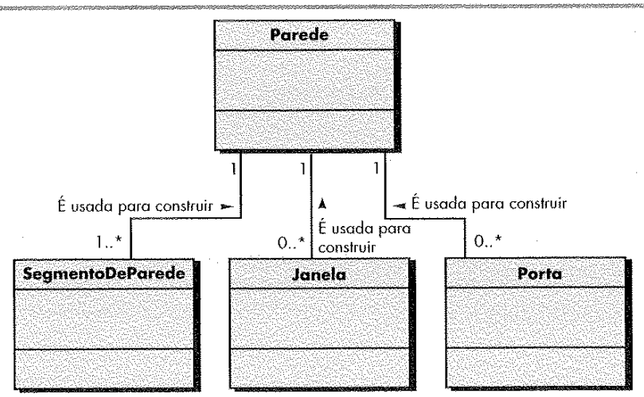
? O que é um estereótipo?

Em muitas instâncias, existe um relacionamento cliente-servidor entre duas classes de análise. Em tais casos, uma classe-cliente depende da classe-servidor de algum modo, e um *relacionamento de dependência* é estabelecido. Dependências são definidas por um estereótipo. *Estereótipo* é um "mecanismo de extensibilidade" [ARL02] na UML que permite a um engenheiro de software definir um elemento especial de modelagem cujas semânticas são definidas pelo cliente. Em UML, estereótipos são representados por sinais duplos de maior e menor (<<estereótipo>>).

Como ilustração de uma simples dependência no sistema de vigilância *CasaSegura*, um objeto **Câmera** (neste caso, a classe-servidor) fornece uma imagem de vídeo para um objeto **JanelaExibição** (neste caso, a classe-cliente). O relacionamento entre esses dois objetos não é uma associação simples, de fato existe uma associação de dependência. Em um caso de uso escrito para vigilância (não mostrado), o modelador descobre que uma senha especial deve ser fornecida para ver localizações específicas de câmera. Um modo de conseguir isso é fazer **Câmera** exigir uma senha e depois dar permissão a **JanelaExibição** para produzir a saída de vídeo. Isso pode ser representado como mostra a Figura 8.18, em que <<acesso>> implica que o uso de saída da câmera é controlado por uma senha especial.

FIGURA 8.17

Multiplicidade



8.7.6 Pacotes de Análise

Uma parte importante da modelagem de análise é a categorização. Vários elementos do modelo de análise (por exemplo, casos de uso, classes de análise) são categorizados de um modo que são empacotados como um agrupamento — chamado de *pacote de análise* — que recebe um nome representativo.

24 Outras relações de multiplicidade — um para um, um para muitos, muitos para muitos, um para um intervalo específico com limitantes inferior e superior, e outros — podem ser indicados como parte de uma associação.

FIGURA 8.18

Dependências

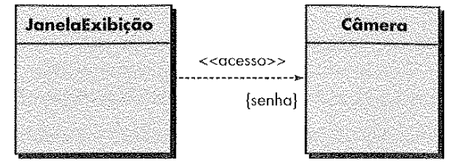
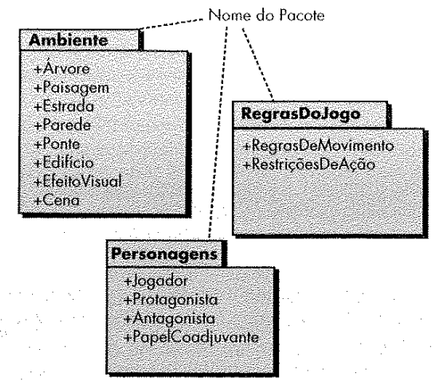


FIGURA 8.19

Pacotes



PONTO CHAVE

Um pacote é usado para montar uma coleção de classes relacionadas.

Para ilustrar o uso de pacotes de análise considere o videogame que introduzimos anteriormente. À medida que o modelo de análise para o videogame é desenvolvido, um grande número de classes é derivado. Algumas focalizam o ambiente do jogo — as cenas visuais que o usuário vê quando o jogo está em curso. Classes como **Árvore**, **Paisagem**, **Estrada**, **Parede**, **Ponte**, **Edifício**, **EfeitoVisual** poderiam cair nessa categoria. Outras focalizam os personagens do jogo, descrevendo suas características físicas, ações e restrições. Classes como **Jogador** (descrito anteriormente), **Protagonista**, **Antagonista** e **PapéisCoadjuvantes** poderiam ser definidas. Outras ainda descrevem as regras do jogo — como um jogador navega pelo ambiente. Classes como **RegrasDeMovimento** e **Restrições DeAção** são candidatas a isso. Muitas outras categorias podem existir. Essas classes podem ser representadas como pacotes de análise, o que mostra a Figura 8.19.

O sinal de adição precedendo o nome da classe de análise em cada pacote indica que as classes têm visibilidade pública e são, portanto, acessíveis por meio de outros pacotes. Embora eles não sejam mostrados na figura, outros símbolos podem preceder um elemento em um pacote. Um sinal de subtração indica que o elemento está oculto de todos os outros pacotes, e um símbolo # indica que o elemento é acessível somente para as classes contidas em um certo pacote.

8.8 Criação de um Modelo Comportamental

? Como fazer o modelo de reação do software a algum evento externo?

Diagramas de classe, cartões indexados CRC e outros modelos orientados à classe discutidos na Seção 8.7 representam elementos estáticos do modelo de análise. Chegou a hora de fazer uma transição para o comportamento dinâmico do sistema ou produto. Para tanto, precisamos representar o comportamento do sistema como uma função de eventos e tempo específicos.

O *modelo comportamental* indica como o software responderá a eventos ou estímulos externos. Para criar um modelo, o analista precisa executar os seguintes passos:

1. Avaliar todos os casos de uso para entender plenamente a seqüência de interação dentro do sistema.

Document shared on www.docsity.com

PONTO CHAVE

Casos de uso são analisados para definir eventos. Para conseguir isso, o caso de uso é examinado quanto a pontos de troca de informação.

2. Identificar os eventos que dirigem a seqüência de interação e entender como esses eventos se relacionam a classes específicas.
3. Criar uma seqüência para cada caso de uso.
4. Construir um diagrama de estados para o sistema.
5. Revisar o modelo comportamental para verificar a precisão e a consistência.

Cada um desses passos é discutido nas seções seguintes.

8.8.1 Identificação dos Eventos no Caso de Uso

Como mencionamos na Seção 8.5, o caso de uso representa uma seqüência de atividades que envolve atores e o sistema. Em geral, um evento ocorre sempre que um sistema e um ator trocam informação. Lembrando a nossa discussão anterior de modelagem comportamental na Seção 8.6.3, é importante notar que um evento não é a informação que foi trocada, mas sim o *fato* de que a informação foi trocada.

Um caso de uso é examinado para encontrar pontos de troca de informação. Para ilustrarmos, reconsideremos o caso de uso para uma pequena parte da função de segurança de *CasaSegura*.

> O proprietário usa o teclado para digitar uma senha de quatro dígitos. A senha é comparada com a senha válida guardada no sistema. Se a senha está incorreta, o painel de controle vai emitir um bip uma vez e reajustar-se para entrada adicional. Se a senha está correta, o painel de controle espera a próxima ação.

As partes sublinhadas do cenário de caso de uso indicam eventos. Um ator deve ser identificado para cada evento; a informação que é trocada deve ser registrada e quaisquer condições ou restrições devem ser listadas.

Como exemplo de um evento típico, considere a frase sublinhada do caso de uso "proprietário usa o teclado para digitar uma senha de quatro dígitos". No contexto do modelo de análise OO, o objeto **Proprietário**,[25] transmite um evento para o objeto **PainelDeControle**. O evento pode ser denominado *senha entrada*. A informação transferida são os quatro dígitos que constituem a senha, mas isso não é parte essencial do modelo comportamental. É importante notar que alguns eventos têm um impacto explícito no fluxo de controle do caso de uso, enquanto outros não têm impacto direto no fluxo de controle. Por exemplo, o evento *senha entrada* não modifica explicitamente o fluxo de controle do caso de uso, mas os resultados do evento *comparação de senha* (derivado da interação "a senha digitada é comparada com a senha válida guardada no sistema") terão um impacto explícito no fluxo de informação e controle do software *CasaSegura*.

Uma vez identificados todos os eventos, eles são associados aos objetos envolvidos. Objetos podem ser responsáveis pela geração de eventos (**Proprietário** gera o evento *senha entrada*) ou pelo reconhecimento de eventos que ocorreram em outro lugar (**PainelDeControle** reconhece o resultado binário do evento *comparação de senha*).

PONTO CHAVE

O sistema tem estados que representam comportamento específico externamente observável; uma classe tem estados que representam seu comportamento à medida que o sistema realiza suas funções.

8.8.2 Representações de Estados

No contexto de modelagem comportamental, devem ser consideradas duas caracterizações de estados diferentes: (1) o estado de cada classe, à medida que o sistema realiza sua função, e (2) o estado do sistema, observado de fora, à medida que o sistema realiza sua função.[26]

O estado de um objeto assume características tanto passivas quanto ativas [CHA93]. Um *estado passivo* é simplesmente o estado atual de todos os atributos de um objeto. Por exemplo, o estado passivo da classe **Jogador** (na aplicação de videogame discutida anteriormente) incluiria os atributos atuais **posição** e **orientação** de **Jogador**, bem como outras características de **Jogador** que são relevantes ao jogo (por exemplo, um atributo que indica **permanência de desejos mágicos**). O *estado ativo* de um objeto indica o estado atual do objeto à medida que ele passa por uma transformação ou processamento contínuo. A classe **Jogador** pode ter os seguintes estados ativos: *em movimento, em repouso, machucado, sendo tratado, preso, perdido* etc. Um evento (algumas vezes chamado de *disparo*) precisa ocorrer para forçar o objeto a realizar a transição de um estado ativo para outro.

Duas diferentes representações comportamentais são discutidas nos parágrafos que se seguem. A primeira indica como uma classe individual muda de estado com base nos eventos externos, e a segunda mostra o comportamento do software como função do tempo.

Diagramas de estado para classes de análise. Um componente do modelo comportamental é um diagrama de estado UML que representa os estados ativos de cada classe e os eventos (disparos) que causam mudanças entre esses estados ativos. A Figura 8.20 mostra o diagrama de estado para a classe **PainelDeControle** da função de segurança *CasaSegura*.

Cada seta mostrada na Figura 8.20 representa uma transição de um estado ativo de uma classe para outra. Os rótulos em cada seta representam o evento que dispara a transição. Apesar de o modelo de estado ativo fornecer conhecimento útil sobre o "histórico de vida" de uma classe, é possível especificar informação adicional para dar mais profundidade ao entendimento do comportamento de uma classe. Além de especificar o evento que faz a transição ocorrer, o analista pode especificar uma guarda e uma ação [CHA93]. Uma *guarda* é uma condição booleana que deve ser satisfeita para que a transição ocorra. Por exemplo, a guarda para a transição do estado "lendo" para o estado "comparando", na Figura 8.20, pode ser determinada pelo exame do caso de uso:

se (entrada da senha = 4 dígitos) então compare com a senha armazenada

PONTO CHAVE

Diferente de um diagrama de estado, que representa o comportamento sem indicar as classes envolvidas, um diagrama de seqüência representa o comportamento pela descrição de como as classes se movem de um estado para outro.

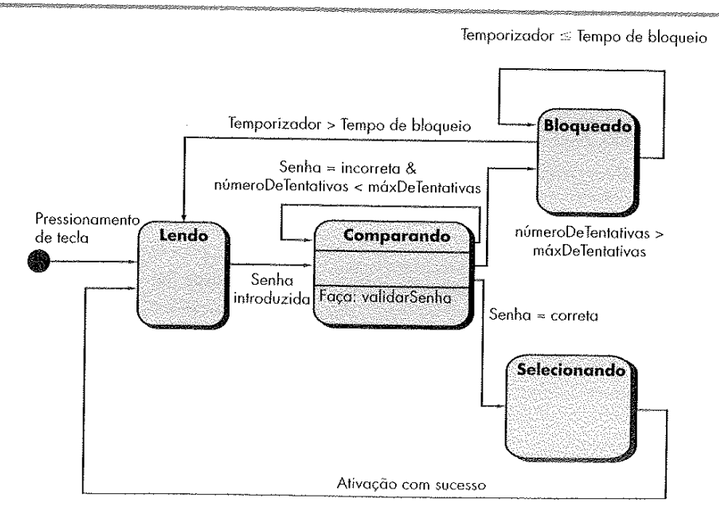
Em geral, a guarda de uma transição depende usualmente do valor de um ou mais atributos de um objeto. Em outras palavras, a guarda depende do estado passivo do objeto.

Uma *ação* ocorre concorrentemente à transição de estado ou como uma conseqüência dela e geralmente envolve uma ou mais operações (responsabilidades) do objeto. Por exemplo, a ação ligada ao evento *senha entrada* (veja a Figura 8.20) é uma operação chamada de *validarSenha()* que obtém acesso ao objeto **senha** e realiza uma comparação dígito por dígito para validar a senha.

Diagrama de seqüência. O segundo tipo de representação comportamental, chamado de *diagrama de seqüência* em UML, indica como eventos provocam transições de objeto para objeto. Uma vez identificados os eventos pelo exame de um caso de uso, o modelador cria um diagrama de seqüência — uma representação de como os eventos causam fluxo de um objeto para outro como função do tempo. Em resumo, o diagrama de seqüência é uma versão abreviada do caso de uso. Ele representa classes-chave e os eventos que fazem o comportamento fluir de classe para classe.

FIGURA 8.20

Diagrama de estado para a classe PainelDeControle

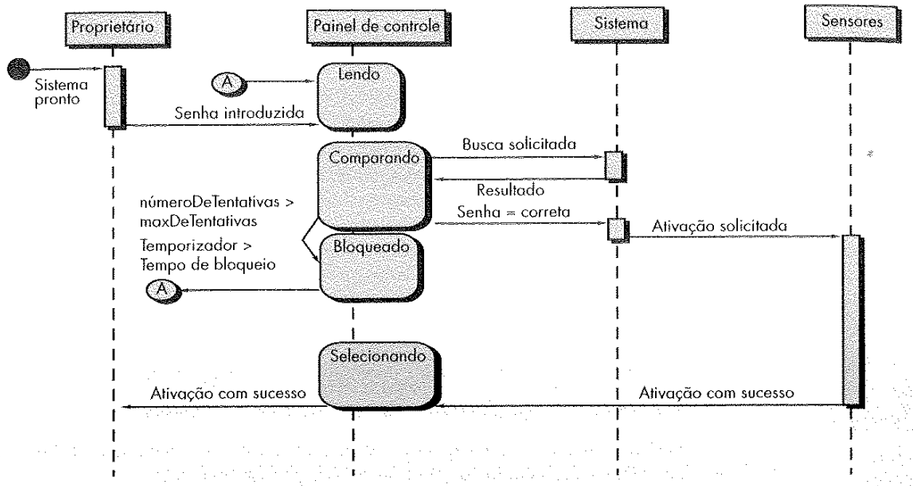


25 Neste exemplo, consideramos que cada usuário (proprietário) que interage com *CasaSegura* tem uma senha de identificação e, por essa razão, é um objeto legítimo.

26 Os diagramas de estado apresentados na Seção 8.6.3 referem-se ao estado do sistema. Nossa discussão nesta seção focaliza o estado de cada classe do modelo de análise.

Document shared on www.docsity.com

FIGURA 8.21 Diagrama de seqüência (parcial) da função de segurança *CasaSegura*



A Figura 8.21 mostra um diagrama de seqüência parcial para a função de segurança do *CasaSegura*. Cada uma das setas representa um evento (derivado de um caso de uso) e indica como o evento canaliza o comportamento entre objetos *CasaSegura*. O tempo é medido verticalmente (de cima para baixo), e os retângulos estreitos verticais representam o tempo gasto no processamento de uma atividade. Estados podem ser mostrados ao longo da linha de tempo vertical.

O primeiro evento, *sistema pronto*, é derivado do ambiente externo e canaliza comportamento para o objeto **Proprietário**. O proprietário entra com uma senha. Um evento de *solicitação de busca* é passado ao **Sistema** que procura a senha em um banco de dados simples e retorna um *resultado* (*encontrada* ou *não encontrada*) ao **PainelDeControle** (agora no estado *comparando*). Uma senha válida resulta em um evento *senha=correta* para o **Sistema** que ativa os sensores com um evento *solicitação de ativação*. Por último, o controle é retornado ao proprietário com o evento *ativação com sucesso*.

FERRAMENTAS DE SOFTWARE

Modelagem de Análise Generalizada em UML

Objetivo: Ferramentas de modelagem de análise fornecem a capacidade de desenvolver modelos baseados em cenário, classe e comportamentais usando a notação UML.

Mecânica: Ferramentas dessa categoria suportam todo o conjunto de diagramas UML necessário para construir um modelo de análise (essas ferramentas também suportam a modelagem de projeto). Além disso, para diagramação, ferramentas dessa categoria (1) executam verificação de consistência e de correção para todos os diagramas UML; (2) fornecem ligações para projeto e geração de código; (3) constroem um banco de dados que permite o gerenciamento e a avaliação de grandes modelos UML, necessários para sistemas complexos.

Ferramentas Representativas[27]

As seguintes ferramentas suportam todo o conjunto de diagramas UML necessário para modelagem de análise:

ArgoUML, uma ferramenta de código aberto (argouml.tigris.org).

Control Center, desenvolvida pela TogetherSoft (www.togethersoft.com).

Enterprise Architect, desenvolvida pela Sparx Systems (www.sparxsystems.com.au).

Object Technology Workbench (OTW), desenvolvida pela OTW Software (www.otwsoftware.com).

Power Designer, desenvolvida pela Sybase (www.sybase.com).

Rational Rose, desenvolvida pela Rational Corporation (www.rational.com).

System Architect, desenvolvida pela Popkin Software (www.popkin.com).

UML Studio, desenvolvida por Pragsoft Coporation (www.pragsoft.com).

Visio, desenvolvida pela Microsoft (www.microsoft.com).

Visual UML, desenvolvida por Visual Object Modelers (www.visualuml.com).

27 As ferramentas mencionadas aqui não representam uma recomendação, mas, em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

Quando um diagrama de seqüência completo tiver sido desenvolvido, todos os eventos que causam transições entre objetos do sistema podem ser organizados em um conjunto de eventos de entrada e de saída (de um objeto). Essa informação é útil na criação de um projeto efetivo para o sistema a ser construído.

8.9 RESUMO

O objetivo da modelagem de análise é criar uma variedade de representações que mostram os requisitos de software quanto à informação, função e comportamento. Para tanto, duas diferentes (mas, potencialmente, complementares) filosofias de modelagem podem ser aplicadas: análise estruturada e análise orientada a objetos. Análise estruturada considera o software um transformador de informação. Ela ajuda o engenheiro de software na identificação dos objetos de dados, seus relacionamentos e o modo pelo qual esses objetos de dados são transformados à medida que fluem através de funções de processamento de software. Análise orientada a objetos examina um domínio de problema definido como um conjunto de casos de uso em um esforço para extrair classes que definam o problema. Cada classe tem um conjunto de atributos e operações. Classes estão relacionadas entre si por uma variedade de modos e são modeladas por meio de diagramas UML. O modelo de análise é composto de quatro elementos de modelagem: modelos baseados em cenário, modelos de fluxo, modelos baseados em classe e modelos comportamentais.

Modelos baseados em cenário representam os requisitos de software sob o ponto de vista do usuário. O caso de uso — uma descrição narrativa ou orientada por gabarito de uma interação entre um ator e o software — é o principal elemento da modelagem. Derivado durante a extração de requisitos, o caso de uso define os passos-chave para uma função ou interação específica. O grau de formalidade e detalhes do caso de uso variam, mas o resultado final fornece a entrada necessária para todas as outras atividades de modelagem de análise. Cenários também podem ser descritos usando um diagrama de atividade — uma representação gráfica parecida com fluxograma que mostra o fluxo de processamento dentro de um cenário específico. Diagramas de raia ilustram como o fluxo de processamento é alocado a vários atores ou classes.

Modelos de fluxo focalizam o fluxo de objetos de dados à medida que são transformados por funções de processamento. Derivados da análise estruturada, modelos de fluxo usam o diagrama de fluxo de dados, uma notação de modelagem que mostra como a entrada é transformada em saída à medida que os objetos de dados se movem através do sistema. Cada função de software que transforma dados é descrita por uma especificação ou narrativa de processo. Além do fluxo de dados, esse elemento de modelagem também mostra o fluxo de controle — uma representação que ilustra como os eventos afetam o comportamento de um sistema.

Modelagem baseada em classe usa informação derivada de elementos de modelagem baseada em cenário e orientada a fluxo para identificar as classes de análise. Pode ser usada uma análise gramatical para extrair candidatos de classes, atributos e operações de narrativas baseadas em texto. Critérios para definição de uma classe são definidos. Os cartões indexados classe-responsabilidade-colaborador podem ser usados para definir relacionamentos entre classes. Além disso, uma

Document shared on www.docsity.com

variedade de notações de modelagem UML pode ser aplicada para definir hierarquias, relacionamentos, associações, agregações e dependências entre classes. Pacotes de análise são usados para categorizar e agrupar classes de um modo que as torna mais gerenciáveis para grandes sistemas.

Os três primeiros elementos de modelagem de análise fornecem uma visão estática do software. A modelagem comportamental mostra o comportamento dinâmico. O modelo comportamental usa entrada de elementos baseados em cenário orientados a fluxo e baseados em classe para representar os estados das classes de análise e do sistema como um todo. Para tanto, os estados são identificados, os eventos que fazem com que uma classe (ou o sistema) transite de um estado para outro são definidos e as ações que ocorrem quando a transição é realizada são também identificados. Diagramas de estado e diagramas de seqüência são a notação UML usada para a modelagem comportamental.

Referências Bibliográficas

[ABB83] Abbott, R., "Program Design by Informal English Descriptions", *CACM*, v. 26, n. 11, nov. 1983, p. 892-94.

[AMB95] Ambler, S., "Using Use-Cases", *Software Development*, jul. 1995, p. 53-61.

[ARA89] Arango, G., e Prieto-Diaz, R., "Domain Analysis: Concepts and Research Directions", *Domain Analysis: Acquisition of Reusable Information for Software Construction*, (Arango, G. e Prieto-Diaz, R., eds.), IEEE Computer Society Press, 1989.

[ARL02] Arlow, J., e Neustadt, I., *UML and the UnificdProcess*, Addison-Wesley, 2002.

[BER93] Berard, E. V., *Essays on Object-Oriented Software Engineering*, Addison-Wesley, 1993.

[BOO86] Booch, G., "Object-Oriented Development", *IEEE Trans. Software Engineering*, v. SE-12, n. 2, fev. 1986, p. 211f.

[BUD96] Budd, T., *An Introduction to Object-Oriented Programmming*, 2ª ed., Addison-Wesley, 1996.

[CAS89] Cashman, M., "Object-Oriented Domain Analysis", *ACM Software Engineering Notes*, v. 14, n. 6, out. 1989, p. 67.

[CHA93] de Champeaux, D., Lea, D., e Faure, P., *Object-Oriented System Development*, Addison-Wesley, 1993.

[CHE77] Chen, P., *The Entity-Relationship Approach to Logical Database Design*, QED Information Systems, 1977.

[COA91] Coad, R., e Yourdon, E., *Object-Oriented Analysis*, 2ª ed., Prentice-Hall, 1991.

[COCOI] Cockburn, A., *Writing Effective Use Cases*, Addison-Wesley, 2001.

[DAV93] Davis, A., *Software Requirements: Objects, Functions and States*, Prentice-Hall, 1993.

[DEM79] DeMarco, T., *Structured Analysis and System Specification*, Prentice-Hall, 1979.

[FIR93] Firesmith, D. G., *Object-Oriented Requirements Analysis and Logical Design*, Wiley, 1993.

[LET03] Lethbridge, T., personal communication on domain analysis, maio 2003.

[OMG03] Object Management Group, OMG Unified Modeling Language Specification, versão 1.5, mar. 2003, disponível em http://www.rational.com/uml/ resources/documentation/.

[SCH02] Schmuller, J., *Teach YourselfUML in 24 Hours*, 2ª ed., SAMS Publishing, 2002.

[SCH98] Schneider, G., e Winters, J., *Applying Use Cases*, Addison-Wesley, 1998.

[STR88] Stroustrup, B., "What Is Object-Oriented Programming?" IEEE Software, v. 5, n. 3, maio 1988, p. 10-20.

[TAY90] Taylor, D. A., *Object-Oriented Technology: A Manager's Guide*, Addison-Wesley, 1990.

[THA00] Thalheim, B., *Entity Relationship Modeling*, Springer-Verlag, 2000.

[TIL93] Tillmann, G., *A Practical Guide to Logical Data Modeling*, McGraw-Hill, 1993.

[UML03] The UML Café, "Customers Don't Print Themselves", disponível em http://www.theumlcafe.com/a0079.htm, maio 2003.

[WIR90] Wirfs-Brock, R., Wilkerson, B., e Weiner, L., *Designing Object-Oriented Software*, Prentice-Hall, 1990.

Problemas e Pontos a Considerar

8.1. É possível começar a codificar imediatamente depois que um modelo de análise tiver sido criado? Explique sua resposta e, então, argumente ao contrário.

8.2. Uma regra prática de análise é que o modelo "deve focalizar os requisitos visíveis no domínio do problema ou do negócio". Que tipos de requisitos *não* são visíveis nesses domínios? Forneça alguns exemplos.

8.3. Qual a finalidade do domínio de análise? Como está relacionado ao conceito de padrões de requisitos?

8.4. Em algumas linhas, tente descrever as principais diferenças entre análise estruturada e análise orientada a objetos.

8.5. É possível desenvolver um modelo de análise efetivo sem desenvolver todos os quatro elementos mostrados na Figura 8.3? Explique.

8.6. Você foi solicitado a construir um dos seguintes sistemas:

a. Um sistema de matrícula em curso baseado em rede para sua universidade.
b. Um sistema de processamento de pedidos baseado na Web para uma loja de computadores.
c. Um sistema simples de faturamento para um pequeno negócio.
d. Um software que substitua um Rolodex e que seja embutido em um telefone sem fio.
e. Um livro de receitas automatizado que seja embutido em um forno elétrico ou de microondas.

Selecione o sistema de seu interesse e descreva objetos de dados, relacionamentos e atributos.

8.7. Desenhe um modelo no nível de contexto (DFD de nível 0) para um dos cinco sistemas listados no Problema 8.6. Redija uma narrativa de processamento no nível de contexto para o sistema.

8.8. Usando o DFD no nível de contexto desenvolvido no Problema 8.7, desenvolva diagramas de fluxo de dados de nível 1 e 2. Use uma "análise gramatical" da narrativa de processamento no nível de contexto para dar início ao seu trabalho. Lembre-se de especificar todo o fluxo de informação rotulando todas as setas entre bolhas. Use nomes significativos para cada transformação.

8.9. Desenvolva CSPECs e PSPECs para o sistema que você selecionou no Problema 8.6. Tente fazer seu modelo tão completo quanto possível.

8.10. O departamento de obras públicas de uma cidade grande decidiu desenvolver um *sistema de acompanhamento e reparo de buracos* (SARB) baseado na Web. Veja a descrição a seguir:

> Os cidadãos podem obter acesso a um site e relatar a localização e gravidade dos buracos. À medida que os buracos são relatados, eles são registrados em um "sistema de reparo do departamento de obras públicas" e lhes é atribuído um número de identificação, armazenado por rua do endereço, tamanho (em uma escala de 1 a 10), localização (no meio da rua, na calçada etc.), bairro (determinado pela rua do endereço) e prioridade de reparo (determinada pelo tamanho do buraco). Dados da ordem de serviço são associados a cada buraco e incluem a localização e tamanho do buraco, número de identificação da equipe de reparo, número de pessoas na equipe, equipamento atribuído, horas aplicadas no reparo, estado do buraco (trabalho em andamento, reparado, reparo temporário, não reparado), quantidade de material de enchimento usado e custo do reparo (calculado pelas horas aplicadas, quantidade de pessoas, material e equipamentos usados). Finalmente, um arquivo de danos é criado para conter informação sobre danos relatados devido ao buraco e incluem nome do cidadão, endereço, número do telefone, tipo de dano, quantia em reais de prejuízo causado pelo dano. O SARP é um sistema Web; todas as consultas devem ser feitas interativamente.

Usando a notação de análise estruturada, desenvolva um modelo para o SARB.

8.11. Descreva os termos orientados a objetos — *encapsulamento* e *herança*.

8.12. Escreva um caso de uso baseado em modelo para o sistema de gestão domiciliar *CasaSegura* descrito informalmente no quadro que se segue à Seção 8.7.4.

8.13. Desenhe um diagrama de caso de uso UML para o sistema SARB introduzido no Problema 8.10. Você terá de fazer algumas suposições sobre o modo pelo qual um usuário interage com o sistema.

8.14. Desenvolva um modelo de classe para o sistema SARB introduzido no Problema 8.10.

8.15. Desenvolva um conjunto completo de cartões indexados do modelo CRC para o sistema SARB introduzido no Problema 8.10.

8.16. Conduza uma revisão dos cartões CRC com seus colegas. Quantas classes, responsabilidades e colaboradores adicionais foram criados como conseqüência da revisão?

8.17. Descreva a diferença entre uma associação e uma dependência para uma classe de análise.

8.18. O que é um pacote de análise e como ele poderia ser usado?

8.19. Como um diagrama de estado para classes de análise difere dos diagramas de estado apresentados para o sistema inteiro?

Document shared on www.docsity.com

Leituras e Fontes de Informação Adicionais

Dúzias de livros têm sido publicados sobre análise estruturada. A maioria cobre o assunto adequadamente, mas apenas alguns fazem um trabalho realmente excelente. O livro de DeMarco and Plauger (*Structured Analysis and System Specification*, Pearson, 1985) é um clássico que continua sendo uma boa introdução para a notação básica. Livros por Kendall e Kendall (*Systems Analysis and Design*, quinta edição, Prentice-Hall, 2002) e Hoffer *et al.* (*Modern Systems Analysis and Design*, Addison-Wesley, terceira edição, 2001) são referências que valem a pena. O livro de Yourdon (*Modern Structured Analysis*, Yourdon-Press, 1989) sobre o assunto continua entre os trabalhos mais abrangentes publicadas até hoje.

Allen (*Data Modeling for Everyone*, Wrox Press, 2002), Simpson e Witt (*Data Modeling Essentials*, segunda edição, Coriolis Group, 2000) Reingruber e Gregory (*Data Modeling Handbook*, Wiley, 1995) apresentam tutoriais detalhados para a criação de modelos de dados com qualidade industrial. Um interessante livro por Hay (*Data Modeling Patterns*, Dorset House, 1995) apresenta padrões de modelos de dados típicos que são encontrados em muitos negócios diferentes. Um tratamento detalhado de modelagem comportamental pode ser encontrado em Kowal (*Behavior Models: Specifying User's Expectations*, Prentice-Hall, 1992).

Casos de uso formam o fundamento da análise orientada a objetos. Livros por Bittner e Spence (*Use Case Modeling*, Addison-Wesley, 2002), Cockburn [COC01], Armour e Miller (*Advanced Use-Case Modeling: Software Systems*, Addison-Wesley, 2000), e Rosenberg e Scott (*Use-Case Driven Object Modeling with UML: A Practical Approach*, Addison-Wesley, 1999) fornecem diretrizes que valem a pena para criação e uso desse importante mecanismo para extração e representação de requisitos.

Valiosas discussões sobre a UML têm sido escritas por Arlow e Neustadt [ARL02], Schmuller [SCH02], Fowler e Scott (*UML Distilled*, segunda edição, Addison-Wesley, 1999), Booch e seus colegas (*The UML User Guide*, Addison-Wesley, 1998), e Rumbaugh e seus colegas (*The Unified Modeling Language Reference Manual*, Addison-Wesley, 1998).

Os métodos subjacentes de análise e projeto que suportam o Processo Unificado são discutidos por Larman (*Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, segunda edição, Prentice-Hall, 2001), Dennis e seus colegas (*System Analysis and Design: An Object-Oriented Approach with UML*, Wiley, 2001), e Rosenberg e Scott (*Use-Case Driven Object Modeling with UML*, Addison-Wesley, 1999). Balcer e Mellor (*Executable UML: A Foundation for Model Driven Architecture*, Addison-Wesley, 2002) discutem a semântica global da UML, os modelos que podem ser criados e um modo de considerar a UML como uma linguagem executável. Starr (*Executable UML: How to Build Class Models*, Prentice-Hall, 2001) fornece diretrizes úteis e sugestões detalhadas para criar classes efetivas de análise e projeto.

Uma grande variedade de fontes de informação sobre modelagem de análise está disponível na Internet. Uma lista atualizada de referências relevantes para a modelagem de análise pode ser encontrada no site deste livro: **http://www.mhhe.com/pressman.**

Engenharia de Projeto

CAPÍTULO 9

Conceitos-Chave

abstração 190
arquitetura 191
projeto
classes 195
elementos do modelo .. 197
qualidade 188
independência funcional .. 193
modularidade 192
ocultamento da informação 193
padrões 191
refabricação 194
refinamento 194

A engenharia de projeto engloba um conjunto de princípios, conceitos e práticas que levam ao desenvolvimento de um sistema ou produto de alta qualidade. Princípios de projeto (discutidos no Capítulo 5) estabelecem uma filosofia prevalecente que guia o projetista no trabalho realizado. Conceitos de projeto devem ser compreendidos antes que a mecânica da prática de projeto seja aplicada, e práticas de projeto em si levam à criação de várias representações do software que norteiam a atividade de construção que se segue.

Engenharia de projeto não é uma frase comumente usada no contexto da engenharia de software, no entanto, deveria ser. Projeto é uma atividade central de engenharia. No início da década de 1990, Mitch Kapor, o criador do Lotus 1-2-3, apresentou um "manifesto de projeto de software" no *Dr. Dobbs Journal*. Ele declarou:

> O que é projeto? É onde você se instala com um pé em dois mundos — o mundo da tecnologia e o mundo das pessoas e objetivos humanos — e você tenta juntar os dois...
>
> O crítico romano de arquitetura Vitruvius adiantou a noção de que edifícios bem projetados eram aqueles que exibiam firmeza, comodidade e prazer. O mesmo poderia ser dito de bom software. *Firmeza:* um programa não deve ter quaisquer erros que inibam sua função. *Comodidade*: um programa deve ser adequado às finalidades para as quais foi planejado. *Prazer*: a experiência de usar o programa deve ser agradável. Temos aí o início de uma teoria de projeto de software.

O objetivo da engenharia de projeto é produzir um modelo ou representação que exiba firmeza, comodidade e prazer. Para tanto, um projetista precisa praticar a diversificação e depois a convergência. Belady [BEL81] afirma que "diversificação é a aquisição de um repertório de alternativas, a matéria-prima do projeto: componentes, soluções de componentes e conhecimento, tudo contido em catálogos, livros-textos e na mente."Uma vez montado esse conjunto diversificado de informação, o projetista deve pegar e escolher elementos do repertório que satisfaça aos requisitos definidos pela engenharia de requisitos (Capítulo 7) e pelo modelo de análise (Capítulo 8). À medida que isso ocorre, alternativas são consideradas e rejeitadas, o engenheiro de projeto converge para "uma particular configuração de componentes e, assim, a criação do produto final" [BEL81].

Panorama

O que é? O projeto é o que virtualmente todo engenheiro quer fazer. É o lugar em que a criatividade impera — em que os requisitos do cliente, as necessidades do negócio e as considerações técnicas se juntam na formulação de um produto ou sistema. O projeto cria uma representação ou modelo do software, mas diferente do modelo de análise (que enfoca a descrição dos dados, função e comportamento requeridos), o modelo de projeto fornece detalhe sobre as estruturas de dados, arquitetura, interfaces e componentes do software necessários para implementar o sistema.

Quem faz? Engenheiros de software conduzem cada uma das tarefas de projeto.

Por que é importante? O projeto permite ao engenheiro de software modelar o sistema ou produto que deve ser construído. Esse modelo pode ser avaliado quanto à qualidade e aperfeiçoado antes que o código seja gerado, testes sejam conduzidos e usuários finais fiquem envolvidos em grande número. Projeto é o lugar em que a qualidade do software é estabelecida.

Quais são os passos? O projeto mostra o software de vários modos diferentes. Primeiro, a arquitetura do sistema ou

Document shared on www.docsity.com

produto precisa ser representada. Depois, as interfaces que conectam o software aos usuários finais, a outros sistemas e dispositivos e aos seus próprios componentes constitutivos são modeladas. Por fim, os componentes de software usados para construir o sistema são projetados. Cada uma dessas visões representa uma diferente ação de projeto, mas todas precisam estar de acordo com um conjunto de conceitos básicos de projeto que norteiam todo o trabalho de projeto de software.

Qual é o produto do trabalho? Um modelo de projeto que inclui representações de arquitetura, interface, componente e implantação é o principal produto do trabalho produzido durante o projeto de software.

Como tenho certeza de que fiz corretamente? O modelo de projeto é avaliado pela equipe de software em um esforço para determinar se contém erros, inconsistências ou omissões; se existem alternativas melhores e se o modelo pode ser implementado dentro das restrições, cronograma e custo que foram estabelecidos.

Diversificação e convergência demandam intuição e julgamento. Essas qualidades são baseadas na experiência de construção de entidades semelhantes, em um conjunto de princípios e/ou heurísticas que guia o modo pelo qual o modelo evolui, em um conjunto de critérios que permite que a qualidade seja julgada, e em um processo de iteração que, em última instância, leva à representação do projeto final.

Engenharia de projeto de software de computador muda continuamente à medida que novos métodos, melhor análise e mais amplo entendimento evoluem. Mesmo atualmente, falta à maioria das metodologias de projeto de software a profundidade, a flexibilidade e a natureza quantitativa, normalmente associadas às disciplinas mais clássicas de engenharia de projeto. No entanto, métodos para projeto de software efetivamente existem, critérios para qualidade de projeto estão disponíveis e notação de projeto pode ser aplicada. Neste capítulo, exploraremos os conceitos e princípios fundamentais aplicáveis a todo projeto de software, os elementos de modelo de projeto e o impacto dos padrões no processo de projeto. Nos Capítulos 10, 11 e 12 examinaremos uma variedade de métodos de projeto de software, na medida em que são aplicados ao projeto arquitetural, de interface e de componente.

9.1 Projeto no Contexto de Engenharia de Software

AVISO

Engenharia de projeto deve sempre começar com uma consideração sobre os dados — a base para todos os outros elementos do projeto. Depois que a base estiver assentada, a arquitetura deve ser derivada. Somente então, você deve realizar as outras tarefas de projeto.

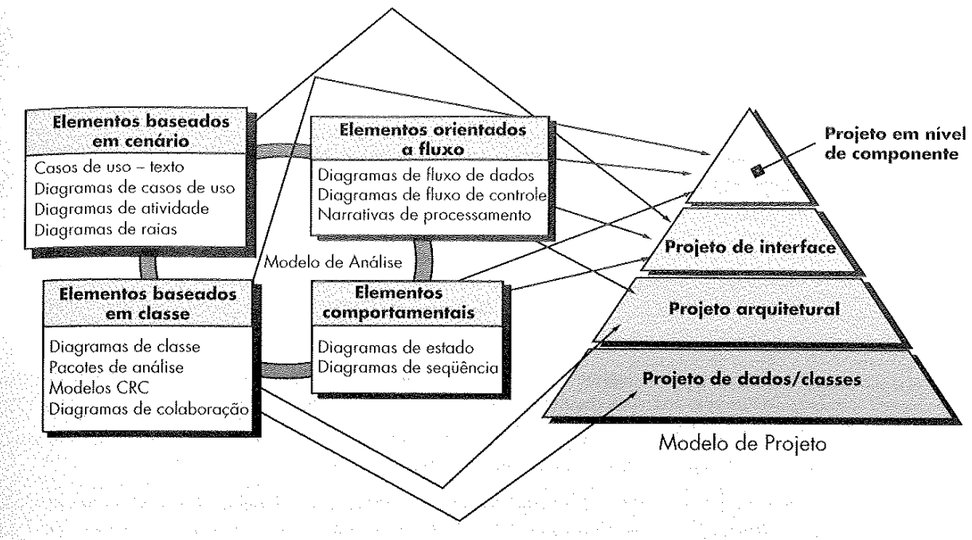
O projeto de software situa-se no núcleo técnico da engenharia de software e é aplicado independentemente do modelo de processo de software usado. Iniciando logo que os requisitos de software tenham sido analisados e modelados, o projeto de software é a última ação do engenheiro de software dentro da atividade de modelagem e prepara a cena para construção (geração de código e teste).

> "O milagre mais comum da engenharia de software é a transição da análise para o projeto e do projeto para o código."
>
> **Richard Duo**

Cada um dos elementos do modelo de análise (veja o Capítulo 8) fornece informação necessária para criar os quatro modelos necessários para uma completa especificação do projeto. O fluxo de informação durante o projeto de software é mostrado na Figura 9.1. O modelo de análise manifestado por elementos baseados em cenário e em classe, orientados a fluxo e comportamentais alimenta a tarefa de projeto. Usando a notação de projeto e os métodos discutidos nos capítulos seguintes, o projeto produz um projeto de dados/classes, um projeto arquitetural, um projeto de interface e um projeto de componentes.

O projeto de dados/classes transforma modelos de análise-classe (veja o Capítulo 8) em realizações de classes de projeto e nas estruturas de dados dos requisitos necessárias para implementar o software. As classes e os relacionamentos definidos nos cartões de índice CRC e o conteúdo detalhado dos dados mostrado pelos atributos das classes e outras notações fornecem a base para a atividade de projeto de dados. Parte do projeto de classes pode ocorrer em conjunção com o projeto da arquitetura do software. O projeto de classe mais detalhado ocorre à medida que cada componente do software é projetado.

Figura 9.1 Tradução do modelo de análise para um modelo de projeto



O projeto arquitetural define os relacionamentos entre os principais elementos estruturais do software, os estilos arquiteturais e padrões de projeto que podem ser usados para satisfazer aos requisitos definidos para o sistema e as restrições que afetam o modo pelo qual o modelo arquitetural pode ser implementado [SHA96]. A representação do projeto arquitetural — o arcabouço de um sistema baseado em computador — pode ser derivada da especificação do sistema, do modelo de análise e da interação dos subsistemas definida no modelo de análise.

O projeto de interface descreve como o software se comunica com os sistemas que interoperam com ele e com as pessoas que o utilizam. Uma interface implica um fluxo de informação (dados e/ou controle) e um tipo de comportamento específico. Assim, os cenários de uso e modelos comportamentais fornecem muito da informação necessária para o projeto da interface.

O projeto em nível de componente transforma elementos estruturais da arquitetura de software em uma descrição procedural dos componentes de software. A informação obtida do modelo baseado em classes, modelos de fluxo e modelos comportamentais serve de base para o projeto de componentes.

> "Há dois modos de construir um projeto de software: um é fazê-lo tão simples que obviamente não haja deficiências; o outro é fazê-lo tão complicado que não haja deficiências óbvias. O primeiro modo é muito mais difícil."
>
> **C. A. R. Hoare**

Durante o projeto tomamos decisões que vão, em última análise, afetar o sucesso da construção do software e, mais importante, a facilidade com a qual o software pode ser mantido. Mas por que o projeto é tão importante?

A importância do projeto de software pode ser definida com uma única palavra — *qualidade*. Projeto é a etapa na qual a qualidade é incorporada na engenharia de software. O projeto nos fornece representações do software que podem ser avaliadas quanto à qualidade. É o único modo pelo qual podemos traduzir precisamente os requisitos do cliente em um produto ou sistema de software acabado. O projeto de software serve de base para todos os passos de engenharia de software e de suporte de software que se seguem. Sem projeto, nos arriscamos a construir um sistema instável — que falhará quando pequenas modificações forem feitas; que pode ser difícil de testar; cuja qua-

Document shared on www.docsity.com

lidade não pode ser avaliada até uma fase avançada do processo de software, na qual o prazo está se esgotando e muito dinheiro já foi gasto.

9.2 Processo de Projeto e Qualidade de Projeto

Projeto de software é um processo iterativo por meio do qual os requisitos são traduzidos em um "documento" para construção do software. Inicialmente, o documento mostra uma visão holística do software. Ou seja, o projeto é representado em um nível alto de abstração — nível que pode ser diretamente relacionado ao objetivo específico do sistema e a requisitos mais detalhados de dados, funcionais e comportamentais. À medida que ocorrem as iterações de projeto, os refinamentos subseqüentes levam a representações de projeto em níveis de abstração muito mais baixos. Esses ainda podem ser relacionados aos requisitos, mas a conexão é mais sutil.

Ao longo do processo, a qualidade do projeto em evolução é avaliada com uma série de revisões técnicas formais ou *walkthroughs* de projeto discutidos no Capítulo 26. McGlaughlin [MCG91] sugere três características que servem de orientação para a avaliação de um bom projeto:

- O projeto deve implementar todos os requisitos explícitos contidos no modelo de análise e acomodar todos os requisitos implícitos desejados pelo cliente.
- O projeto deve ser um guia legível, compreensível para aqueles que geram código e para os que testam e, subseqüentemente, dão suporte ao software.
- O projeto deve fornecer um quadro completo do software, focalizando os domínios de dados, funcional e comportamental sob uma perspectiva de implementação.

Cada uma dessas características é na verdade uma meta do processo de projeto. Entretanto, como cada uma dessas metas é atingida?

> "[E]screver um pedaço inteligente de código que funcione é uma coisa, projetar algo que possa dar suporte a negócios duradouros é outra."
>
> **C. Ferguson**

Diretrizes de qualidade. Para avaliarmos a qualidade de uma representação do projeto, precisamos estabelecer critérios técnicos para um bom projeto. Mais adiante, neste capítulo, discutiremos conceitos de projeto que também servem como critérios de qualidade de software. Por enquanto, apresentamos as seguintes diretrizes:

? Quais são as características de um bom projeto?

1. Um projeto deve exibir uma arquitetura que (a) tenha sido criada por meio de estilos ou padrões arquiteturais reconhecíveis, (b) seja composta de componentes que exibam boas características de projeto (discutidas mais adiante neste capítulo), e (c) pode ser implementada de modo evolucionário,[1] facilitando assim a implementação e o teste.
   Quais são as características de um bom projeto?
2. Um projeto deve ser modular; o software deve ser logicamente particionado em elementos ou subsistemas.
3. Um projeto deve conter representações distintas dos dados, arquitetura, interfaces e componentes.
4. Um projeto deve levar a estruturas de dados adequadas às classes a ser implementadas e baseadas em padrões de dados reconhecíveis.
5. Um projeto deve levar a componentes que exibam características funcionais independentes.
6. Um projeto deve levar a interfaces que reduzam a complexidade das conexões entre componentes e com o ambiente externo.
7. Um projeto deve ser derivado por meio de um método passível de repetição que seja conduzido pela informação obtida durante a análise dos requisitos do software.

1 Para sistemas menores, o projeto pode algumas vezes ser desenvolvido linearmente.

8. Um projeto deve ser representado usando uma notação que efetivamente comunique seu significado.

Essas diretrizes de projeto não são satisfeitas por acaso. A engenharia de projeto incentiva um bom projeto por meio da aplicação de princípios fundamentais de projeto, metodologia sistemática e revisões precisas.

INFO

Avaliação da Qualidade do Projeto – A Revisão Técnica Formal

O projeto é importante porque permite à equipe de software avaliar a qualidade[2] do software antes de ele ser implementado — na hora em que erros, omissões ou inconsistências são fáceis e não dispendiosas de corrigir. Mas como fazemos avaliação de qualidade durante o projeto? O software não pode ser testado porque não há software executável para testar. O que fazer?

Durante o projeto, a qualidade é avaliada pela condução de uma série de revisões técnicas formais (RTFs; ou *Formal Technical Review* — FTR). RTFs são discutidas em detalhe[3] no Capítulo 26, mas vale a pena fornecer um resumo da técnica neste ponto. Uma RTF é um encontro conduzido por membros da equipe de software. Normalmente, duas, três ou quatro pessoas participam, dependendo do escopo da informação do projeto a ser revisada. Cada pessoa desempenha um papel: o *revisor chefe* planeja o encontro, estabelece uma agenda e então conduz o encontro; o *registrador* faz anotações de modo que nada seja perdido; o *produtor* é a pessoa cujo produto de trabalho (o projeto de um componente de software) está sendo revisado. Antes da reunião, a cada membro da equipe de revisão é dada uma cópia do produto de trabalho do projeto e é solicitado que a leia, procurando erros, omissões ou ambigüidades. Quando a reunião começa, o objetivo é anotar todos os problemas que existem em um produto de trabalho para que eles possam ser corrigidos antes que a implementação comece. A RTF tipicamente dura de 90 minutos a duas horas. Na conclusão da RTF, a equipe de revisão determina se mais ações são necessárias da parte do produtor antes que o resultado do trabalho possa ser aprovado como parte do modelo de projeto final.

> "Qualidade não é algo que você coloca sobre assuntos e objetos como um enfeite em uma árvore de Natal."
>
> **Robert Pirsig**

Atributos de qualidade. A Hewlett-Packard [GRA87] desenvolveu um conjunto de atributos de qualidade de software que recebeu a sigla FURPS — Functionality (funcionalidade), Usability (usabilidade), Reliability (confiabilidade), Performance (desempenho) e Supportability (suportabilidade). Os atributos de qualidade FURPS representam uma meta para todo o projeto de software:

AVISO

Projetistas de software tendem a se concentrar no problema a ser resolvido. Contudo não podem esquecer de que os atributos FURPS fazem parte do problema. Eles precisam ser considerados.

- *Funcionalidade* é avaliada pela observação do conjunto de características e capacidades do programa, generalidade das funções entregues e segurança do sistema global.
- *Usabilidade* é avaliada considerando fatores humanos (veja o Capítulo 12), estética, consistência e documentação globais.
- *Confiabilidade* é avaliada medindo-se a freqüência e a severidade das falhas, a precisão dos resultados de saída, o tempo médio entre falhas (*mean-time-to-failure* — MTTF), a capacidade de recuperação de falhas e a previsibilidade do programa.
- *Desempenho* é medido pela velocidade de processamento, tempo de resposta, consumo de recursos, vazão (*throughput*) e eficiência.
- *Suportabilidade* combina a capacidade de estender o programa (extensibilidade), adaptabilidade, reparabilidade — esses três atributos representam um termo mais comum, *manutenabilidade* —, além de testabilidade, compatibilidade, configurabilidade (a capacidade de organizar e

2 Os fatores de qualidade discutidos no Capítulo 15 podem apoiar a equipe de revisão à medida que ela avalia a qualidade.

3 Você pode considerar a possibilidade de examinar a Seção 26.4 agora. RTFs é uma parte crítica do processo de projeto e é um mecanismo importante para atingir a qualidade do projeto.

Document shared on www.docsity.com

controlar elementos da configuração do software; veja o Capítulo 27), facilidade com a qual o sistema pode ser instalado e facilidade com a qual problemas podem ser localizados.

Nem todo atributo de qualidade de software é igualmente ponderado à medida que o projeto de software é desenvolvido. Uma aplicação pode salientar funcionalidade com ênfase especial em segurança. Outra pode exigir desempenho com ênfase particular na velocidade de processamento. Uma terceira pode enfocar a confiabilidade. Indiferentemente da ponderação, é importante notar que esses atributos de qualidade devem ser considerados quando o projeto começa, *não* depois que o projeto estiver completo e a construção, começada.

CONJUNTO DE TAREFAS

Conjunto Genérico de Tarefas de Projeto

1. Examinar o modelo de domínio de informação e projetar estruturas de dados adequadas para os objetos de dados e seus atributos.
2. Usando o modelo de análise, selecionar um estilo arquitetural (padrão) que seja adequado para o software.
3. Particionar o modelo de análise em subsistemas de projeto e alocá-los na arquitetura. Certifique-se de que cada subsistema é funcionalmente coeso.
   Projetar as interfaces dos subsistemas.
   Alocar classes de análise ou funções para cada subsistema.
4. Criar um conjunto de classes de projeto ou componentes. Traduzir cada descrição de classe de análise em uma classe de projeto.
   Verificar cada classe de projeto segundo o critério de projeto; considerar tópicos de herança.
   Definir métodos e mensagens associadas a cada classe de projeto.
   Avaliar e selecionar padrões de projeto para uma classe de projeto ou um subsistema.
   Rever as classes de projeto e revisar quando necessário.
5. Projetar qualquer interface exigida por sistemas ou dispositivos externos.
6. Projetar a interface de usuário.
   Revisar os resultados da análise de tarefas.
   Especificar a seqüência de ações com base nos cenários de usuários.
   Criar um modelo comportamental da interface.
   Definir objetos de interface, mecanismos de controle.
   Rever o projeto de interface e revisar quando necessário.
7. Conduzir o projeto de componentes.
   Especificar todos os algoritmos em um nível relativamente baixo de abstração.
   Refinar a interface de cada componente.
   Definir as estruturas de dados de componentes.
   Rever cada componente e corrigir todos os erros descobertos.
8. Desenvolver um modelo de implantação.

9.3 CONCEITOS DE PROJETO

Um conjunto de conceitos fundamentais de projeto de software tem evoluído durante a história da engenharia de software. Apesar de o grau de interesse em cada conceito ter variado ao longo dos anos, cada um resistiu ao teste do tempo. Cada um fornece ao projetista de software uma base por meio da qual métodos mais sofisticados de projeto podem ser aplicados.

M. A. Jackson certa vez disse: "O princípio da sabedoria de um [engenheiro de software] é reconhecer a diferença entre conseguir que um programa funcione e consegui-lo corretamente" [JAC75]. Conceitos fundamentais de projeto de software fornecem o arcabouço necessário para "fazê-lo corretamente".

9.3.1 Abstração

Quando consideramos uma solução modular para qualquer problema, muitos *níveis de abstração* podem ser colocados. No nível mais alto de abstração, uma solução é enunciada em termos amplos usando a linguagem do ambiente do problema. Nos níveis mais baixos de abstração, uma descrição mais detalhada da solução é fornecida.

> "Abstração é um dos modos fundamentais pelos quais nós, como seres humanos, enfrentamos a complexidade."
>
> **Grady Booch**

AVISO

Como projetista, trabalhe duro para deduzir tanto abstrações procedurais quanto de dados que sirvam ao problema, porém também possam ser reutilizadas em outras situações.

À medida que nos deslocamos entre diferentes níveis de abstração, trabalhamos para criar abstrações procedimentais e de dados. Uma *abstração procedural* refere-se a uma seqüência de instruções que têm uma função específica e limitada. O nome de uma abstração procedural implica essas funções, mas detalhes específicos são omitidos. Um exemplo de abstração procedural seria a palavra *abrir* para uma porta. *Abrir* implica uma longa seqüência de passos procedurais (por exemplo, ande até a porta, procure e segure a maçaneta, gire a maçaneta e puxe a porta, afaste-se da porta em movimento etc.).[4]

Abstração de dados é uma coleção de dados com uma denominação que descreve um objeto de dados. No contexto da abstração procedural *abrir*, podemos definir uma abstração de dados chamada de **porta**. Como qualquer objeto de dados, a abstração de dados para **porta** abrangeria um conjunto de atributos que descrevem a porta (**tipo de porta, direção de giro, mecanismo de abertura, peso, dimensões**). Daí decorre que a abstração procedimental *abrir* faria uso da informação contida nos atributos da abstração de dados **porta**.

9.3.2 Arquitetura

Arquitetura de software refere-se à "estrutura global do software e aos modos pelos quais essa estrutura fornece integridade conceitual para um sistema"[SHA95a]. Em sua forma mais simples, arquitetura é a estrutura ou organização dos componentes de programa (módulos), o modo pelo qual esses componentes interagem e as estruturas dos dados que são usadas pelos componentes. Em um sentido mais amplo, no entanto, *componentes* podem ser generalizados para representar os principais elementos do sistema e suas interações.

> "Uma arquitetura de software é o produto do trabalho de desenvolvimento que dá o retorno de investimento mais alto no que diz respeito a qualidade, cronograma e custo."
>
> **Len Bass *et al.***

Veja na Web

Uma discussão profunda de arquitetura de software pode ser encontrada em www.sei.cmu.edu/ata/ata_init.html.

Uma meta do projeto de software é derivar um quadro arquitetural do sistema. Esse quadro serve como arcabouço com base no qual atividades de projeto detalhado são conduzidas. Um conjunto de padrões de software arquiteturais permite ao engenheiro de software reusar conceitos de projeto.

O projeto arquitetural pode ser representado usando um ou vários modelos diferentes [GAR95]. *Modelos estruturais* representam a arquitetura como uma coleção organizada de componentes de programa. *Modelos de arcabouço* aumentam o nível de abstração de projeto tentando identificar arcabouços de projeto arquitetural repetitíveis (padrões), que são encontrados em tipos de aplicação similares. *Modelos dinâmicos* tratam dos aspectos comportamentais da arquitetura de programa, indicando como a estrutura ou a configuração do sistema pode mudar em função de eventos externos. *Modelos de processo* focalizam o projeto dos processos de negócio ou técnicos que o sistema deve acomodar. Por fim, *modelos funcionais* podem ser usados para representar a hierarquia funcional de um sistema. O projeto arquitetural será discutido no Capítulo 10.

AVISO

Não deixe que a arquitetura apenas aconteça. Se o fizer, você vai gastar o resto do projeto tentando encaixar nela o projeto à força. Projete a arquitetura explicitamente.

9.3.3 Padrões

Brad Appleton define um *padrão de projeto* da seguinte maneira: "O padrão é uma porção identificada de conhecimento profundo, que transmite a essência de uma solução provada, para um problema recorrente em certo contexto, em meio a preocupações concorrentes" [APP98]. Dito de outro modo, um padrão de projeto descreve uma estrutura de projeto que resolve um problema particular de projeto em um contexto específico e em meio a "forças" que podem ter impacto na maneira pela qual o padrão é aplicado e usado.

4 Deve-se notar, no entanto, que um conjunto de operações pode ser substituído por outro, desde que a função implicada pela abstração procedural permaneça a mesma. Assim, os passos necessários para implementar *abrir* modificar-se-iam dramaticamente se a porta fosse automática e acoplada a um sensor.

Document shared on www.docsity.com

> "Cada padrão descreve um problema que ocorre repetidas vezes em nosso ambiente, e então descreve o núcleo da solução daquele problema, de tal modo que você pode usar essa solução um milhão de vezes, sem nunca fazê-lo do mesmo modo duas vezes."
>
> **Christopher Alexander**

A intenção de cada padrão de projeto é fornecer uma descrição que habilite o projetista a determinar (1) se o padrão é aplicável ao trabalho corrente, (2) se o padrão pode ser reutilizado (conseqüentemente, ganhando tempo de projeto), e (3) se o padrão pode servir como guia para desenvolvimento de um padrão similar, mas funcionalmente ou estruturalmente diferente dele. Padrões de projeto serão discutidos em mais detalhes na Seção 9.5.

9.3.4 Modularidade

Arquitetura de software e padrões de projeto incorporam *modularidade*; o software é dividido em componentes nomeados separadamente e endereçáveis, algumas vezes chamados de *módulos*, que são integrados para satisfazer aos requisitos do problema.

Tem sido dito que a "modularidade é o atributo individual do software que permite a um programa ser gerenciável intelectualmente" [MYE78]. Software monolítico (um programa grande composto de um único módulo) não pode ser facilmente compreendido por um engenheiro de software. O número de caminhos de controle, intervalos de referência, número de variáveis e a complexidade global fariam que a compreensão fosse próxima do impossível. Para ilustrar esse ponto, considere o seguinte argumento baseado em observações da solução de problemas por seres humanos.

Considere dois problemas, p_1 e p_2. Se a complexidade perceptível de p_1 é maior do que a complexidade perceptível de p_2, tem-se que o esforço requerido para resolver p_1 é maior do que o requerido para resolver p_2. No caso geral, esse resultado é intuitivamente óbvio. Realmente, leva-se mais tempo para resolver um problema difícil.

Também segue-se que a complexidade perceptível de dois problemas, quando combinados, é em geral maior que a soma da complexidade perceptível quando cada problema é considerado separadamente. Isso leva à estratégia "dividir e conquistar" — é mais fácil resolver um problema complexo quando você o divide em partes gerenciáveis. Isso tem implicações importantes com respeito à modularidade e ao software. É na realidade um argumento a favor da modularidade.

AVISO

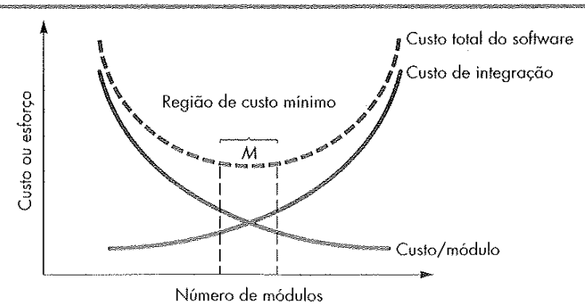
Não modularize demais. A simplicidade de cada módulo será comprometida pela complexidade da integração.

É possível concluir que, se subdividirmos o software indefinidamente, o esforço necessário para desenvolvê-lo tornar-se-á desprezivelmente pequeno! Infelizmente, outras forças entram em jogo, fazendo com que essa conclusão seja (tristemente) inválida. Observando a Figura 9.2, o esforço (custo) para desenvolver um módulo de software individual de fato diminui à medida que o número total de módulos aumenta. Dado o mesmo conjunto de requisitos, mais módulos significa menor tamanho individual. No entanto, à medida que o número de módulos cresce, o esforço (custo) associado à integração dos módulos também cresce. Essa característica leva à curva de custo, ou esforço total, mostrada na figura. Há um número M de módulos que resultaria no custo de desenvolvimento, mas não temos a sofisticação necessária para prever M com segurança.

As curvas mostradas na Figura 9.2 de fato fornecem diretrizes úteis quando a modularidade é considerada. Devemos modularizar, mas tomar o cuidado de ficar na proximidade de M. Submodularização ou supermodularização devem ser evitadas. Mas como podemos saber "a proximidade de M"? Quão modular devemos fazer o software? As respostas a essas questões requerem entendimento de outros conceitos de projeto considerados posteriormente neste capítulo.

FIGURA 9.2

Modularidade e custo de software



Modularizamos um projeto (e o programa resultante) de modo que o desenvolvimento possa ser mais facilmente planejado; incrementos de software possam ser definidos e liberados; modificações possam ser mais facilmente acomodadas; teste e depuração possam ser conduzidos mais eficientemente; e manutenção a longo prazo possa ser conduzida sem sérios efeitos colaterais.

9.3.5 Ocultamento da Informação

PONTO CHAVE

A intenção do ocultamento de informação é esconder os detalhes das estruturas de dados e processamento procedural por trás da interface do módulo. Conhecimento de detalhes não precisa ser algo que os usuários do módulo devam saber.

O conceito de modularidade leva todo projetista de software a uma questão fundamental: "Como decompomos uma solução de software para obter o melhor conjunto de módulos?". O princípio de *ocultamento da informação* [PAR72] sugere que os módulos sejam "caracterizados por decisões de projeto que (cada um) esconde de todos os outros". Em outras palavras, módulos devem ser especificados e projetados para que a informação (algoritmos e dados) contida em um módulo seja inacessível a outros módulos que não necessitam dessa informação.

Ocultamento implica que efetiva modularidade pode ser conseguida pela definição de um conjunto de módulos independentes que informam uns aos outros apenas o necessário para realizar uma função do software. Abstração ajuda a definir as entidades procedurais (ou informacionais) que constituem o software. Ocultamento define e impõe restrições de acesso, tanto a detalhes de processamento dentro de um módulo quanto a qualquer estrutura de dados local usada pelo módulo [ROS75].

O uso de ocultamento da informação como critério de projeto para sistemas modulares fornece os maiores benefícios quando são necessárias modificações durante o teste e, posteriormente, durante a manutenção do software. Como a maior parte dos dados e procedimentos estão ocultas de outras partes do software, erros inadvertidamente introduzidos durante a modificação são menos prováveis de serem propagados para outros locais do software.

9.3.6 Independência Funcional

O conceito de *independência funcional* é uma decorrência direta da modularidade e dos conceitos de abstração e ocultamento da informação. Em trabalhos marcantes sobre projeto de software, Wirth [WIR71] e Parnas [PAR72] tratam de técnicas de refinamento e destacam a independência modular. Stevens, Myers e Constantine [STE74] solidificaram o conceito em trabalho posterior.

PONTO CHAVE

Coesão é uma indicação qualitativa do grau em que um módulo focaliza apenas uma coisa.

A independência funcional é obtida pelo desenvolvimento de módulos com função de "finalidade única" e uma "aversão" à interação excessiva com outros módulos. Dito de outro modo, queremos projetar software para que cada módulo cuide de uma subfunção específica dos requisitos e tenha uma interface simples quando visto de outras partes da estrutura do programa. É justo perguntar por que a independência é importante.

Software com efetiva modularidade, isto é, módulos independentes, é mais fácil de desenvolver, porque a função pode ser compartimentalizada e as interfaces são simplificadas (considere as ramificações quando o desenvolvimento é conduzido por uma equipe). Módulos independentes são mais fáceis de manter (e testar), porque os efeitos secundários causados por modificação de projeto ou código são limitados, a propagação de erros é reduzida e os módulos reusáveis são possíveis. Para resumir, independência funcional é a chave para um bom projeto, e o projeto é a chave da qualidade de software.

PONTO CHAVE

Acoplamento é uma indicação qualitativa do grau com que um módulo está conectado a outros módulos e ao mundo exterior.

Independência é avaliada usando dois critérios qualitativos: coesão e acoplamento. *Coesão* indica a robustez funcional relativa de um módulo. *Acoplamento* indica a interdependência relativa entre módulos.

Coesão é uma extensão natural do conceito de ocultamento da informação descrito na Seção 9.3.5. Um módulo coeso realiza uma única tarefa, requerendo pouca interação com outros componentes em outras partes do programa. Dito simplesmente, um módulo coeso deveria (idealmente) fazer apenas uma coisa.

Acoplamento é uma indicação da interconexão entre módulos em uma estrutura de software. O acoplamento depende da complexidade da interface entre módulos, do ponto em que é feita a

Document shared on www.docsity.com

entrada ou referência a um módulo e de quais dados passam pela interface. Em projeto de software, lutamos por acoplamento mais baixo possível. Conectividade simples entre módulos resulta em software bem mais fácil de entender e menos propenso a "efeito de propagação" [STE74] (também chamado de "efeito gatilho"), que acontece quando erros que ocorrem em um lugar se propagam por todo o sistema.

9.3.7 Refinamento

Refinamento passo a passo é uma estratégia de projeto descendente, originalmente proposta por Niklaus Wirth [WIR71]. Um programa é desenvolvido pelo refinamento sucessivo de níveis de detalhes procedurais. Uma hierarquia é desenvolvida pela decomposição de um enunciado macroscópico de função (uma abstração procedural) passo a passo até alcançar declarações em linguagem de programação.

AVISO

Há uma tendência de ir imediatamente até o último detalhe, pulando os passos de refinamento. Isso leva a erros e omissões e torna o projeto muito mais difícil de revisar. Faça passo a passo o refinamento.

Refinamento é na verdade um processo de *elaboração*. Começamos com um enunciado da função (ou descrição da informação) que é definida em um alto nível de abstração. O enunciado descreve a função ou informação conceitualmente, mas não fornece qualquer informação sobre o funcionamento interno da função ou a estrutura interna da informação. O refinamento leva o projetista a elaborar o enunciado original, fornecendo mais e mais detalhes à medida que cada refinamento (elaboração) sucessivo ocorre.

Abstração e refinamento são conceitos complementares. A abstração permite ao projetista especificar procedimentos e dados e ainda suprimir detalhes de baixo nível. O refinamento ajuda o projetista a revelar os detalhes de baixo nível à proporção que o projeto progride. Ambos os conceitos ajudam o projetista a criar um modelo completo de projeto à medida que o projeto evolui.

> "Eu não falhei. Apenas encontrei 10 mil modos que não funcionam."
>
> **Thomas Edison**

9.3.8 Refabricação

Veja na Web

Excelentes recursos para refabricação podem ser encontrados em www.refactoring.com.

Uma importante atividade de projeto sugerida por muitos métodos ágeis (veja o Capítulo 4), *refabricação* é uma técnica de reorganização que simplifica o projeto (ou código) de um componente sem mudar sua função ou comportamento. Fowler [FOW99] define refabricação da seguinte maneira: "Refabricação é o processo de modificar um sistema de software de modo tal que ele não altere o comportamento externo do código [projeto], mas melhore a sua estrutura interna".

Quando o software é refabricado, o projeto existente é examinado quanto à redundância, elementos de projeto não usados, algoritmos ineficientes ou desnecessários, estruturas de dados impróprias ou pobremente construídas, ou qualquer outra falha de projeto que possa ser corrigida para se alcançar um projeto melhor. Por exemplo, uma primeira iteração de projeto pode produzir um componente que tem baixa coesão (isto é, ele executa três funções que têm apenas relacionamento limitado entre si). O projetista pode decidir que o componente deve ser refabricado em três componentes separados, cada um exibindo alta coesão. O resultado será um software mais fácil de integrar, mais fácil de testar e mais fácil de manter.

CASASEGURA

Conceitos de Projeto

A cena: A sala de Vinod, quando a modelagem de projeto tem início.

Os participantes: Vinod, Jamie e Ed — membros da equipe de engenharia de software do *CasaSegura*. Além de Shakira, um novo membro da equipe.

A conversa:

(Todos os quatro membros da equipe acabaram de voltar de um seminário matinal intitulado "Aplicação de Conceitos Básicos de Projeto", oferecido por uma professora local de Ciência da Computação.)

Vinod: Você aprendeu alguma coisa nova indo ao seminário?

Ed: Conhecia a maioria dos assuntos, mas suponho não ser uma idéia ruim ouvir isso novamente.

Jamie: Quando estudante de Ciência da Computação, nunca realmente entendi por que ocultamento de informação era tão importante quanto eles diziam ser.

Vinod: Porque... em última análise... é uma técnica para reduzir a propagação de erros em um programa. Na verdade, independência funcional também resulta na mesma coisa.

Shakira: Não fiz Ciência da Computação, de modo que muita coisa que o instrutor mencionou é nova pra mim. Posso gerar um código bom e rápido. Não vejo por que esse assunto seja tão importante.

Jamie: Tenho visto seu trabalho, Shak, e quer saber de uma coisa: você faz uma porção dessas coisas naturalmente... por causa disso é que seus projetos e código funcionam.

Shakira (sorrindo): Bem, eu sempre tento particionar o código, mantendo-o focalizado em algo, mantenho as interfaces simples e restritas, reuso código sempre que posso... essa espécie de coisas.

Ed: Modularidade, independência funcional, ocultamento, padrões... veja.

Jamie: Eu ainda me lembro do primeiro curso de programação que fiz ... eles nos ensinaram a refinar o código iterativamente.

Vinod: Alguma coisa pode ser aplicada ao projeto, você sabe.

Ed: O único conceito do qual não tinha ouvido antes é "refabricação".

Shakira: Isso é usado em Programação Extrema (*Extreme Programming*), acho que ela falou.

Ed: É. Não é tão diferente de refinamento, apenas você o faz depois que o projeto ou código esteja completado. Uma espécie de passo de otimização através do software, se você quer saber.

Jamie: Vamos voltar ao projeto do *CasaSegura*. Penso que deveríamos colocar esses conceitos na nossa lista de tarefas de revisão enquanto desenvolvemos um modelo de projeto do *CasaSegura*.

Vinod: Eu concordo, mas também é importante pensarmos sobre eles enquanto desenvolvemos o projeto.

9.3.9 Classes de Projeto

No Capítulo 8, notamos que o modelo de análise define um conjunto completo de classes de análise. Cada uma dessas classes descreve algum elemento do domínio do problema, focalizando aspectos do problema que são visíveis ao usuário ou ao cliente. O nível de abstração de uma classe de análise é relativamente alto.

À medida que o modelo de projeto evolui, a equipe de software deve definir um conjunto de *classes de projeto* que (1) refina as classes de análise, fornecendo detalhes de projeto que vão permitir que as classes sejam implementadas, e (2) crie um conjunto de classes de projeto que implemente uma infra-estrutura de software para apoiar a solução do negócio. Cinco diferentes tipos de classes de projeto, cada um representando uma camada diferente do projeto de arquitetura, são sugeridos [AMB01]:

? Que tipos de classe o projetista cria?

- *Classes de interface com o usuário* definem todas as abstrações necessárias para a interação humano-computador (IHC) (ou *human computer interaction*, HCI). Em muitos casos, IHC ocorre no contexto de uma *metáfora* (por exemplo, um talão de cheques, um formulário de pedido, uma máquina de fax), e as classes de projeto para a interface podem ser representações visuais dos elementos da metáfora.
- *Classes do domínio de negócio* são freqüentemente refinamentos de classes de análise definidas anteriormente. As classes identificam os atributos e serviços (métodos) necessários para implementar algum elemento do domínio de negócio.
- *Classes de processo* implementam abstrações de mais baixo nível de negócio necessárias para a completa gestão das classes de domínio de negócios.
- *Classes persistentes* representam depósitos de dados (por exemplo, um banco de dados) que vão persistir além da execução do software.
- *Classes de sistema* implementam funções de gestão e controle de software que habilitam o sistema a operar e se comunicar em seu ambiente e com o mundo externo.

Document shared on www.docsity.com

À medida que o modelo de projeto evolui, a equipe de software deve desenvolver um conjunto completo de atributos e operações para cada classe de projeto. O nível de abstração é reduzido à medida que cada classe de análise é transformada em uma representação de projeto. Classes de análise representam objetos (e serviços associados que são aplicados a eles) usando o jargão do domínio de negócios. Classes de projeto apresentam significativamente mais detalhe técnico como guia para implementação.

Arlow e Neustadt [ARL02] sugerem que cada classe de projeto seja revisada para garantir que ela esteja "bem formada". Eles definem quatro características de uma classe de projeto bem formada:

? O que é uma classe de projeto bem formada?

Completeza e suficiência. Uma classe de projeto deve ser o encapsulamento completo de todos os atributos e métodos que se podem razoavelmente esperar (com base na interpretação inteligente do nome da classe) que existam para a classe. Por exemplo, a classe **Cena** definida para o software de edição de vídeo somente fica completa se contiver todos os atributos e métodos que podem ser razoavelmente associados à criação de uma cena de vídeo. Suficiência garante que a classe de projeto contenha somente aqueles métodos que são suficientes para atingir o objetivo da classe, nem mais nem menos.

Primitivismo. Métodos associados a uma classe de projeto devem ser enfocados na realização de um serviço para a classe. Uma vez o serviço implementado com um método, a classe não deve fornecer um outro modo para realizar a mesma coisa. Por exemplo, a classe **VideoClipe** do software de edição de vídeo poderia ter atributos **ponto-inicial** e **ponto-final** para indicar os pontos inicial e final do clipe (observe que uma fita virgem carregada no sistema pode ser maior do que o clipe que é usado). Os métodos *ajustarPontoInicial()* e *ajustarPontoFinal()* fornecem os únicos meios para estabelecer os pontos inicial e final do clipe.

Alta coesão. Uma classe de projeto coesa tem um conjunto de responsabilidades pequeno e enfocado e aplica atributos e métodos especificamente para implementar aquelas responsabilidades. Por exemplo, a classe **VideoClipe** do software de edição de vídeo poderia conter um conjunto de métodos para edição de videoclipe. Contanto que cada método enfoque somente os atributos associados com o videoclipe, a coesão é mantida.

Baixo acoplamento. No modelo de projeto, é necessário projetar classes para colaborarem entre si. No entanto, a colaboração deve ser restrita a um mínimo aceitável. Se um modelo de projeto é altamente acoplado (todas as classes de projeto colaboram com todas as outras classes de projeto) o sistema é difícil de implementar, testar e manter ao longo do tempo. Em geral, classes de projeto em um subsistema deveriam ter apenas conhecimento limitado das classes de outros subsistemas. Essa restrição, chamada de *Lei de Demeter* [LIE03], sugere que um método deve somente enviar mensagens para métodos em classes vizinhas.[5]

CASASEGURA

Refinamento de uma Classe de Análise em uma Classe de Projeto

A cena: A sala de Ed, à medida que a modelagem de projeto continua.

Os personagens: Vinod e Ed — membros da equipe de engenharia de software do *CasaSegura*.

A conversa:

(Ed está trabalhando na classe **PlantaBaixa** [veja quadro de discussão na Seção 8.7.4 e Figura 8.1.4] e a refinou para o modelo de projeto.)

Ed: Então, você se lembra da classe **PlantaBaixa,** certo? Ela é usada como parte das funções de vigilância e gestão da casa.

Vinod (concordando com a cabeça): Sim, acho que me lembro de que a usamos como parte das nossas discussões CRC para gestão da casa.

Ed: Usamos. De qualquer jeito, eu a estou refinando para o projeto. Quero mostrar como vamos realmente implementar a classe **PlantaBaixa**. Minha idéia é implementá-la como um conjunto de listas encadeadas [uma estrutura de dados específica]. Então... tive que refinar a classe de análise **PlantaBaixa** (veja a Figura 8.1.4) e, na verdade, simplificá-la.

5 Um modo menos formal de citar a Lei de Demeter é "Cada unidade deve falar somente com seus amigos, ou seja, não falar com estranhos".

Vinod: A classe de análise mostrava somente coisas do domínio do problema, bem, na verdade, na tela do computador, o que era visível para o usuário final, certo?

Ed: É, mas na classe de projeto **PlantaBaixa** tive de adicionar algumas coisas específicas de implementação. Precisava mostrar que **PlantaBaixa** é uma agregação de segmentos, daí a classe **Segmento** — e que a classe **Segmento** é composta de listas de segmentos de parede, janelas, portas etc. A classe **Câmera** colabora com **PlantaBaixa** e obviamente podem existir muitas câmeras na planta baixa.

Vinod: Ufa, vamos ver uma figura dessa nova classe de projeto **PlantaBaixa**.

(Ed mostra a Vinod o desenho apresentado na Figura 9.3).

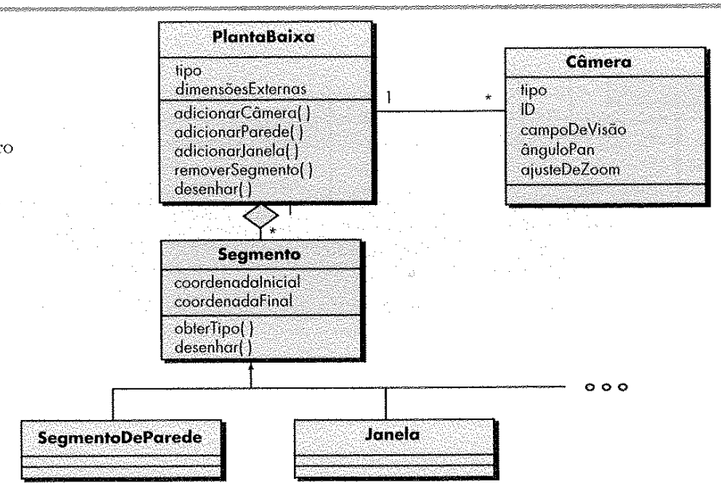
Vinod: Certo, entendo o que você está tentando fazer. Isso permite que você modifique a planta baixa facilmente porque novos itens podem ser adicionados ou removidos da lista — a agregação — sem quaisquer problemas.

Ed (concordando com a cabeça): É, acho que vai funcionar.

Vinod: Eu também.

FIGURA 9.3

Classe de projeto para PlantaBaixa e agregação composta para a classe (veja o quadro de discussão)



9.4 O MODELO DE PROJETO

PONTO CHAVE

O modelo de projeto tem quatro elementos principais: dados, arquitetura, componentes e interface.

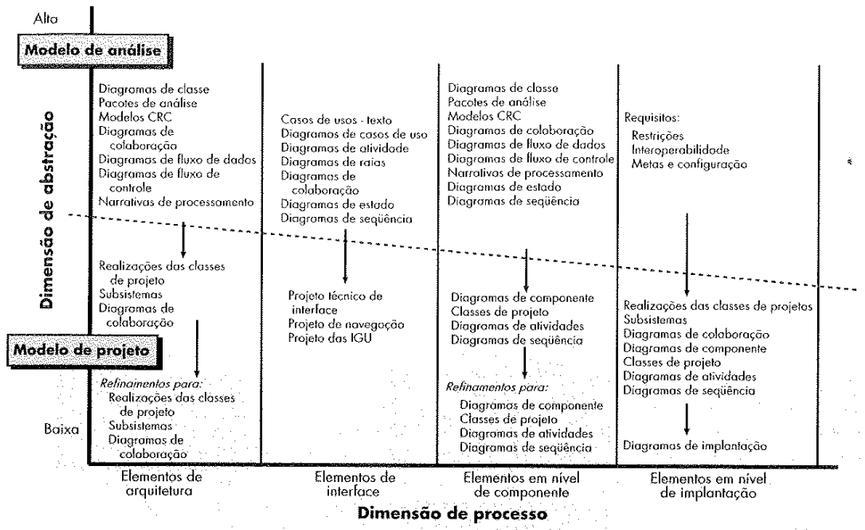
O *modelo de projeto* pode ser visto em duas dimensões diferentes como ilustrado na Figura 9.4. A dimensão *processo* indica a evolução do modelo de projeto à medida que as tarefas são executadas como parte do processo de software. A dimensão *abstração* representa o nível de detalhe à medida que cada elemento do modelo de análise é transformado em um projeto equivalente e, então, refinado iterativamente. Observando a figura, a linha pontilhada indica a fronteira entre os modelos de análise e projeto. Em alguns casos, é possível uma distinção clara entre os modelos de análise e projeto. Em outros casos, o modelo de análise paulatinamente se mistura ao modelo de projeto e uma distinção clara é menos óbvia.

Os elementos do modelo de projeto usam muitos dos mesmos diagramas UML que foram usados no modelo de análise. A diferença é que esses diagramas são refinados e elaborados como parte do projeto; mais detalhe específico de implementação é fornecido e são enfatizadas a estrutura e o estilo arquitetural, os componentes que residem na arquitetura e as interfaces entre os componentes e o mundo externo.

> "Questões sobre se o projeto é necessário ou acessível estão muito além do ponto: o projeto é inevitável. A alternativa para um projeto bom é um projeto ruim, não nenhum projeto."
>
> Douglas Martin

Document shared on www.docsity.com

FIGURA 9.4 Dimensões do modelo de projeto



É importante mencionar, no entanto, que os elementos do modelo anotados ao longo do eixo horizontal não são sempre desenvolvidos de forma seqüencial. Na maioria dos casos, o projeto de arquitetura preliminar prepara o palco e é seguido pelo projeto de interface e projeto em nível de componente, que freqüentemente ocorrem em paralelo. O modelo de implantação é usualmente adiado até que o projeto tenha sido totalmente desenvolvido.

PONTO CHAVE

No nível arquitetural (aplicação), o projeto de dados focaliza os arquivos ou banco de dados; no nível de componente, o projeto de dados considera as estruturas de dados necessárias para implementar os objetos de dados locais.

9.4.1 Elementos de Projeto de Dados

Como outras atividades de engenharia de software, *projeto de dados* (algumas vezes denominado *arquitetura de dados*) cria um modelo de dados e/ou informação que é representado no nível mais alto de abstração (a visão dos dados do cliente/usuário). Esse modelo de dados é então refinado em representações cada vez mais específicas da implementação que podem ser processadas pelo sistema baseado em computador. Em muitas aplicações de software, a arquitetura dos dados terá uma profunda influência na arquitetura do software que deve processá-los.

A estrutura de dados tem sido sempre uma parte importante do projeto de software. No nível de componente de programa, o projeto das estruturas de dados e dos algoritmos associados necessários para manipulá-las é essencial para a criação de aplicações de alta qualidade. No nível de aplicação, a tradução do modelo de dados (derivado como parte da engenharia de requisitos) para um banco de dados é fundamental para satisfazer aos objetivos de negócio de um sistema. Em nível de negócio, a coleção de informação armazenada em diferentes bancos de dados e reorganizada em um "armazém de dados" (*data warehouse*) possibilita a mineração de dados ou descoberta de conhecimento que pode ter impacto no sucesso do próprio negócio. Em qualquer caso, o projeto de dados desempenha uma papel importante. O projeto de dados será discutido em mais detalhe no Capítulo 10.

9.4.2 Elementos do Projeto Arquitetural

O *projeto arquitetural* do software é o equivalente à planta baixa de uma casa. A planta baixa representa a disposição global dos cômodos, seu tamanho, forma e relacionamento entre eles, e as portas e janelas que permitem movimento para dentro e para fora dos cômodos. A planta baixa nos dá uma visão geral da casa. Elementos de projeto arquitetural nos dão uma visão geral do software.

PONTO CHAVE

O modelo arquitetural é derivado do domínio de aplicação, do modelo de análise e dos estilos e padrões disponíveis.

> "Você pode usar uma borracha na prancheta de desenho ou uma marreta no canteiro de obra."
>
> **Frank Lloyd Wright**

O modelo arquitetural [SHA96] é derivado de três fontes: (1) informação sobre o domínio de aplicação para o software a ser construído; (2) elementos específicos do modelo de análise como diagramas de fluxo de dados ou classes de análise, seus relacionamentos e colaborações para o problema em pauta, e (3) a disponibilidade de padrões arquiteturais (Seção 9.5) e estilos (Capítulo 10).

9.4.3 Elementos de Projeto da Interface

O *projeto da interface* de software é equivalente a um conjunto de desenhos detalhados (e especificações) para as portas, janelas e instalações de uma casa. Esses desenhos representam o tamanho e a forma das portas e janelas, a maneira pela qual elas operam, o modo pelo qual as conexões com as redes públicas (água, eletricidade, gás, telefone) entram na casa e são distribuídas entre os cômodos representados na planta baixa. Eles nos contam onde a campainha da porta está localizada, se um interfone é para ser usado para anunciar a presença de uma visita e como o sistema de segurança deve ser instalado. Em essência, desenhos detalhados (e especificações) das portas, janelas e instalações domiciliares contam-nos como coisas e informação fluem para dentro e para fora da casa e pelos cômodos que são parte da planta baixa. Os elementos do projeto de interface de software contam-nos como a informação flui para dentro e para fora do sistema e como ela é disseminada entre os componentes definidos como parte da arquitetura.

> "O público está mais familiarizado com projeto ruim do que com projeto bom. Ele está, na verdade, condicionado a preferir projeto ruim porque é com o que ele convive. A novidade torna-se ameaçadora, o velho, tranqüilizador."
>
> **Paul Rand**

PONTO CHAVE

Há três partes nos elementos de projeto de interface: a interface com o usuário, interfaces com sistemas externos à aplicação e interfaces com componentes da aplicação.

Há três importantes elementos no projeto de interface: (1) a interface com o usuário (IU); (2) interfaces externas com outros sistemas, dispositivos, redes ou outros produtores ou consumidores de informação; e (3) interfaces internas entre vários componentes de projeto. Esses elementos do projeto de interface permitem ao software comunicar-se externamente e possibilitam comunicação interna e colaboração entre os componentes que compõem a arquitetura de software.

Projeto de IU é uma ação importante de engenharia de software e considerado em detalhes no Capítulo 12. Ele incorpora elementos estéticos (leiaute, cor, gráficos, mecanismos de interação), elementos ergonômicos (disposição e localização da informação, metáforas, navegação pela IU) e elementos técnicos (padrões de IU, componentes reusáveis). Em geral, a IU é um subsistema individual na arquitetura global da aplicação.

O projeto de interfaces externas requer informação definitiva sobre a entidade para qual a informação é enviada ou recebida. Em todo caso, essa informação deveria ser coletada durante a engenharia de requisitos (veja o Capítulo 7) e verificada quando o projeto de interface[6] começa. O projeto de interfaces externas deve incorporar verificação de erros e (quando necessário) características de segurança adequadas.

O projeto de interfaces internas é alinhado juntamente com o projeto de componentes (veja o Capítulo 11). As realizações de projeto das classes de análise representam todas as operações e esquemas de mensagem necessários para permitir comunicação e colaboração entre operações de várias classes. Cada mensagem deve ser projetada para acomodar a transferência de informação necessária e os requisitos funcionais específicos da operação solicitada.

Veja na Web

Informações extremamente valiosas sobre projeto de IU podem ser encontradas em www.useit.com.

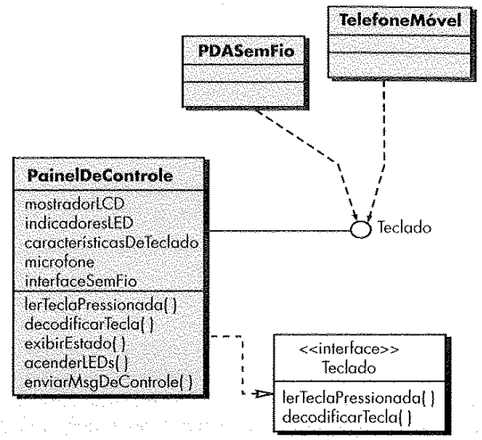
Em alguns casos, uma interface é modelada de modo muito parecido com o de uma classe. A UML define *interface* da seguinte maneira [OMG01]: "Uma interface é um especificador para operações [públicas] externamente visíveis de uma classe, componente ou outro classificador (inclusive subsistemas) sem especificação de estrutura interna". Dito de modo mais simples, uma interface é um conjunto de operações que descreve alguma parte do comportamento de uma classe e dá acesso a essas operações.

6 Não é incomum que as características de interface se modifiquem ao longo do tempo. Assim, um projetista deve garantir que a especificação da interface seja sempre atualizada.

Document shared on www.docsity.com

FIGURA 9.5

Representação UML da interface PainelDeControle



Por exemplo, a função de segurança *CasaSegura* faz uso de um painel de controle que permite ao proprietário controlar certos aspectos da função de segurança. Em uma versão avançada do sistema, funções de painel de controle podem ser implementadas via PDA sem fio ou telefone móvel.

A classe **PainelDeControle** (veja a Figura 9.5) fornece o comportamento associado com um teclado e, conseqüentemente, deve implementar operações *lerTeclaPressionada()* e *decodificarTecla()*. Se essas operações são fornecidas por outras classes (nesse caso, **PDASemFio** e **TelefoneMóvel**), é útil definir uma interface como mostra a figura. A interface **Teclado** é mostrada como um estereótipo <<interface>> ou como um círculo pequeno, rotulado, conectado à classe por uma linha. A interface é definida sem atributos e tem o conjunto de operações necessário para conseguir o comportamento de um teclado.

> "Um erro comum que as pessoas cometem quando tentam projetar algo completamente seguro tem sido subestimar a engenhosidade dos inteiramente doidos."
>
> Douglas Adams

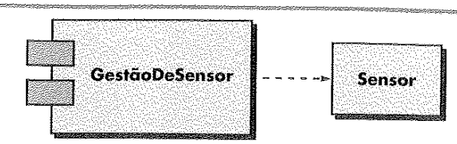
A linha pontilhada com um seta no final (Figura 9.5) indica que a classe **PainelDeControle** fornece operações de **Teclado** como parte do seu comportamento. Em UML, isso é caracterizado como *realização*. Parte do comportamento de **PainelDeControle** será implementada pela realização das operações de **Teclado**. Essas operações serão fornecidas a outras classes que têm acesso à interface.

9.4.4 Elementos de Projeto em Nível de Componente

O projeto em nível de componente para software é equivalente a um conjunto de desenhos detalhados (e especificações) para cada cômodo de uma casa. Esses desenhos representam a fiação e o encanamento dentro de cada cômodo, a localização de tomadas e interruptores, torneiras, pias, chuveiros, banheira, ralos, armários e *closet*. Eles também descrevem a pavimentação a ser usada, as molduras a ser aplicadas, e todos os outros detalhes associados a uma sala. O projeto em nível de componente do software descreve completamente os detalhes internos de cada componente de software. Para tanto, o projeto define estruturas de dados para todos os objetos de dados locais e detalhes algorítmicos para todo o processamento que ocorre em um componente e em uma interface que dá acesso a todas as operações do componente (comportamentos).

FIGURA 9.6

Diagrama de componente UML de GestãoDeSensor



> "Os detalhes não são detalhes. Eles fazem o projeto."
>
> Charles Eames

No contexto da engenharia de software orientada a objetos, um componente é representado nas formas diagramáticas UML como mostra a Figura 9.6. Nesta figura, está representado um componente chamado **GestãoDeSensor** (parte da função de segurança *CasaSegura*). A linha pontilhada conecta o componente a uma classe chamada **Sensor** que é associada a ele. O componente **GestãoDeSensor** realiza todas as funções associadas com os sensores de *CasaSegura* monitorando e configurando-os. (No Capítulo 11, é apresentada discussão adicional dos diagramas de componente.)

Os detalhes de projeto de um componente podem ser modelados em muitos níveis diferentes de abstração. Um diagrama de atividade pode ser usado para representar a lógica de processamento. Fluxo procedural detalhado de um componente pode ser representado por meio de pseudocódigo (representação semelhante a uma linguagem de programação descrita no Capítulo 11) ou de algumas formas diagramáticas (diagrama de atividade ou fluxograma).

9.4.5 Elementos de Projeto em Nível de Implantação

Elementos de projeto em nível de implantação indicam como a funcionalidade e os subsistemas do software serão alocados no ambiente computacional físico que vai apoiar o software. Por exemplo, os elementos do produto *CasaSegura* são configurados para operar nos três principais ambientes computacionais — um PC baseado na casa, o painel de controle de *CasaSegura* e um servidor alojado na CPI Corp. (fornecendo acesso baseado na Internet ao sistema).

Durante o projeto, um diagrama de implantação UML é desenvolvido e refinado como mostra a Figura 9.7; no qual três ambientes computacionais são apresentados (na verdade, deveria haver mais, inclusive sensores, câmeras e outros). Os subsistemas (funcionalidade) alojados em cada elemento computacional são indicados. Por exemplo, o PC aloja subsistemas que implementam segurança, vigilância, gestão residencial e características de comunicação. Além disso, um subsistema de acesso externo foi projetado para gerir todas as tentativas de acesso ao sistema *CasaSegura* de uma fonte externa. Cada subsistema seria elaborado para indicar os componentes que ele implementa.

O diagrama mostrado na Figura 9.7 está na *forma de descritor*. Isso significa que o diagrama de implantação mostra o ambiente computacional, mas não indica explicitamente detalhes de configuração. Por exemplo, o "computador pessoal" não é melhor identificado. Ele poderia ser um PC "Wintel" ou um Macintosh, uma estação de trabalho Sun ou uma caixa Linux. Esses detalhes são fornecidos quando o diagrama de implantação é revisitado na *forma de instância* durante estágios posteriores do projeto ou quando a construção começa. Cada instância da implantação (uma configuração de hardware específica, à qual é atribuído um nome) é identificada.

PONTO CHAVE

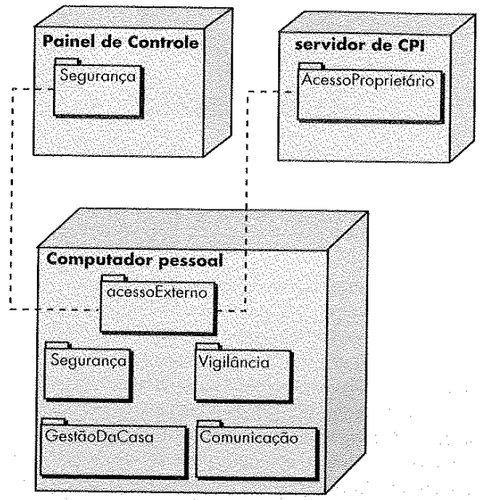
Diagramas de implantação começam na forma de descritor, em que o ambiente de implantação é descrito em termos gerais. Mais tarde, a forma de instância é usada e os elementos da configuração são explicitamente descritos.

> "De vez em quando saia, faça um pequeno relaxamento, de modo que quando voltar ao trabalho seu julgamento seja mais seguro. Afaste-se um pouco, porque então o trabalho parecerá menor, maior parte dele poderá ser abrangida por uma olhada, e uma falta de harmonia e proporção é mais prontamente vista."
>
> Leonardo DaVinci

Document shared on www.docsity.com

FIGURA 9.7

Diagrama de implantação UML



9.5 Projeto de Software Baseado em Padrão

Veja na Web

Se você precisa encontrar um padrão de projeto (ou outro) visite www.patterndepot.com/pages/.

Os melhores projetistas em qualquer campo têm uma habilidade incomum de ver padrões que caracterizam um problema e padrões correspondentes que podem ser combinados para criar uma solução. Durante todo o processo de projeto, um engenheiro de software deve procurar toda a oportunidade de reutilizar padrões de projeto existentes (quando eles satisfazem às necessidades do projeto) em vez de criar outros.

9.5.1 Descrição de um Padrão de Projeto

Disciplinas amadurecidas de engenharia fazem uso de milhares de padrões de projeto. Por exemplo, um engenheiro mecânico usa um eixo chaveado de dois passos como um padrão de projeto. Inerente aos padrões estão os atributos (os diâmetros do eixo, as dimensões do chaveamento etc.) e as operações (a rotação do eixo, a conexão do eixo). Um engenheiro eletricista usa um circuito integrado (padrão de projeto extremamente complexo) para resolver um elemento específico de um problema novo. Padrões de projeto podem ser descritos por meio do gabarito mostrado no quadro a seguir.

INFO

Gabarito de Padrão de Projeto

Nome do padrão — descreve a essência do padrão em um curto, mas expressivo, nome.

Intenção — descreve o padrão e o que ele faz.

Também-conhecido-como — lista os sinônimos do padrão.

Motivação — fornece um exemplo do problema.

Aplicabilidade — notifica situações específicas de projeto nas quais o padrão é aplicável.

Estrutura — descreve as classes necessárias para implementar o padrão.

Participantes — descreve as responsabilidades das classes que são necessárias para implementar o padrão.

Colaborações — descreve como os participantes colaboram para cumprir suas responsabilidades.

Conseqüências — descreve as "influências de projeto" que afetam o padrão e os potenciais compromissos que devem ser considerados quando o padrão é implementado.

Padrões relacionados — referências cruzadas relacionadas a padrões de projeto.

Uma descrição de padrão de projeto pode também considerar um conjunto de influências de projeto. *Influências de projeto* descrevem requisitos não funcionais (facilidade de manutenção, portabilidade) associados ao software ao qual o padrão deve ser aplicado. Além disso, as influências definem as restrições que podem restringir o modo pelo qual o projeto deve ser implementado. Em essência, as influências de projeto descrevem o ambiente e as condições que devem existir para tornar o padrão de projeto aplicável. As características do padrão (classes, responsabilidades e colaborações) indicam os atributos do projeto que podem ser ajustados para permitir ao padrão acomodar uma variedade de problemas. Esses atributos representam características do projeto que podem ser pesquisadas (por exemplo, com uma base de dados), de modo que um padrão adequado possa ser encontrado. Por fim, as diretrizes associadas com o uso de um padrão de projeto fornecem uma indicação das ramificações das decisões de projeto.

> "Padrões são semiprontos — o que significa que você sempre tem de acabá-los e adaptá-los ao seu próprio ambiente."
>
> **Martin Fowler**

PONTO CHAVE

Influências do projeto são as características do problema e atributos da solução que restringem o modo pelo qual o projeto pode ser desenvolvido.

Os nomes dos padrões de projeto devem ser escolhidos com cuidado. Um dos problemas técnicos chave no reuso de software é a inabilidade de encontrar padrões existentes reutilizáveis quando existem centenas de milhares de padrões candidatos. A busca pelo padrão "certo" é imensamente auxiliada por um nome de padrão significativo.

9.5.2 Uso de Padrões no Projeto

Padrões de projeto podem ser usados durante todo o projeto de software. Uma vez desenvolvido o modelo de análise (veja o Capítulo 8), o projetista pode examinar uma representação detalhada do problema a ser resolvido e as restrições impostas pelo problema. A descrição do problema é examinada em vários níveis de abstração para determinar se ela é responsável por um ou mais dos seguintes tipos de padrões de projeto:

? Que tipos de padrões de projeto estão disponíveis ao engenheiro de software?

Padrões Arquiteturais. Esses padrões definem uma estrutura global do software, indicam o relacionamento entre subsistemas e componentes do software e definem as regras para especificar relacionamentos entre os elementos (classes, pacotes, componentes, subsistemas) da arquitetura.

Padrões de Projeto. Esses padrões atendem a um elemento específico do projeto tal como uma agregação de componentes para resolver algum problema de projeto, relacionamentos entre componentes ou os mecanismos para efetuar a comunicação componente a componente.

Idiomas. Algumas vezes chamados de *padrões de código*, esses padrões específicos de linguagem geralmente implementam um elemento algorítmico de um componente, um protocolo específico de interface, ou um mecanismo para comunicação entre componentes.

Cada um desses tipos de padrão difere no nível de abstração com o qual é representado e no grau em que fornece instrução direta para a atividade de construção (nesse caso, codificação) do processo de software.

9.5.3 Frameworks

Em alguns casos pode ser necessário fornecer uma infra-estrutura do esqueleto de implementação específica, chamada de *arcabouço (framework)*, para o trabalho de projeto. O projetista pode selecionar uma *"miniarquitetura reusável* que fornece a estrutura e o comportamento genéricos para uma família de abstrações de software, dentro de um contexto (...) que especifica sua colaboração e uso em determinado domínio" [APP98].

Document shared on www.docsity.com

PONTO CHAVE

Arcabouço é um esqueleto de código que pode ser preenchido com classes ou funcionalidade específicas projetadas para atender ao problema em mãos.

Um arcabouço não é um padrão arquitetural, mas um esqueleto com uma coleção de "pontos conectáveis" (também chamados de *ganchos* [*hooks*] e *encaixes* [*slots*]) que lhe possibilitam adaptar-se a um domínio específico de problema. Os pontos de conexão possibilitam a um projetista integrar classes ou funcionalidades específicas do problema no esqueleto. Em um contexto orientado a objetos, arcabouço é uma coleção de classes em cooperação.

Em resumo, o projetista de um arcabouço alegará que uma miniarquitetura reusável é aplicável a todo o software a ser desenvolvido em um domínio de aplicação limitado. Para serem mais efetivos, arcabouços são aplicados sem modificações. Elementos adicionais de projeto podem ser adicionados, mas somente via pontos de conexão que permitem ao projetista preencher o esqueleto do arcabouço.

9.6 Resumo

A engenharia de projeto começa quando a primeira iteração da engenharia de requisitos é concluída. O objetivo do projeto de software é aplicar um conjunto de princípios, conceitos e práticas que leva ao desenvolvimento de um sistema ou produto de alta qualidade; é criar um modelo de software que vai implementar todos os requisitos do cliente corretamente e trazer satisfação àqueles que o utilizam. A engenharia de projeto deve examinar várias alternativas de projeto e convergir para uma solução que melhor satisfaça às necessidades dos interessados no projeto.

O processo de projeto vai de uma visão "global" do software a uma visão mais concentrada que define o detalhe necessário para implementar um sistema. O processo começa enfocando a arquitetura. Subsistemas são definidos, mecanismos de comunicação entre subsistemas são estabelecidos; componentes são identificados; e uma descrição detalhada de cada componente é desenvolvida. Adicionalmente, interfaces externas, internas e com o usuário são projetadas.

Conceitos de projeto têm evoluído durante a primeira metade de século de trabalho de engenharia de software. Eles descrevem atributos de software de computador que devem estar presentes independentemente do processo de engenharia de software escolhido, dos métodos de projeto aplicados ou das linguagens de programação usadas.

O modelo de projeto inclui quatro diferentes elementos. À medida que cada um deles é desenvolvido, uma visão mais completa do projeto evolui. O elemento arquitetural usa informação derivada do domínio de aplicação, o modelo de análise, catálogos de padrões e estilos disponíveis para derivar uma representação estrutural completa do software, seus subsistemas e componentes. Elementos de projeto de interface modelam as interfaces externas e internas e a interface com o usuário. Elementos no nível de componente definem cada um dos módulos (componentes) que compõem a arquitetura. Por fim, elementos de projeto no nível de implantação alocam a arquitetura, seus componentes e interfaces à configuração física que vai alojar o software.

Projeto baseado em padrão é uma técnica que reusa elementos de projeto que foram provados com sucesso no passado. Cada padrão arquitetural, padrão de projeto ou idioma é catalogado, profundamente documentado e cuidadosamente considerado à medida que ele é avaliado para inclusão em uma aplicação específica. Arcabouço, uma extensão de padrões, fornece um esqueleto arquitetural para o projeto de subsistemas completos em um domínio de aplicação específico.

Referências Bibliográficas

[AMB01] Ambler, S., *The Object Primer*, Cambridge Univ. Press, 2ª ed., 2001.

[APP98] Appleton, B., "Patterns and Software: Essential Concepts and Terminology", disponível em http://www.enteract.com/~bradapp/docs/patterns-intro.html.

[ARL02] Arlow, J., Neustadt, I., *UML and the Unified Process*, Addison-Wesley, 2002.

[BEL81] Belady, L., Foreword to *Software Design: Methods and Techniques* (L. J. Peters, autor), Yourdon Press, 1981.

[FOW00] Fowler, M., *et al*, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2000.

[GAM95] Gamma, E., *et al*, *Design Patterns*, Addison-Wesley, 1995.

[GAR95] Garlan, D. e Shaw, M., "An Introduction to Software Architecture", *Advances in Software Engineering and Knowledge Engineering*, v. I (V. Ambriola e G. Tortora, editores), World Scientific Publishing Company, 1995.

[GRA87] Grady, R. B. e D. L. Caswell, *Software Metrics: Establishing a Company-Wide Program*, Prentice-Hall, 1987.

[JAC75] Jackson, M. A., *Principles of Program Design*, Academic Press, 1975.

[LIE03] Lieberherr, K., "Demeter: Aspect-Oriented Programming", maio 2003, disponível em http://www.ccs.neu.edu/home/lieber/LoD.html.

[MAI03] Maioriello, J., "What Are Design Patterns and Do I Need Them?", developer.com, 2003, disponível em http://www.developer.com/design/article.php/ 1474561.

[MCG91] McGlaughlin, R., "Some Notes on Program Design," *Software Engineering Notes*, v. 16, n. 4, p. 53-54, out. 1991.

[MYE78] Myers, G., *Composite Structured Design*, Van Nostrand,1978.

[OMG01] Object Management Group, *OMG Unified Modeling Language Specification*, versão 1.4, set. 2001.

[PAR72] Parnas, D. L., "On Criteria to Be Used in Decomposing Systems into Modules", *CACM*, v. 14, n. I, p. 221-227, abr. 1972.

[ROS75] Ross, D., Goodenough, J. e Irvine, C., "Software Engineering: Process, Principles and Goals", *IEEE Computer*, v. 8, n. 5, maio 1975.

[SCH02] Schmuller, J., *Teach Yourself UML*, SAMS Publishing, 2002.

[SHA96] Shaw, M. e Garlan, D., *Software Architecture*, Prentice-Hall, 1996.

[STA02] "Metaphor," *The Stanford HCI Learning Space*, 2002, disponível em http://hci.stanford.edu/ hcils/concepts/metaphor.html.

[STE74] Stevens, W., Myers, G. e Constantine, L., "Structured Design", *IBM Systems Journal*, v. 13, n. 2, p. 115-139, 1974.

[WIR71] Wirth, N., "Program Development by Stepwise Refinement", *CACM*, v. 14, n. 4, p. 221-27, 1971.

Problemas e Pontos a Considerar

9.1. Você projeta software quando "escreve" um programa? O que faz o projeto de software diferente da codificação?

9.2. Se um projeto de software não é um programa (e não é), então é o quê?

9.3. Como avaliamos a qualidade de um projeto de software?

9.4. Examine o conjunto de tarefas apresentado para o projeto. Onde a qualidade é avaliada nesse conjunto de tarefas? Como isso é conseguido?

9.5. Forneça exemplos de três abstrações de dados e das abstrações procedurais que podem ser usadas para manipulá-las.

9.6. Descreva a arquitetura de software com suas próprias palavras.

9.7. Sugira um padrão de projeto que você encontra em uma categoria de coisas de uso diários (eletrônica voltada para o consumidor, automóveis, eletrodomésticos). Documente o padrão completamente usando o gabarito fornecido na Seção 9.5.

9.8. Existe algum caso em que problemas complexos exigem menos esforço para solução? Como tal caso poderia afetar o argumento a favor da modularidade?

9.9. Quando um projeto modular deve ser implementado como um software monolítico? Como isso pode ser obtido? Desempenho é a única justificativa para a implementação de software monolítico?

9.10. Discuta o relacionamento entre o conceito de ocultamento da informação como atributo de efetiva modularidade e o conceito de independência de módulo.

9.11. Como os conceitos de acoplamento e portabilidade de software estão relacionados? Forneça exemplos para apoiar sua discussão.

9.12. Aplique uma "abordagem de refinamentos sucessivos" para desenvolver três níveis diferentes de abstração procedural para um ou mais dos seguintes programas:

a) um preenchedor de cheques que, dada uma quantia numérica, imprima-a por extenso, padrão normalmente exigido em um cheque;

b) a resolução iterativa das raízes de uma equação transcendental;

c) um algoritmo simples de escalonamento de tarefas para um sistema operacional.

Document shared on www.docsity.com

9.13. Faça alguma pesquisa sobre Programação Extrema e redija um trabalho curto sobre o uso de refabricação no processo de desenvolvimento ágil de software.

9.14. Visite um repositório de padrões de projeto (na Web) e gaste alguns minutos navegando pelos padrões. Escolha um e apresente-o à sua classe.

Leituras e Fontes de Informação Adicionais

Donald Norman escreveu dois livros (*The Design of Everyday Things*, Doubleday, 1990, e *The Psychology of Everyday Things*, HarperCollins, 1988) que se tornaram clássicos da literatura de projeto e "devem" ser lidos por todos os que projetam qualquer coisa que os humanos usam. Adams (*Conceptual Blockbusting*, terceira edição, Addison-Wesley, 1986) escreveu um livro que é de leitura essencial para os projetistas que querem ampliar o seu modo de pensar. Por fim, um texto clássico por Polya (*How to Solve It*, Princeton University Press, segunda edição, 1988) fornece um processo genérico problema-solução que pode ajudar os projetistas de software quando estão diante de problemas complexos.

Seguindo a mesma tradição, Winograd *et al.* (*Bringing Design to Software*, Addison-Wesley, 1996) discutem projetos de software que funcionam ou não e por quê. Um livro fascinante editado por Wixon e Ramsey (*Field Methods Casebook for Software Design*, Wiley, 1996) sugere métodos de pesquisa de campo (muito parecidos com aqueles usados por antropólogos) para entender como os usuários finais fazem o trabalho que eles executam, e depois fornece diretrizes para projetar software que atende a suas necessidades. Beyer e Holtzblatt (*Contextual Design: A Customer-Centered Approach to Systems Designs*, Academic Press, 1997) oferecem uma outra visão de projeto de software que integra o cliente/usuário em todos os pontos do processo de projeto de software.

McConnell (*Code Complete*, Microsoft Press, 1993) apresenta uma excelente discussão de tópicos práticos de projeto de software de computador de alta qualidade. Robertson (*Simple Program Design*, terceira edição, Boyd and Fraser Publishing, 1999) oferece uma discussão introdutória de projeto de software que é útil para aqueles que estão começando seus estudos no assunto. Fowler e seus colegas (*Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999) discutem técnicas para otimização incremental de projetos de software.

Na década passada, muitos livros sobre projeto baseado em padrões foram escritos por engenheiros de software. Gamma e seus colegas [GAM95] escreveram um livro pioneiro sobre o assunto. Outros livros escritos por Douglass (*Real-Time Design Patterns*, Addison-Wesley, 2002), Metsker (*Design Patterns Java Workbook*, Addison-Wesley, 2002), Juric *et al.* (*J2EE Design Patterns Applied*, Wrox Press, 2002), Marinescu e Roman (*EJB Design Patterns*, Wiley, 2002), e Shalloway e Trott (*Design Patterns Explained*, Addison-Wesley, 2001) discutem padrões de projeto em aplicações e ambientes de linguagens específicos. Além disso, livros clássicos do arquiteto Christopher Alexander (*Notes on the Synthesis of Form*, Harvard University Press, 1964 e *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, 1977) são muito lidos por projetistas de software que pretendem entender completamente padrões de projeto.

Uma ampla variedade de fontes de informação sobre engenharia de projeto está disponível na Internet. Uma lista atualizada de referências que são relevantes pode ser encontrada no site deste livro em **http://www.mhhe.com/pressman.**

Projeto Arquitetural

CAPÍTULO 10

Conceitos-Chave

ADL 223
arquétipos 218
arquitetural 223
avaliação 221
complexidade 223
definição 208
diagrama de contexto .. 217
mapeamento 224
padrões 215
projeto 216
ATAM 221
componentes 218
estilos 211
fatoração 227
projeto de dados 209

O projeto tem sido descrito como um processo de vários passos no qual representações da estrutura do programa, características de interface e detalhes procedurais são sintetizados com base nos requisitos de informação. Essa descrição é estendida por Freeman [FRE80]:

> Projeto é uma atividade preocupada com a tomada de decisões importantes, freqüentemente de natureza estrutural. Ele compartilha com a programação uma preocupação em abstrair a representação da informação e de seqüências de processamento, mas o nível de detalhe é bastante diferente nos extremos. O projeto constrói representações de programas coerentes e bem planejadas que se concentram nos inter-relacionamentos das partes em alto nível e nas operações lógicas envolvidas nos níveis inferiores.

Como mencionamos no Capítulo 9, projeto é orientado por informação. Métodos de projeto de software são derivados pela consideração de cada um dos três domínios do modelo de análise. Os domínios informacional, funcional e comportamental servem de guia para a criação do projeto de software.

Neste capítulo, apresentamos os métodos necessários para criar "representações coerentes e bem planejadas" das camadas de dados e arquitetural do modelo de projeto. O objetivo é fornecer uma abordagem sistemática para a derivação do projeto arquitetural — a documentação preliminar por meio da qual o software é construído.

Panorama

O que é? Projeto arquitetural representa a estrutura dos componentes de dados e programas que são necessários para construir um sistema baseado em computador. Ele considera o estilo arquitetural que o sistema vai adotar, a estrutura e as propriedades dos componentes que constituem o sistema e os inter-relacionamentos que ocorrem entre todos os componentes arquiteturais de um sistema.

Quem faz? Apesar de um engenheiro de software poder projetar tanto os dados quanto a arquitetura, o trabalho é freqüentemente distribuído entre especialistas, quando sistemas grandes e complexos precisam ser construídos. Um projetista de banco de dados ou centro de dados (data warehouse) cria a arquitetura de dados para um sistema. O "arquiteto de sistema" seleciona um estilo arquitetural adequado aos requisitos derivados durante a engenharia de sistemas e a análise de requisitos do software.

Por que é importante? Você não tentaria construir uma casa sem uma planta, tentaria? Você também não começaria a desenhar as plantas esboçando a disposição dos encanamentos da casa. Você precisaria olhar o quadro geral — a casa em si — antes de se preocupar com os detalhes. Isso é o que projeto arquitetural faz — fornece o quadro geral e garante que você faça direito.

Quais são os passos? O projeto arquitetural começa com o projeto dos dados e depois prossegue para a derivação de uma ou mais representações da estrutura arquitetural do sistema. Estilos ou padrões arquiteturais alternativos são analisados para derivar a estrutura mais adequada aos requisitos do cliente e aos atributos de qualidade. Uma vez selecionada uma alternativa, a arquitetura é elaborada usando um método de projeto arquitetural.

Qual é o produto do trabalho? Durante o projeto arquitetural é criado um modelo abrangendo a arquitetura dos dados e a estrutura do programa. Além disso, são descritos propriedades e relacionamentos (interações) de componentes.

Como tenho certeza de que fiz corretamente? Em cada estágio, produtos de trabalho do projeto de software são revisados quanto à clareza, correção, completeza e consistência com os requisitos e entre si.

Document shared on www.docsity.com

10.1 Arquitetura de Software

No seu livro, que constitui um referencial sobre o assunto, Shaw e Garlan [SHA96] discutem arquitetura de software da seguinte maneira:

> Desde quando o primeiro programa foi dividido em módulos, os sistemas de software passaram a ter arquiteturas, e os programadores têm sido responsáveis pelas interações entre os módulos e as propriedades globais da montagem. Historicamente, as arquiteturas eram implícitas — acidentes de implementação ou sistemas herdados do passado. Bons desenvolvedores de software têm adotado freqüentemente um ou mais padrões arquiteturais como estratégias para a organização de sistemas, mas eles usam esses padrões informalmente e não têm meios para torná-los explícitos no sistema resultante.

Hoje, a arquitetura de software efetiva e a sua representação explícita tornaram-se temas dominantes em engenharia de software.

> "A arquitetura de um sistema é um arcabouço abrangente que descreve sua forma e estrutura — seus componentes e como eles se articulam."
>
> **Jerrold Grochow**

10.1.1 O Que É Arquitetura?

Quando discutimos a arquitetura de um edifício, muitos atributos diferentes vêm à mente. No nível mais simplista, consideramos a forma geral da estrutura física mas, na realidade, arquitetura é muito mais. É a maneira pela qual os vários componentes do edifício são integrados para formar um todo coeso. É o modo pelo qual o edifício se ajusta ao seu ambiente e se mistura com os outros edifícios da vizinhança. É o grau com que o edifício alcança sua finalidade declarada e satisfaz às necessidades do seu proprietário. É o sentido estético da estrutura — o impacto visual do edifício — e o modo pelo qual texturas, cores e materiais são combinados para criar a fachada externa e o "ambiente de convivência" interno. São pequenos detalhes — o projeto de dispositivos de iluminação, tipo de piso, colocação de ornatos de paredes, a lista é quase sem fim. E, por fim, é arte.

PONTO CHAVE

Arquitetura de software deve modelar a estrutura de um sistema e a maneira pela qual componentes de dados e procedimentais colaboram entre si.

Mas e quanto à *arquitetura de software*? Bass, Clements e Kazman [BAS03] definem esse termo vago do seguinte modo:

> A arquitetura de software de um programa ou sistema computacional é a estrutura ou estruturas do sistema que abrange os componentes de software, as propriedades externamente visíveis desses componentes e as relações entre eles.

A arquitetura não é o software operacional. Ao contrário, é a representação que permite ao engenheiro de software (1) analisar a efetividade do projeto em satisfazer a seus requisitos declarados, (2) considerar alternativas arquiteturais em um estágio em que fazer modificações de projeto é ainda relativamente fácil, e (3) reduzir os riscos associados à construção do software.

> "Case depressa com a sua arquitetura, arrependa-se quando quiser."
>
> **Barry Boehm**

Veja na Web

Indicadores úteis para muitos sites de arquitetura de software podem ser encontrados em www2.umassd.edu/SECenter/SAResources.html.

Essa definição enfatiza o papel dos "componentes de software" em qualquer representação arquitetural. No contexto do projeto arquitetural, um componente de software pode ser algo tão simples quanto um módulo de programa ou uma classe orientada a objetos, mas também pode ser ampliado para incluir bancos de dados e *middleware*, que permite a configuração de uma rede de clientes e servidores.

Neste livro, o projeto da arquitetura de software considera dois níveis da pirâmide de projeto (veja a Figura 9.1) — *projeto de dados* e *projeto arquitetural*. No contexto da discussão anterior, projeto de dados nos permite representar o componente de dados da arquitetura em sistemas convencionais e definições de classe (encapsulando atributos e operações) em sistemas orientados a objetos. O projeto arquitetural focaliza a representação da estrutura de componentes de software, suas propriedades e interações.

10.1.2 Por Que a Arquitetura É Importante?

PONTO CHAVE

O modelo arquitetural fornece uma visão do sistema segundo o gestaltismo, permitindo ao engenheiro de software examiná-lo em seu todo.

Em um livro dedicado à arquitetura de software, Bass e seus colegas [BAS03] identificam três razões-chave pelas quais a arquitetura de software é importante:

- Representações da arquitetura de software constituem um facilitador da comunicação entre todas as partes interessadas (envolvidas) no desenvolvimento de um sistema baseado em computador.
- A arquitetura destaca decisões iniciais de projeto que terão um impacto profundo em todo o trabalho de engenharia de software que se segue e, igualmente importante, no sucesso final do sistema como uma entidade operacional.
- A arquitetura "constitui um modelo relativamente pequeno, intelectualmente inteligível de como o sistema é estruturado e como seus componentes trabalham em conjunto" [BAS03].

O modelo do projeto arquitetural e os padrões arquiteturais nele contidos são transferíveis. Estilos e padrões arquiteturais (veja a Seção 10.3.1) podem ser aplicados ao projeto de outros sistemas e representam um conjunto de abstrações que permitem a engenheiros de software descrever a arquitetura de modo previsível.

10.2 Projeto de Dados

A ação do *projeto de dados* interpreta objetos de dados definidos como parte do modelo de análise (veja o Capítulo 8) em estruturas de dados para os componente de software e, quando necessário, uma arquitetura de banco de dados para a aplicação. Em algumas situações, um banco de dados deve ser projetado e construído especificamente para um novo sistema. Em outros, no entanto, um ou mais bancos de dados existentes são usados.

10.2.1 Projeto de Dados Arquitetural

Hoje, pequenos e grandes negócios estão inundados por dados. Não é fora do comum, mesmo para um negócio de tamanho moderado, ter dúzias de bancos de dados servindo várias aplicações abrangendo centenas de gigabytes de dados. O desafio é extrair informação útil desse ambiente de dados, particularmente quando a informação desejada é transfuncional (informação que pode ser obtida apenas se dados específicos de comercialização são cruzados com dados de engenharia de produto).

> "Qualidade de dados é a diferença entre um armazém de dados e um lixão de dados."
>
> **Jarrett Rosenberg**

Veja na Web

Informação sobre tecnologias de armazém de dados pode ser obtida em www.datawarehouse.com.

Para enfrentar esse desafio, a comunidade de TI (tecnologia da informação) para a área de negócios desenvolveu técnicas de *mineração de dados* (*data mining*), também chamadas de *descoberta de conhecimento em bancos de dados* (*knowledge discovery in databases*, KDD), que navegam por bancos de dados existentes em uma tentativa de extrair informação apropriada para o negócio. No entanto, a existência de múltiplos bancos de dados, suas estruturas diferentes, o grau de detalhe contido nos bancos de dados e muitos outros fatores tornam a mineração de dados difícil em um ambiente de bancos de dados existente. Uma solução alternativa, chamada de *armazém de dados* (*data warehouse*), acrescenta uma camada adicional à arquitetura dos dados.

Um armazém de dados é um ambiente de dados separado que não está diretamente integrado com as aplicações cotidianas, mas abrange todos os dados usados por um negócio [MAT96]. Em

Document shared on www.docsity.com

um certo sentido, o armazém de dados é um banco de dados de grande porte e independente que tem acesso aos dados armazenados em bancos de dados que servem ao conjunto de aplicações necessárias para um negócio.

Uma discussão detalhada do projeto de estruturas de dados, bancos de dados e armazém de dados é melhor tratada em livros dedicados a esses assuntos (por exemplo, [DAT00], [PRE98], [KIM98]). O leitor interessado deve ver a seção *Leituras e Fontes de Informação Adicionais* deste capítulo para mais referências.

FERRAMENTAS DE SOFTWARE

Mineração/Armazéns de Dados

Objetivo: Ferramentas de mineração de dados suportam a identificação de relacionamentos significativos entre atributos que descrevem um objeto de dados específico ou um conjunto de objetos de dados. Ferramentas de armazém de dados suportam o projeto de modelos de dados para um armazém de dados.

Mecânica: A mecânica das ferramentas varia. Em geral, ferramentas de mineração aceitam grandes conjuntos de dados como entrada e permitem ao usuário consultar os dados em um esforço de melhor entender os relacionamentos entre os vários itens de dados. Ferramentas de armazém de dados usadas para projeto fornecem relacionamentos de entidades ou outras habilidades de modelagem.

Ferramentas Representativas[1]

Mineração de Dados

Business Objects, desenvolvida por Business Objects, SA (www.business.objects.com), é um conjunto de ferramentas de projeto de dados que suporta a "integração de dados, consulta, relatório, análise e analítica".

SPSS, desenvolvida por SPSS, Inc. (www.spss.com) fornece um vasto conjunto de funções estatísticas que permitem a análise de grandes conjuntos de dados.

Armazéns de Dados

Industry Warehouse Studio, desenvolvida pela Sybase (www.sybase.com), fornece uma infra-estrutura empacotada de armazém de dados que "deflagra" o projeto de armazém de dados.

IFW Business Inteligence Suite, desenvolvida por Modelware (www.modelwarepl.com) é um conjunto de modelos, ferramentas de software e projetos de banco de dados que "fornecem um caminho rápido para o projeto e implementação de armazéns e entrepostos de dados".

Uma lista abrangente de ferramentas e recursos de mineração/armazém de dados pode ser encontrada em Data Warehousing Information Center (www.dwinfocenter.org).

10.2.2 Projeto de Dados no Nível de Componentes

O projeto de dados de componentes focaliza a representação de estruturas de dados que são diretamente acessíveis a um ou mais componentes de software. Wasserman [WAS80] propôs um conjunto de princípios que pode ser usado para especificar e projetar tais estruturas de dados. Na verdade, o projeto de dados começa durante a criação do modelo de análise. Lembrando que a análise de requisitos e o projeto freqüentemente têm alguma superposição, consideremos o seguinte conjunto de princípios (adaptado de [WAS80]) para especificação dos dados:

? Que princípios são aplicáveis ao projeto de dados?

1. *Os princípios sistemáticos de análise aplicados à função e ao comportamento devem também ser aplicados aos dados.* Representações de fluxo de dados e conteúdo devem também ser desenvolvidas e revisadas, objetos de dados devem ser identificados, organizações alternativas de dados devem ser consideradas e o impacto da modelagem de dados no projeto de software deve ser avaliado.
2. *Todas as estruturas de dados e as operações a ser realizadas em cada uma devem ser identificadas.* O projeto de uma estrutura de dados eficiente deve levar em consideração as operações a ser realizadas na estrutura de dados.
3. *Um mecanismo para definir o conteúdo de cada objeto de dados deve ser estabelecido e usado para definir tanto os dados quanto as operações aplicadas a ele.* Diagramas de classe (veja o Capítulo 8) definem os itens de dados (atributos) contidos em uma classe e o processamento (operações) aplicado a esses itens de dados.
4. *Decisões de baixo nível de projeto de dados devem ser adiadas até mais adiante no processo de projeto.* Um processo de refinamento passo a passo pode ser usado para o projeto dos dados. A organização geral de dados pode ser definida durante a análise de requisitos, refinada durante o trabalho de projeto de dados e especificada em detalhes durante o projeto de componentes.
5. *A representação da estrutura de dados deve ser conhecida apenas por aqueles módulos que precisam fazer uso direto dos dados contidos na estrutura.* O conceito de ocultamento da informação e o conceito relacionado a acoplamento (veja o Capítulo 9) fornecem conhecimentos importantes quanto à qualidade de um projeto de software.
6. *Uma biblioteca de estruturas de dados úteis e das operações que podem ser aplicadas a elas deve ser desenvolvida.* Uma biblioteca de classes consegue isso.
7. *O projeto de software e a linguagem de programação devem suportar a especificação e a realização dos tipos abstratos de dados.* A implementação de uma estrutura de dados sofisticada pode ficar demasiadamente difícil, se não houver meios para especificar diretamente a estrutura na linguagem de programação escolhida para a implementação.

Esses princípios formam a base para uma abordagem de projeto de dados de componentes que pode ser integrada tanto nas atividades de análise quanto de projeto.

1 Os produtos mencionados aqui não representam uma recomendação, mas, em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

10.3 Estilos e Padrões Arquiteturais

Quando um construtor usa a frase "colonial americano com *hall* central" para descrever uma casa, a maioria das pessoas familiarizadas com casas nos Estados Unidos consegue conceber uma imagem geral do aspecto que a casa vai ter e de como será a disposição dos cômodos. O construtor usou um *estilo arquitetural* como mecanismo descritivo para diferenciar a casa de outros estilos (mediterrâneo, colonial brasileiro etc). Porém, mais importante, o estilo arquitetural é também um gabarito para construção. Mais detalhes da casa precisam ser definidos, suas dimensões finais precisam ser especificadas, características sob medida precisam ser adicionadas, os materiais de construção precisam ser determinados, mas o estilo — "colonial americano com hall central" — guia o construtor no seu trabalho.

> "Existe por trás da mente de todo artista um padrão ou tipo de arquitetura."
>
> G. K. Chesterton

? O que é um estilo arquitetural?

O software construído para sistemas baseados em computador também exibe um ou vários estilos arquiteturais. Cada estilo descreve uma categoria de sistemas que abrange (1) um conjunto de componentes (um banco de dados, módulos computacionais) que realizam a função que é requisito do sistema; (2) um conjunto de conectores que possibilita "comunicação, coordenação e cooperação" entre os componentes; (3) restrições que definem como os componentes podem ser integrados para formar o sistema; e (4) modelos semânticos que possibilitam ao projetista entender as propriedades gerais de um sistema pela análise das propriedades conhecidas de duas partes constitutivas [BAS03].

Um estilo arquitetural é uma transformação imposta sobre o projeto de um sistema completo. O objetivo é estabelecer uma estrutura para todos os componentes do sistema. No caso em que uma arquitetura existente deve ser submetida a reengenharia (veja o Capítulo 31), a imposição de um estilo arquitetural resultará em modificações fundamentais na estrutura do software, inclusive uma reatribuição da funcionalidade dos componentes [BOS00].

Um *padrão arquitetural*, como um estilo arquitetural, impõe uma transformação no projeto de arquitetura. No entanto, o padrão difere do estilo em um certo número de modos fundamentais: (1) o escopo de um padrão é menos amplo, enfoca um aspecto da arquitetura em vez de toda a arqui-

Document shared on www.docsity.com

Veja na Web

Estilos arquiteturais baseados em atributo (attribute-based architectural styles, ABAS) podem ser usados como blocos construtivos de arquiteturas de software. Mais informações podem ser obtidas em www.sei.cmu.edu/ata/abas.html.

tetura; (2) um padrão impõe uma regra na arquitetura, descrevendo como o software vai manipular em algum tópico de sua funcionalidade em termos de infra-estrutura (por exemplo, concorrência) [BOS00]; (3) padrões arquiteturais tendem a atender tópicos comportamentais específicos no contexto arquitetural, por exemplo, como uma aplicação de tempo real manipula sincronização ou interrupções. Padrões podem ser usados em conjunto com um estilo arquitetural para estabelecer a forma da estrutura global de um sistema. Na seção seguinte, consideramos estilos e padrões arquiteturais comumente usados para software.

10.3.1 Uma Breve Taxinomia de Estilos Arquiteturais

Apesar de milhões de sistemas baseados em computador terem sido criados ao longo dos últimos 50 anos, a grande maioria pode ser categorizada (veja [SHA96], [BUS96], [BAS03]) em um dos relativamente poucos estilos arquiteturais:

Arquitetura centrada nos dados. Um depósito de dados (por exemplo, um arquivo ou banco de dados) fica no centro dessa arquitetura e dá acesso freqüentemente a outros componentes que atualizam, adicionam, retiram ou modificam de outra forma os dados contidos no depósito. A Figura 10.1 ilustra um estilo típico centrado nos dados. Software-cliente tem acesso a um repositório central. Em alguns casos o repositório de dados é passivo, isto é, o software-cliente tem acesso aos dados independentemente de quaisquer modificações nos dados ou das ações dos outros softwares-clientes. Uma variante dessa abordagem transforma o repositório em um "quadro-negro", que envia notificações ao software-cliente quando dados de seu interesse sofrem modificações.

Arquiteturas centradas em dados promovem *integrabilidade* [BAS03]. Componentes existentes podem ser modificados e novos componentes-clientes podem ser adicionados à arquitetura sem preocupação com os outros clientes (porque os componentes-clientes operam independentemente). Além disso, dados podem ser passados entre clientes usando o mecanismo de quadro-negro (o componente quadro-negro serve para coordenar a transferência de informação entre clientes). Os componentes-clientes executam processos independentemente.

Arquitetura de fluxo de dados. Essa arquitetura é aplicada quando dados de entrada devem ser transformados, por meio de uma série de componentes computacionais ou manipulativos, em dados de saída. Uma estrutura tubo e filtro (veja a Figura 10.2) tem um conjunto de componentes, chamados de *filtros*, conectados por *tubos* que transmitem dados de um componente para o próximo. Cada filtro trabalha independentemente dos componentes afluentes e efluentes, é projetado para esperar entrada de dados de uma certa forma e produzir dados de saída (para o filtro seguinte) de um modo específico. No entanto, o filtro não exige conhecimento do trabalho dos filtros vizinhos.

FIGURA 10.1

Arquitetura centrada nos dados

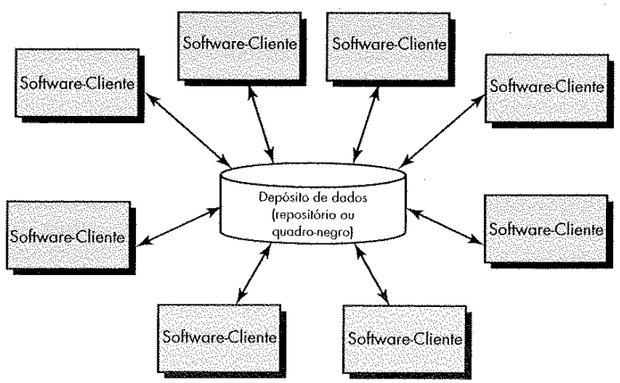
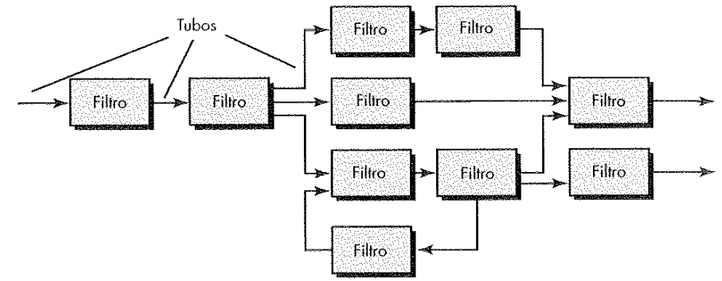


FIGURA 10.2

Arquitetura de fluxo de dados



> "O uso de estilos e padrões de projeto é comum em disciplinas de engenharia."
>
> Mary Shaw e David Garlan

Se o fluxo de dados se degenera em uma única linha de transformações, é denominado *seqüencial por lotes*. Essa estrutura aceita um lote de dados e então aplica uma série de componentes seqüenciais (filtros) para transformá-lo.

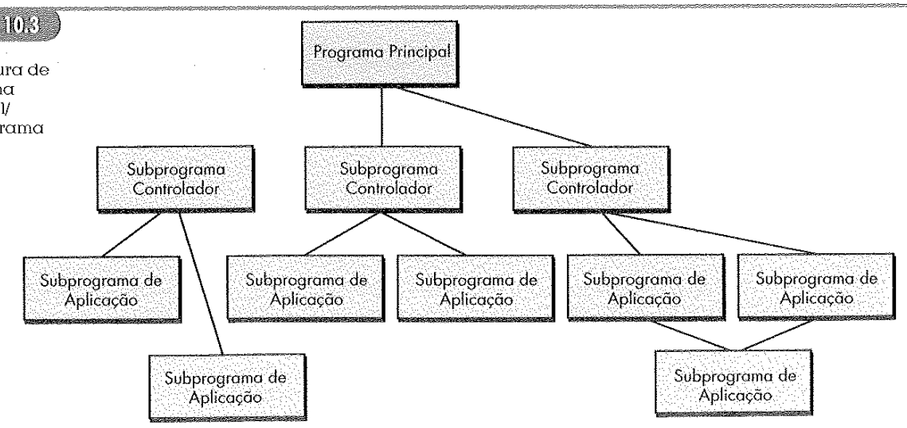
Arquitetura de chamada e retorno. Esse estilo arquitetural permite ao projetista de software (arquiteto de sistema) conseguir uma estrutura de programa relativamente fácil de modificar e ampliar. Dois subestilos [BAS03] existem nessa categoria:

- *Arquiteturas de programa principal/subprograma.* Essa estrutura clássica de programa decompõe a função em uma hierarquia de controle em que um programa "principal" aciona um certo número de componentes de programa, que, por sua vez, invocam outros componentes. A Figura 10.3 mostra uma arquitetura desse tipo.
- *Arquiteturas de chamada de procedimentos remotos.* Os componentes de uma arquitetura de programa principal/subprograma são distribuídos entre vários computadores em uma rede.

Arquitetura orientada a objetos. Os componentes de um sistema encapsulam os dados e as operações que devem ser aplicadas para manipular os dados. A comunicação e a coordenação entre componentes são obtidas por meio de passagem de mensagens.

FIGURA 10.3

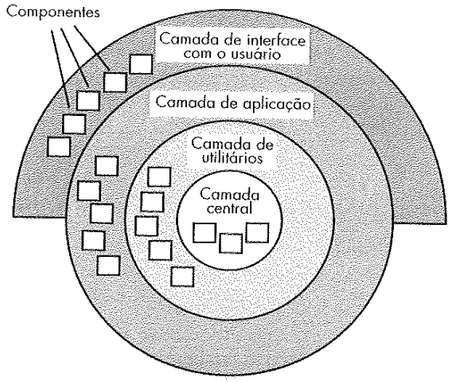
Arquitetura de programa principal/subprograma



Document shared on www.docsity.com

FIGURA 10.4

Arquitetura em camadas



Arquitetura em camadas. A estrutura básica de uma arquitetura em camadas é ilustrada na Figura 10.4. Um certo número de camadas diferentes é definido, cada uma realizando operações que se tornam progressivamente mais próximas do conjunto de instruções de máquina. Na camada exterior, os componentes servem as operações da interface com o usuário. Na camada mais interna os componentes realizam a interface com o sistema operacional. As camadas intermediárias fornecem serviços utilitários e funções do software de aplicação.

Esses estilos arquiteturais são apenas um pequeno subconjunto dos que estão disponíveis para o projetista de software.[2] Depois que a engenharia de requisitos descobre as características e restrições do sistema a ser construído, o estilo arquitetural ou combinação de estilos, que melhor se encaixa nessas características e restrições, pode ser escolhido. Em muitos casos, mais de um estilo pode ser adequado e alternativas poderiam ser projetadas e avaliadas. Por exemplo, um estilo em camadas (adequado à maioria dos sistemas) pode ser combinado com uma arquitetura centrada nos dados em muitas aplicações de banco de dados.

CASASEGURA

Escolha de um Estilo Arquitetural

A cena: Sala de Jamie, à medida que a modelagem de projeto continua.

Os personagens: Jamie e Ed — membros da equipe de engenharia de software do *CasaSegura*.

A conversa:

Ed (franzindo a testa): Nós estamos modelando a função de segurança usando UML... você sabe, classes, relacionamentos, essa espécie de coisa. Assim, acho que a arquitetura[3] orientada a objetos é o caminho certo a percorrer.

Jamie: Mas...?

Ed: Mas... tenho problemas em visualizar o que uma arquitetura orientada a objetos é. Eu entendo uma arquitetura de chamada e retorno, parece uma hierarquia de processo convencional, mas OO... não sei. Parece uma coisa amorfa.

Jamie (sorrindo): Amorfa, hein?

Ed: Sim... o que eu quero dizer é que não consigo visualizar uma estrutura real, apenas classes de projeto flutuando no espaço.

Jamie: Bem, isso não é verdade. Há hierarquias de classes... pense na hierarquia (agregação) que fizemos para o objeto **PlantaBaixa** [veja a Figura 9.3]. Uma arquitetura OO é uma combinação daquela arquitetura e das interconexões — você sabe, colaborações entre as classes. Podemos mostrá-la descrevendo completamente os atributos e operações, a passagem de mensagens e a estrutura das classes.

Ed: Eu vou gastar uma hora mapeando uma arquitetura de chamada e retorno, depois vou voltar e considerar uma arquitetura OO.

Jamie: Doug não terá problema com isso. Ele disse que deveríamos considerar alternativas arquiteturais. Por falar nisso, não há absolutamente razão para que ambas as arquiteturas não possam ser usadas em combinação umas com as outras.

Ed: Bom. Eu concordo.

2 Veja [BOS00], [HOF00], [BAS03], [SHA97], [BUS96] e [SHA96] para discussão detalhada de estilos e padrões arquiteturais.

3 Pode ser argumentado que a arquitetura do *CasaSegura* deveria ser considerada em mais alto nível do que a arquitetura anotada. *CasaSegura* tem uma variedade de subsistemas — funcionalidade de monitoramento da casa, o site Web da companhia de monitoramento e o subsistema que roda no PC do proprietário. Nos subsistemas, processos concorrentes (por exemplo, aqueles que monitoram sensores) e manipulação de eventos são prioritários. Algumas decisões arquiteturais nesse nível são feitas durante a engenharia de sistemas ou produto (veja o Capítulo 6), mas o projeto arquitetural na engenharia de software deve considerar muito bem todos esses tópicos.

10.3.2. Padrões Arquiteturais

Se um construtor de casa decide construir um colonial americano com *hall* central, há um único estilo arquitetural que pode ser aplicado. Os detalhes do estilo (número de lareiras, fachada da casa, disposição das portas e janelas) podem variar consideravelmente, mas, uma vez tomada a decisão sobre a arquitetura global da casa, o estilo é imposto ao projeto.[4]

Padrões arquiteturais são um pouco diferente.[5] Por exemplo, toda casa (e todo estilo arquitetural para casa) utiliza o padrão *cozinha*. O padrão *cozinha* define a necessidade da disposição dos eletrodomésticos básicos de uma cozinha, a necessidade de uma pia, a necessidade de armários e, possivelmente, regras para a colocação dessas coisas relativamente ao fluxo de trabalho na cozinha. Além disso, o padrão pode especificar a necessidade de tampas de balcões, iluminação, interruptores de parede, uma ilha central, pavimentação etc. Obviamente, há mais do que um único projeto para uma cozinha, mas todo projeto pode ser concebido no contexto da "solução" sugerida pelo padrão *cozinha*.

Como já mencionamos, padrões arquiteturais para software definem uma abordagem específica para tratar de algumas características comportamentais do sistema. Bosch [BOS00] define um certo número de domínios de padrões arquiteturais. Exemplos representativos são fornecidos nos parágrafos que se seguem.

PONTO CHAVE

Uma arquitetura de software pode ter um certo número de padrões arquiteturais que trate de tópicos como concorrência, persistência e distribuição.

Concorrência. Muitas aplicações precisam tratar de múltiplas tarefas de um modo que simule paralelismo (isso ocorre sempre que múltiplas tarefas ou componentes "paralelos" são gerenciados por um único processador). Há um certo número de diferentes modos pelos quais uma aplicação pode tratar da concorrência, e cada um pode ser apresentado por um diferente padrão arquitetural. Por exemplo, uma abordagem deve usar um padrão de *gestão de processo do sistema operacional* que fornece características incorporadas ao SO, permitindo que os componentes sejam executados concorrentemente. O padrão também incorpora funcionalidade do SO que gerencia a comunicação entre processos, escalonamento e outras capacidades necessárias para atingir concorrência. Outra abordagem poderia definir um escalonador de tarefas para a aplicação. Um padrão *escalonador de tarefa* contém um conjunto de objetos ativos em que cada um contém uma operação *ticar()* [BOS00]. O escalonador periodicamente invoca *ticar()* para cada objeto, que então executa as funções a realizar antes de devolver o controle para o escalonador, que então chama a operação *ticar()* para o próximo objeto concorrente.

Persistência. Dados persistem se sobrevivem depois da execução do processo que os criou. Dados persistentes são armazenados em um banco de dados ou arquivo e podem ser lidos ou modificados por outros processos posteriormente. Em ambientes orientados a objetos, a idéia

4 Isso implica que existirá um vestíbulo central e um corredor de entrada, que as salas serão colocadas à esquerda e à direita do vestíbulo, que a casa terá dois (ou mais) andares, que os quartos serão no andar superior etc. Essas "regras" são impostas uma vez tomada a decisão de usar o estilo colonial americano com *hall* central.

5 É importante notar que não há concordância universal sobre essa terminologia. Algumas pessoas (por exemplo, [BUS96]) usam os termos *estilos* e *padrões* como sinônimos, enquanto outras fazem a distinção sutil sugerida nesta seção.

Document shared on www.docsity.com

de objeto persistente estende o conceito de persistência um pouco mais. Os valores de todos os atributos do objeto, o estado geral do objeto, e outra informação suplementar são armazenados para uso e recuperação futuros. Em geral, dois padrões arquiteturais são utilizados para conseguir persistência — um padrão *sistema de gestão de banco de dados* que aplica a capacidade de armazenamento e recuperação de um SGBD (DBMS, em inglês) à arquitetura da aplicação ou um padrão de *persistência da aplicação* que constrói características de persistência em uma arquitetura de aplicação (por exemplo, software para processamento de texto que gerencia sua própria estrutura de documento).

Distribuição. O problema de distribuição trata da maneira pela qual sistemas ou componentes de sistemas comunicam-se entre si em um ambiente distribuído. Há dois elementos para esse problema: (1) o modo pelo qual as entidades se conectam entre si, e (2) a natureza da comunicação que ocorre. O padrão arquitetural mais comum estabelecido para tratar do problema de distribuição é o padrão *broker* (corretor). O *broker* atua como um "intermediário" entre o componente-cliente e um componente-servidor. O cliente envia uma mensagem para o *broker* (contendo todas as informações apropriadas para que a comunicação seja efetuada), e o broker completa a conexão. CORBA (veja o Capítulo 30) é um exemplo de uma arquitetura *broker*.

Antes que qualquer um dos padrões arquiteturais mencionados possa ser escolhido, ele deve ser avaliado quanto à sua adequação à aplicação e ao estilo arquitetural global bem como quanto à sua manutenibilidade, confiabilidade, segurança e desempenho.

10.3.3 Organização e Refinamento

Como o processo de projeto freqüentemente oferece ao engenheiro de software várias alternativas arquiteturais, é importante estabelecer um conjunto de critérios de projeto que pode ser usado para avaliar um projeto arquitetural derivado. As questões a seguir [BAS03] propiciam uma idéia mais precisa do estilo arquitetural disponível.

? Como devo avaliar um estilo arquitetural derivado?

Controle. Como o controle é gerido na arquitetura? Existe uma hierarquia de controle separada e, se existir, qual o papel dos componentes nessa hierarquia de controle? Como os componentes transferem controle do sistema? Como o controle é compartilhado entre os componentes? Qual a topologia de controle (a forma geométrica que o controle toma)? O controle é sincronizado ou os componentes operam de maneira assíncrona?

Dados. Como os dados são passados entre os componentes? O fluxo de dados é contínuo ou os objetos de dados são passados esporadicamente para o sistema? Qual é o modo de transferência de dados (os dados são passados de um componente para outro ou estão disponíveis globalmente para serem compartilhados entre os componentes do sistema)? Existem componentes de dados (por exemplo, quadro-negro ou repositório) e, se existirem, qual é o seu papel? Como os componentes funcionais interagem com os componentes de dados? Os componentes de dados são passivos ou ativos (o componente de dados interage ativamente com outros componentes do sistema)? Como dados e controle interagem dentro do sistema?

Essas questões fornecem ao projetista uma avaliação inicial da qualidade do projeto e lançam a fundação para uma análise mais detalhada da arquitetura.

10.4 Projeto Arquitetural

À medida que um projeto arquitetural começa, o software a ser desenvolvido deve ser colocado no contexto — isto é, o projeto deve definir as entidades externas (outros sistemas, dispositivos, pessoas) com as quais o software interage e a natureza da interação. Essa informação pode ser, geralmente, obtida do modelo de análise e todas as outras informações coletadas durante a engenharia de requisitos. Uma vez que o contexto é modelado e todas as interfaces externas do software tenham sido descritas, o projetista especifica a estrutura do sistema definindo e refinando os componentes de software que implementam a arquitetura. Esse processo continua iterativamente até que uma estrutura arquitetural completa tenha sido derivada.

> "Um médico pode enterrar seus erros, mas um arquiteto pode apenas aconselhar seu cliente a plantar heras."
>
> Frank Lloyd Wright

PONTO CHAVE

O contexto arquitetural representa como o software interage com entidades externas aos seus limites.

10.4.1 Representação do Sistema no Contexto

No Capítulo 6, observamos que um engenheiro de sistema precisa modelar o contexto. Um diagrama de contexto do sistema (veja a Figura 6.4) satisfaz a esse requisito representando o fluxo de informação para dentro e para fora do sistema, a interface do usuário e o apoio de processamento relevante. No projeto arquitetural, um arquiteto de software usa um diagrama de contexto arquitetural (*architectural context diagram*, ACD) para modelar a maneira pela qual o software interage com entidades externas aos seus limites. A estrutura genérica do diagrama de contexto arquitetural é mostrada na Figura 10.5.

? Como os sistemas interoperam uns com os outros?

Referindo-se à figura, sistemas que interoperam com o *sistema-alvo* (o sistema para o qual o projeto arquitetural deve ser desenvolvido) são representados por:

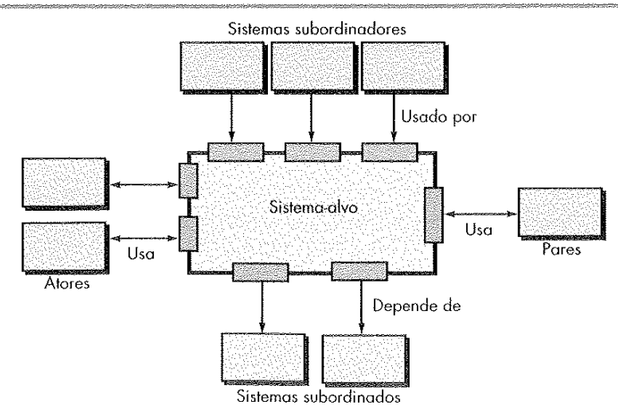
- *Sistemas subordinadores* — aqueles sistemas que usam o sistema-alvo como parte de algum esquema de processamento de mais alto nível.
- *Sistemas subordinados* — aqueles sistemas que são usados pelo sistema-alvo e fornecem dados ou processamento necessários para completar a funcionalidade do sistema-alvo.
- *Sistemas no nível de pares* — aqueles sistemas que interagem em base par-a-par (a informação é produzida ou consumida entre os pares e o sistema-alvo).
- *Atores* — aquelas entidades (pessoas, dispositivos) que interagem com o sistema-alvo produzindo ou consumindo informação necessária para o requisito de processamento.

Cada uma dessas entidades externas comunica-se com o sistema-alvo por meio de uma interface (os retângulos menores sombreados).

Para ilustrarmos o uso do ACD, consideremos novamente a função de segurança residencial do produto *CasaSegura*. O produto controlador global de *CasaSegura* e o sistema baseado na Internet são ambos subordinadores da função de segurança e são mostrados acima da função na Figura 10.6. A função de vigilância é um *sistema par* e usa (é usada por) a função de segurança residencial em funções posteriores do produto. O proprietário e o painel de controle são atores que são produtores e consumidores de informação usada/produzida pelo software de segurança. Por fim, os sensores são usados pelo software de segurança e são mostrados como subordinados a ele.

FIGURA 10.5

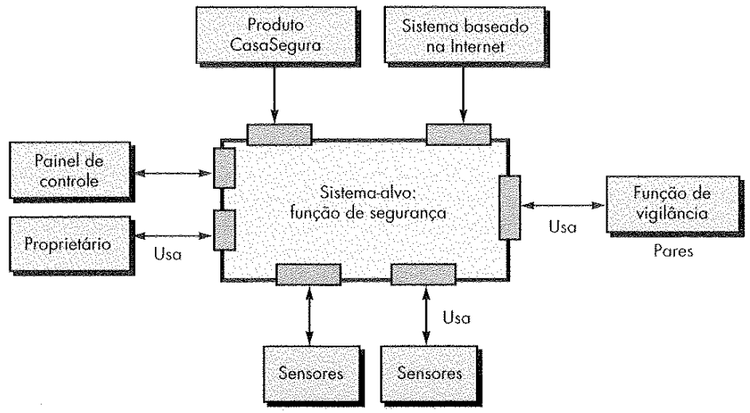
Diagrama de contexto arquitetural (adaptado de [BOS00])



Document shared on www.docsity.com

FIGURA 10.6

Diagrama de contexto arquitetural para a função de segurança de *CasaSegura*



Como parte do projeto arquitetural, os detalhes de cada interface apresentada na Figura10.6 deveriam ser especificados. Todos os dados que fluem para dentro e para fora do sistema-alvo devem ser identificados neste estágio.

10.4.2 Definição de Arquétipos

PONTO CHAVE

Arquétipos são blocos construtivos abstratos de um projeto arquitetural.

Um *arquétipo* é uma classe ou padrão que representa uma abstração central crítica para o projeto de uma arquitetura para o sistema-alvo. Em geral, é necessário um conjunto relativamente pequeno de arquétipos para projetar mesmo sistemas relativamente complexos. A arquitetura do sistema-alvo é composta desses arquétipos, que representam elementos estáveis da arquitetura, mas podem ser instanciados de muitos modos diferentes com base no comportamento do sistema.

Em muitos casos, arquétipos podem ser derivados pelo exame das classes de análise definidas como parte do modelo de análise. Continuando nossa discussão da função de segurança do *CasaSegura*, podemos definir os seguintes arquétipos:

- **Nós.** Representam uma coleção coesiva de elementos de entrada e saída da função de segurança. Por exemplo, um nó pode ser composto de (1) vários sensores, e (2) uma variedade de indicadores de alarme (saída).
- **Detectores.** Uma abstração que engloba todos os equipamentos de sensoriamento que alimentam informação no sistema-alvo.
- **Indicadores.** Uma abstração que representa todos os mecanismos (sirene do alarme, luzes piscantes, campainha) para indicar que uma condição de alarme está ocorrendo.
- **Controlador.** Uma abstração que representa o mecanismo que permite armar ou desarmar um nó. Se os controladores residem em uma rede, eles têm a habilidade de se comunicar uns com os outros.

Cada um desses arquétipos é representado por meio da notação UML, como mostra a Figura 10.7.

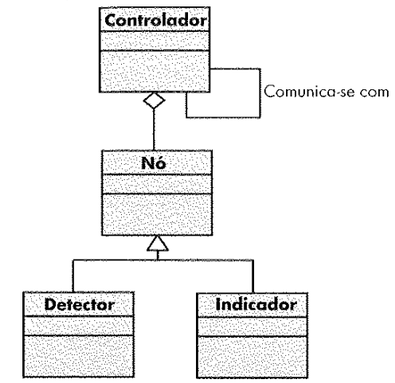
Vale lembrar que os arquétipos formam a base da arquitetura, mas são abstrações que precisam ser mais refinadas à medida que o projeto arquitetural prossegue. Por exemplo, **Detector** pode ser refinado em uma hierarquia de classes de sensores.

10.4.3 Refinar a Arquitetura em Componentes

À medida que a arquitetura de software é refinada em componentes, a estrutura do sistema começa a emergir. Contudo, como são escolhidos esses componentes? A fim de responder a essa questão, o

FIGURA 10.7

Relacionamentos UML para os arquétipos da função de segurança do *CasaSegura* (adaptado de [BOS00])



AVISO

Componentes da arquitetura de software são derivados de três fontes — o domínio da aplicação, o domínio de infra-estrutura e o domínio de interface. Como a modelagem de análise não trata de infra-estrutura, preveja suficiente tempo de projeto para considerá-lo cuidadosamente.

projetista arquitetural começa com as classes que foram descritas como parte do modelo de análise[6]. Essas classes de análise representam entidades do domínio de aplicação (negócio) que devem ser atendidas pela arquitetura de software. Assim, o domínio de aplicação é uma fonte para a derivação e refinamento de componentes. Uma outra fonte é o domínio de infra-estrutura. A arquitetura deve acomodar muitos componentes de infra-estrutura que habilitam componentes da aplicação, mas não têm conexão de negócio com o domínio de aplicação. Por exemplo, componentes de gestão de memória, componentes de comunicação, componentes de banco de dados e componentes de gestão de tarefas são freqüentemente integrados na arquitetura de software.

As interfaces representadas no diagrama de contexto da arquitetura (veja a Seção 10.4.1) implicam um ou mais componentes especializados que processam os dados que fluem pela interface. Em alguns casos (por exemplo, interface gráfica com o usuário), deve ser projetado um subsistema completo da arquitetura com muitos componentes.

> "A estrutura de um sistema de software fornece a ecologia na qual o código nasce, amadurece e morre. Um hábitat bem projetado permite evolução com sucesso de todos os componentes necessários a um sistema de software."
>
> R. Pattis

Continuando o exemplo da função de segurança do *CasaSegura*, podemos definir o conjunto de componentes de mais alto nível que atende à seguinte funcionalidade:

- *Gestão de comunicação externa* — coordena a comunicação da função de segurança com entidades externas, por exemplo, sistemas baseados em Internet, notificação externa de alarme.
- *Processamento do painel de controle* — gerencia toda a funcionalidade do painel de controle.
- *Gestão de detecção* — coordena o acesso para todos os detectores conectados ao sistema.
- *Processamento de alarme* — verifica e atua sobre todas as condições de alarme.

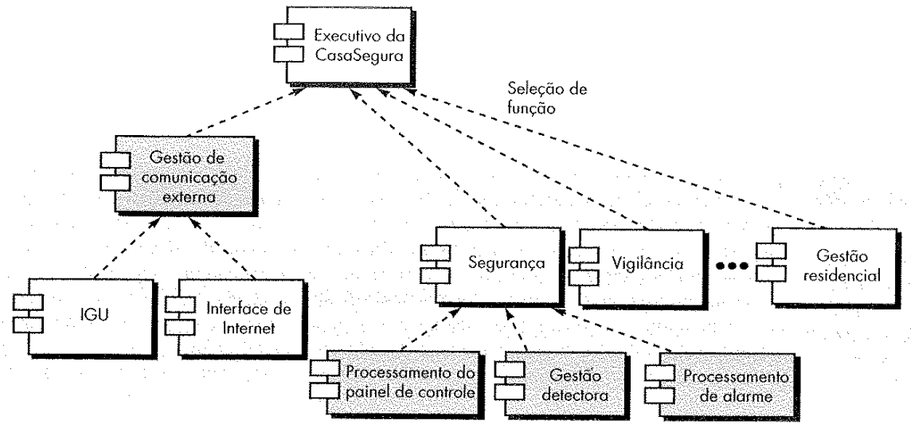
Cada um desses componentes de mais alto nível deveria ser elaborado iterativamente e então posicionado na arquitetura global do *CasaSegura*. Classes de projeto (com atributos e operações adequados) seriam definidas para cada um. É importante notar, no entanto, que os detalhes de projeto de todos os atributos e operações não seriam especificados até o projeto em nível de componentes (veja o Capítulo 11).

6 Se uma abordagem convencional (não orientada a objetos) for escolhida, os componentes podem ser derivados do modelo de fluxo de dados. Discutiremos essa abordagem na Seção 10.6.

Document shared on www.docsity.com

A estrutura global da arquitetura (representada como um diagrama de componente UML) é ilustrada na Figura 10.8. Transações são adquiridas pelo *Gestão de comunicação externa* à medida que elas entram pelos componentes que processam a *Interface Gráfica do Usuário do CasaSegura* e a *Interface de Internet*. Essa informação é gerenciada por um componente *executivo do CasaSegura* que seleciona a função produto adequada (nesse caso, segurança). O componente de *processamento do painel de controle* interage com o proprietário para armar/desarmar a função de segurança. O componente *gestão detectora* percorre os sensores para detectar uma condição de alarme, e o componente *processamento de alarme* produz saída quando um alarme é detectado.

FIGURA 10.8 Estrutura arquitetural global para *CasaSegura* com componentes de mais alto nível



10.4.4 Descrição das Instanciações do Sistema

O projeto arquitetural que tem sido modelado até este ponto é ainda relativamente de alto nível. O contexto do sistema foi representado; arquétipos que indicam as abstrações importantes no domínio do problema foram definidos; a estrutura global do sistema está evidente; e os principais componentes de software foram identificados. Entretanto, é ainda necessário mais refinamento (lembrando que todo o projeto é iterativo).

FERRAMENTAS DE SOFTWARE

Projeto Arquitetural

Objetivo: Ferramentas de projeto arquitetural modelam a estrutura de software pela representação dos componentes de interface, dependências e relacionamentos, e interações.

Mecânica: A mecânica da ferramenta varia. Na maioria dos casos, a capacidade do projeto arquitetural é parte da funcionalidade fornecida por ferramentas automatizadas para modelagem de análise e de projeto.

Ferramentas Representativas[7]

Adalon, desenvolvida por Synthis Corp. (www.synthis.com), é uma ferramenta de projeto especializada para projeto e construção de arquiteturas específicas de componentes baseados na Web.

ObjectiF, desenvolvida por microTOOL GmbH (www.microtool.com), é uma ferramenta de projeto baseada na UML que leva a arquiteturas (por exemplo, Coldfusion, J2EE, Fusebox) de uso fácil para engenharia de software baseada em componentes (veja o Capítulo 30).

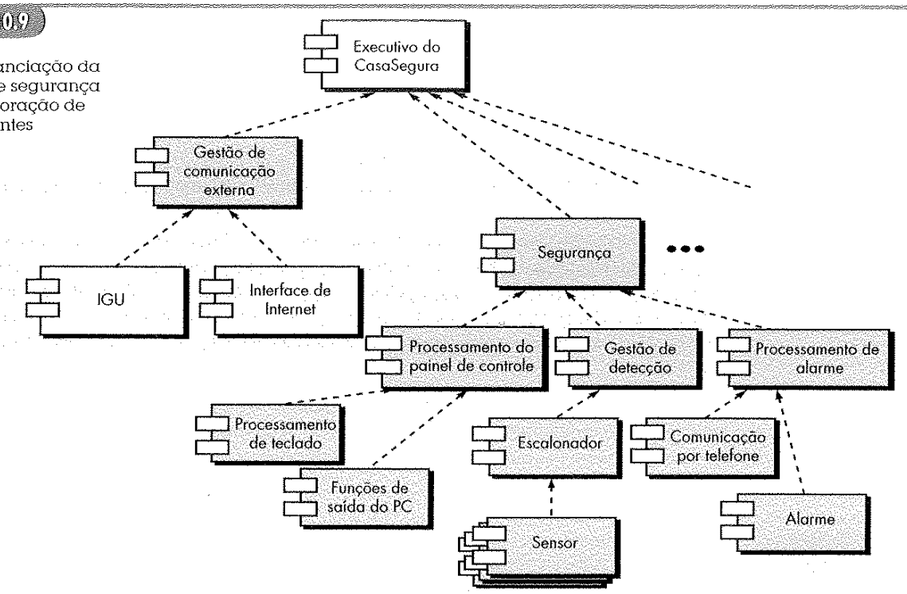
Rational Rose, desenvolvida por Rational (www.rational.com), é uma ferramenta de projeto baseada na UML que suporta todos os aspectos do projeto arquitetural.

7 Os produtos mencionados aqui não representam uma recomendação, mas em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

Para tanto, uma *instanciação* real da arquitetura é desenvolvida. Com isso, queremos dizer que a arquitetura é aplicada a um problema específico com o objetivo de demonstrar que a estrutura e os componentes são adequados.

A Figura 10.9 mostra uma instanciação da arquitetura do *CasaSegura* para o sistema de segurança. Componentes da Figura 10.8 são melhor refinados para mostrar detalhe adicional. Por exemplo, o componente *gestão de detecção* interage com um componente de infra-estrutura *escalonador* que implementa o percurso "concorrente" de cada objeto *sensor* usado pelo sistema de segurança. Elaboração análoga é realizada para cada um dos componentes representados na Figura 10.8.

FIGURA 10.9 Uma instanciação da função de segurança com elaboração de componentes



10.5 AVALIAÇÃO DE ALTERNATIVAS DE PROJETO ARQUITETURAL

Na melhor das hipóteses, o projeto resulta em um número de alternativas arquiteturais em que cada uma é avaliada para determinar qual a mais adequada para o problema a ser resolvido. Nas seções a seguir, consideramos a avaliação de projetos arquiteturais alternativos.

> "Talvez estejo no porão. Deixe-me subir e verificar."
>
> **M. C. Escher**

10.5.1 Um Método de Análise dos Compromissos da Arquitetura

O Software Engineering Institute (SEI) desenvolveu um *método de análise de compromissos da arquitetura* (*architecture trade-off analysis method*, ATAM) [KAZ98] que estabelece um processo iterativo de avaliação de arquiteturas de software. As seguintes atividades de análise de projeto são realizadas iterativamente:

Document shared on www.docsity.com

Veja na Web

Informações mais aprofundadas sobre ATAM podem ser obtidas em www.sei.cmu.edu/ata/ata_method.html.

1. *Reúna cenários.* Um conjunto de casos de uso (veja os Capítulos 7 e 8) é desenvolvido para representar o sistema sob o ponto de vista do usuário.
2. *Elicite requisitos, restrições e descrição do ambiente.* Essa informação é necessária como parte da engenharia de requisitos e é usada para garantir que todas as preocupações do interessado foram consideradas.
3. *Descreva os padrões/estilos arquiteturais que foram escolhidos para cuidar dos cenários e dos requisitos.*
4. *Avalie os atributos de qualidade considerando cada um isoladamente.* Atributos de qualidade para avaliação do projeto arquitetural incluem confiabilidade, desempenho, segurança, manutenibilidade, flexibilidade, testabilidade, portabilidade, reusabilidade e interoperabilidade.
5. *Identifique a sensibilidade de atributos de qualidade aos diversos atributos arquiteturais para um estilo arquitetural específico.* Isso pode ser conseguido fazendo pequenas modificações na arquitetura e determinando quão sensível um atributo de qualidade, por exemplo, desempenho, é à modificação. Quaisquer atributos que são significativamente afetados pela variação na arquitetura são chamados de *pontos sensíveis.*
6. *Critique as arquiteturas candidatas (desenvolvidas no passo 3) usando a análise de sensibilidade conduzida no passo 5.* O SEI descreve essa abordagem do seguinte modo [KAZ98]:

> Uma vez determinados os pontos arquiteturais sensíveis, encontrar os pontos de compromisso é simplesmente identificar os elementos arquiteturais para os quais múltiplos atributos são sensíveis. Por exemplo, o desempenho de uma arquitetura cliente-servidor pode ser altamente sensível ao número de servidores (o desempenho melhora, dentro de algum intervalo, aumentando o número de servidores). O número de servidores, então, é um ponto de compromisso relativamente a essa arquitetura.

Esses seis passos representam a primeira iteração ATAM. Baseando-se nos resultados dos passos 5 e 6, algumas arquiteturas alternativas podem ser eliminadas, uma ou mais das arquiteturas remanescentes podem ser modificada e representada em mais detalhes e, então, os passos ATAM são reaplicados.[8]

CASASEGURA

Avaliação da Arquitetura

A cena: Escritório de Doug Miller à medida que a modelagem do projeto arquitetural prossegue.

Os personagens: Vinod, Jamie, Shakira e Ed — membros da equipe de engenharia de software de *CasaSegura*. Também Doug Miller, gerente do grupo de engenharia de software.

A conversa:

Doug: Eu sei que vocês estão derivando um par de diferentes arquiteturas para o produto *CasaSegura* e essa é uma boa coisa. Acho que minha questão é, como vamos escolher a que é melhor?

Ed: Estou trabalhando em um estilo de chamada e retorno, assim Jamie ou eu vamos derivar uma arquitetura OO.

Doug: Está certo, e como nós vamos escolher?

Shakira: Eu fiz um curso de projeto em meu último ano de faculdade, e me lembro que há uma porção de modos para fazê-lo.

Vinod: Há, mas eles são um pouco acadêmicos. Veja, acho que podemos fazer a nossa avaliação e escolher o adequado usando casos de uso e cenários.

Doug: Não é a mesma coisa?

Vinod: Não, quando você estiver falando sobre avaliação arquitetural. Nós já temos um conjunto completo de casos de uso. Assim, aplicamos cada um a ambas as arquiteturas e vemos como o sistema reage — como os componentes e conectores funcionam no contexto dos casos de uso.

Ed: Essa é uma boa idéia. Garante que não deixemos nada de fora.

8 O *Método de Análise de Arquitetura de Software (Software Architecture Analysis Method, SAAM)* é uma alternativa para o ATAM e pode valer a pena ser examinado pelos leitores interessados em análise arquitetural. Um artigo sobre SAAM pode ser encontrado em http://www.sei.cmu.edu/publications/articles/saam-method-propert-sas.html.

Vinod: Certo, mas também nos diz se o projeto arquitetural é enrolado, se o sistema tem que se contorcer para realizar o serviço.

Jamie: Cenários não é justamente um outro nome para casos de uso?

Vinod: Não, nesse caso um cenário implica alguma coisa diferente.

Doug: Você está falando sobre um cenário de qualidade ou um cenário de modificação, certo?

Vinod: Sim. O que nós fazemos é voltar até os interessados e perguntar-lhes como é provável que o *CasaSegura* se modifique no próximos, digamos, três anos. Você sabe, novas versões, características, essa espécie de coisa. Nós construímos um conjunto de cenários de modificação. Também desenvolvemos um conjunto de cenários de qualidade que definem os atributos que gostaríamos de ver na arquitetura do software.

Jamie: E os aplicamos às alternativas.

Vinod: Exatamente. O estilo que trata melhor os casos de uso e cenários é o que nós escolhemos.

10.5.2 Complexidade Arquitetural

Uma técnica útil para avaliar a complexidade global da arquitetura proposta é considerar dependências entre componentes dentro da arquitetura. Essas dependências são guiadas pelo fluxo de informação/controle dentro do sistema. Zhao [ZHA98] sugere três tipos de dependências:

> *Dependências de compartilhamento* representam relações de dependência entre consumidores que usam o mesmo recurso ou produtores que produzem para os mesmos consumidores. Por exemplo, para dois componentes *u* e *v*, se *u* e *v* se referem aos mesmos dados globais, então existe uma relação de dependência de compartilhamento entre *u* e *v*.
>
> *Dependências de fluxo* representam relações de dependência entre produtores e consumidores de recursos. Por exemplo, para dois componentes *u* e *v*, se *u* deve ser completado antes que o controle flua para *v* (pré-requisito), ou se *u* se comunica com *v* por parâmetros, então existe uma relação de dependência de fluxo entre *u* e *v*.
>
> *Dependências de restrição* representam restrições no fluxo relativo de controle entre um conjunto de atividades. Por exemplo, para dois componentes *u* e *v*, *u* e *v* não podem ser executados ao mesmo tempo (exclusão mútua), então existe uma relação de dependência de restrição entre *u* e *v*.

As dependências de compartilhamento de fluxo mencionadas por Zhao são semelhantes em alguns modos ao conceito de acoplamento discutido no Capítulo 9. Acoplamento é um importante conceito de projeto que é aplicável à arquitetura e ao componente. Métricas simples para avaliar acoplamento serão discutidas no Capítulo 15.

10.5.3 Linguagens de Descrição Arquitetural

A arquitetura de uma casa tem um conjunto de ferramentas e notações padronizadas que permitem que o projeto seja representado de um modo não ambíguo e compreensível. Apesar da arquitetura do software poder se apoiar na notação UML, em outras formas diagramáticas e algumas ferramentas relacionadas há necessidade de uma abordagem mais formal para a especificação de um projeto arquitetural.

Linguagem de descrição arquitetural (*architectural description language*, ADL) fornece uma sintaxe e semântica para descrever uma arquitetura de software. Hofmann e seus colegas [HOF01] sugerem que uma ADL deve fornecer ao projetista a habilidade de decompor componentes arquiteturais, compor componentes individuais em blocos arquiteturais maiores e representar as interfaces (mecanismos de conexão) entre componentes. Uma vez estabelecidas técnicas descritivas baseadas em linguagem para projeto arquitetural, é mais provável que métodos mais efetivos de avaliação para arquiteturas sejam estabelecidos à medida que o projeto evolui.

Document shared on www.docsity.com

FERRAMENTAS DE SOFTWARE

Linguagens de Descrição Arquitetural

O seguinte resumo de um número de ADLs importantes foi preparado por Rickard Land [LAN02] e é reproduzido com a permissão do autor. Ressalta-se que as cinco primeiras ADLs listadas foram desenvolvidas com o objetivo de pesquisa e não são produtos comerciais.

Rapide (poset.stanford.edu/rapide/) [LUC95] apóia-se na noção de conjuntos parcialmente ordenados.

UniCon (www.cs.cmu.edu/~UniCon) [SHA96] define arquiteturas de software em termos de abstrações que os projetistas acham úteis.

Aesop (www.cs.cmu.edu/~able/aesop/) [GAR94] trata o problema de reuso de estilo.

Wright (www.cs.cmu.edu/~able/wright/) [ALL97] formaliza estilos arquiteturais usando predicados, permitindo assim verificações estáticas para determinar a consistência e completeza de uma arquitetura.

Acme (www.cs.cmu.edu/~acme/) [GAR00] é uma ADL de segunda geração.

UML (www.uml.org/) inclui muitos dos artefatos necessários para descrições arquiteturais, mas não é tão completa quanto outras ADLs.

10.6 Mapear Fluxo de Dados para Uma Arquitetura de Software

Os estilos discutidos na Seção 10.3.1 representam arquiteturas radicalmente diferentes, assim, não deve ser surpresa que não exista uma aplicação abrangente que consiga a transição entre o modelo de análise e diversos estilos arquiteturais. De fato, não há aplicação prática para alguns estilos arquiteturais. O projetista deve abordar a tradução dos requisitos para o projeto nesses estilos usando as técnicas discutidas na Seção 10.4.

Para ilustrarmos uma abordagem de aplicação arquitetural, consideramos a técnica de aplicação para a arquitetura de *chamada e retorno* — uma estrutura extremamente comum para muitos tipos de sistemas. Essa técnica de aplicação permite a um projetista derivar arquiteturas de chamada e retorno razoavelmente complexas de diagramas de fluxo de dados do modelo de análise. A técnica, às vezes denominada *projeto estruturado*, é apresentada em livros por Myers [MYE78] e Yourdon e Constantine [YOU79].

O projeto estruturado é freqüentemente caracterizado como um método de projeto orientado a fluxo de dados, porque fornece uma transição conveniente de um diagrama de fluxo de dados para uma arquitetura de software (veja o Capítulo 8). O tipo de fluxo de informação é o guia da abordagem de transição.

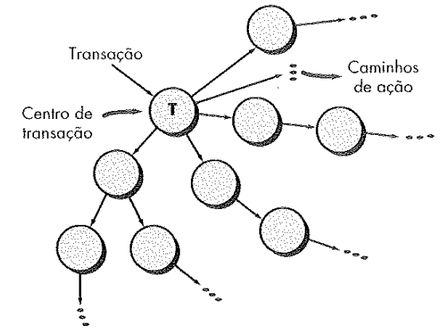
10.6.1 Fluxo de Transformação

A informação deve entrar e sair do software em uma forma de "mundo exterior". Por exemplo, dados digitados em um teclado, tons em uma linha telefônica e imagens de vídeo em uma aplicação multimídia são todas formas de informação do mundo exterior. Tais dados externalizados devem ser convertidos em uma forma interna para processamento. A informação penetra no sistema por caminhos que transformam os dados externos em uma forma interna. Esses caminhos são identificados como *fluxo aferente* (ou fluxo de entrada). No núcleo do software, ocorre uma transição. Os dados que chegam são passados por um *centro de transformação* e começam a se mover por caminhos que agora levam "para fora" do software. Os dados que se movem ao longo desses caminhos são chamados de *fluxo eferente* (ou fluxo de saída). O fluxo global de dados ocorre de um modo seqüencial e segue um caminho, ou apenas alguns caminhos "em linha reta".[9] Quando o segmento de um diagrama de fluxo de dados exibe essas características, um *fluxo de transformação* está presente.

9 Uma aplicação óbvia para esse tipo de fluxo de informação é a arquitetura de fluxo de dados descrita na Seção 10.3.1. Há muitos casos, no entanto, em que a arquitetura de fluxo de dados pode não ser a melhor escolha para um sistema complexo. Exemplos incluem sistemas que vão passar por modificações substanciais ao longo do tempo ou sistemas nos quais o processamento associado com o fluxo de dados não é necessariamente seqüencial.

FIGURA 10.10

Fluxo de transação



10.6.2 Fluxo de Transação

O fluxo de informação é freqüentemente caracterizado por um único item de dados, chamado de *transação*, que dispara outro fluxo de dados ao longo de um ou vários caminhos. Quando um DFD toma a forma mostrada na Figura 10.10, um fluxo de transação está presente.

O fluxo de transação é caracterizado por dados movendo-se ao longo de um caminho eferente que converte informação do mundo exterior em uma transação. A transação é avaliada e, com base no seu valor, é iniciado fluxo ao longo de um dentre vários *caminhos de ação*. O centro retratado no fluxo de informação do qual emanam vários caminhos de ação é chamado de *centro de transações*.

Deve-se notar que, dentro do DFD de um sistema de grande porte, tanto o fluxo de transação quanto o de transformação podem estar presentes. Por exemplo, em um fluxo orientado a transações, o fluxo de informação ao longo de um caminho de ação pode ter características de fluxo de transformação.

10.6.3 Mapeamento de Transformação

Mapeamento de transformação é um conjunto de passos de projeto que permite a um DFD, com características de fluxo de transformação, ser aplicado a um estilo arquitetural específico. Para ilustrarmos essa abordagem, novamente vamos considerar a função de segurança do *CasaSegura*.[10] Um elemento do modelo de análise é um conjunto de diagramas de fluxo de dados que descrevem fluxo de informação dentro de uma função de segurança. Para mapear esses diagramas de fluxo de dados em uma arquitetura, os seguintes passos de projeto são iniciados:

AVISO

Se o DFD nesse instante for mais refinado, procure criar bolhas que exibam alta coesão.

Passo 1. Revise o modelo fundamental do sistema. O modelo fundamental do sistema ou diagrama de contexto representa a função de segurança como uma única transformação, indicando os produtores e consumidores externos de dados que fluem para dentro e para fora da função. A Figura 10.11 mostra o modelo de fluxo de dados no nível 0 e a Figura 10.12 mostra o diagrama de fluxo de dados refinado para a função de segurança.

Passo 2. Revise e refine os diagramas de fluxo de dados para o software. A informação obtida nos modelos de análise é refinada para produzir mais detalhes. Por exemplo, o DFD de nível 2 para *monitore sensores* (Figura 10.13) é examinado e um diagrama de fluxo de dados de nível 3 é derivado, como mostra a Figura 10.14. No nível 3, cada transformação no diagrama de fluxo de dados exibe coesão relativamente alta (veja o Capítulo 9). O processo correspondente a uma transformação realiza uma função única e distinta, que pode ser implementada como um componente no software *CasaSegura*. Assim, o DFD da Figura 10.14 contém detalhes suficientes para uma "primeira aproximação" do projeto de arquitetura para o subsistema *monitore sensores*, e prosseguimos sem refinamento adicional.

10 Consideramos somente a parte da função de segurança do *CasaSegura* que usa o painel de controle. Outras características, discutidas anteriomente neste livro e neste capítulo, não serão consideradas aqui.

Document shared on www.docsity.com

FIGURA 10.11

DFD de nível de contexto para a função de segurança do *CasaSegura*

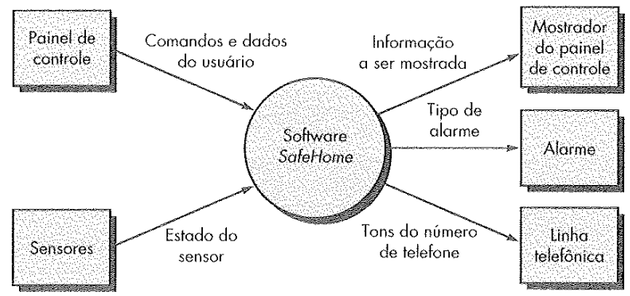


FIGURA 10.12

DFD de nível 1 para a função de segurança do *CasaSegura*

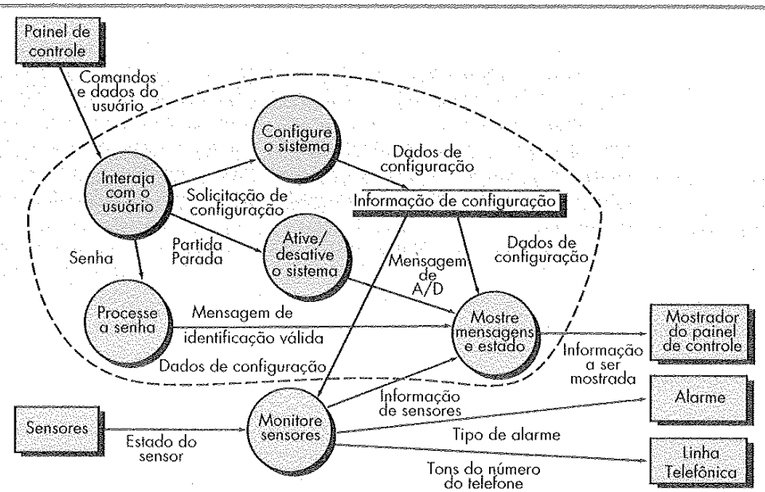


FIGURA 10.13

DFD de nível 2 que refina a transformação *monitore sensores*

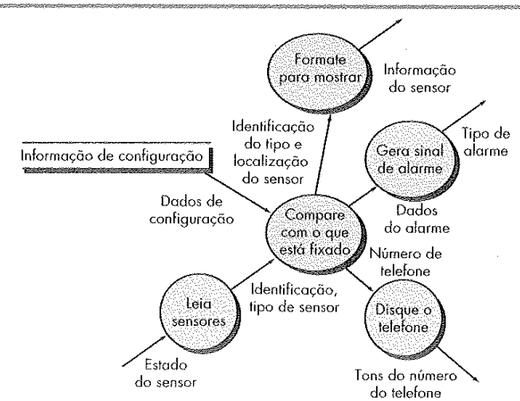
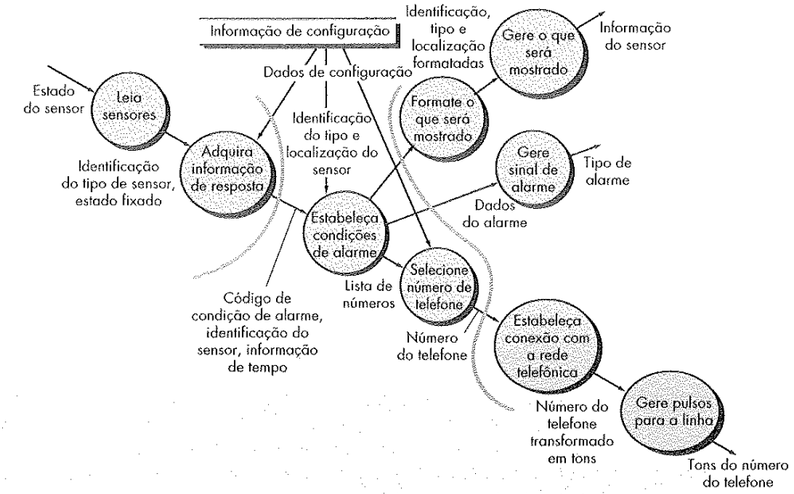


FIGURA 10.14 DFD de nível 3 com separações de fluxo para *monitore sensores*



PONTO CHAVE

Você freqüentemente encontrará ambos os tipos de fluxo de dados dentro do mesmo modelo orientado a fluxo. Os fluxos são particionados e a estrutura do programa é derivada usando a aplicação apropriada.

Passo 3. Determine se o DFD tem características de fluxo de transformação ou de transação. Em geral, o fluxo de informação em um sistema pode sempre ser representado como uma transformação. No entanto, quando uma característica óbvia de transação (veja a Figura 10.10) é encontrada, recomenda-se um mapeamento de projeto diferente. Neste passo, o projetista seleciona características de fluxo global (abrangendo todo o software) baseado na natureza prevalente do DFD. Além disso, regiões locais de fluxo de transformação ou de transação são isoladas. Esses *subfluxos* podem ser usados para refinar a arquitetura do programa, originada da característica global descrita anteriormente. Por enquanto, concentramos nossa atenção apenas no fluxo de dados do subsistema *monitore sensores*, mostrado na Figura 10.14.

Avaliando o DFD (veja a Figura 10.14), vemos dados entrando no software ao longo de um caminho aferente e saindo ao longo de três caminhos eferentes. Nenhum centro de transação distinto está implícito (apesar de a transformação estabelecer condições de alarme que poderiam ser consideradas como tal). Assim sendo, uma característica geral de transformação será considerada para o fluxo de informação.

AVISO

Varie a posição dos separadores de fluxo em um esforço para explorar estruturas alternativas de programa. Isso leva pouquíssimo tempo e pode propiciar conhecimento importante.

Passo 4. Isole o centro de transformação especificando o limite entre o fluxo aferente e o eferente. Na seção anterior, o fluxo aferente foi descrito como um caminho que converte a informação da forma externa para a interna; o fluxo eferente converte da forma interna para a externa. Os limites entre o fluxo aferente e eferente estão abertos à interpretação. Diferentes projetistas podem selecionar pontos ligeiramente diferentes do fluxo como posições do limite. De fato, soluções alternativas de projeto podem ser criadas variando a colocação dos limites de fluxo. Apesar de termos que tomar cuidado com a seleção dos limites, a variação de uma bolha ao longo de um caminho de fluxo geralmente terá pouco impacto na estrutura final do programa.

Limites de fluxo, para o exemplo, são ilustrados como curvas acinzentadas perpendiculares ao fluxo na Figura 10.14. As transformações (bolhas) que constituem o centro de transformação estão entre os dois limites acinzentados, que ocorrem verticalmente na figura. Uma discussão pode ser feita para reajustar um limite (por exemplo, um limite de fluxo aferente separando *leia sensores* e *adquira informação de resposta* poderia ser proposto). A ênfase neste passo de projeto deve ser na seleção de limites razoáveis, em vez de iteração prolongada na colocação dos limites.

Passo 5. Realize "fatoração de primeiro nível". A arquitetura do programa derivada usando essa aplicação resulta em uma distribuição de controle descendente. A *fatoração* resulta em uma estrutura de programa na qual os componenetes de mais alto nível realizam tomada de decisão e

Document shared on www.docsity.com

AVISO

Não se torne dogmático neste estágio. Pode ser necessário estabelecer dois ou mais controladores para o processamento de entrada ou de cálculos, com base na complexidade do sistema a ser construído. Se o bom senso indica essa abordagem, adote-a!

os componentes de baixo nível realizam principalmente trabalho de entrada, cálculos e saída. Componentes de nível intermediário realizam algum controle e fazem quantidades de trabalho moderadas.

Quando é encontrado um fluxo de transformação, um DFD é aplicado em uma estrutura específica (uma arquitetura chamada e retorno) que provê controle para o processamento da informação aferente, de transformação e eferente. Essa fatoração de primeiro nível para o subsistema *monitore sensores* é apresentada na Figura 10.15. Um controlador principal (denominado *executivo de monitore sensores*) fica no topo da estrutura do programa e coordena as seguintes funções de controle subordinadas:

- Um controlador de processamento da informação aferente, chamado de *controlador de entrada do sensor*, coordena o recebimento de todos os dados de entrada.
- Um controlador de transformação de fluxo, chamado de *controlador de condições de alarme*, supervisiona todas as operações nos dados em forma interna (por exemplo, um módulo que aciona vários procedimentos de transformação de dados).
- Um controlador do processamento da informação eferente, chamado de *controlador de saída do alarme*, coordena a produção da informação de saída.

Apesar de uma estrutura em garfo de três dentes estar representada na Figura 10.15, fluxos complexos em sistemas de grande porte podem determinar dois ou mais módulos de controle para cada uma das funções genéricas de controle descritas anteriormente. O número de módulos no primeiro nível deve ser limitado ao mínimo que as funções de controle possam realizar e ainda manter boas características de independência funcional.

AVISO

Mantenha os módulos "trabalhadores" embaixo, na estrutura do programa. Isso vai levar a uma estrutura que é mais fácil de manter.

Passo 6. Realize "fatoração de segundo nível". Fatoração de segundo nível é realizada aplicando-se transformações individuais (bolhas) de um DFD em módulos adequados de uma arquitetura. Começando no limite do centro de transformação e movendo-se para fora ao longo de caminhos aferentes e depois eferentes, as transformações são aplicadas nos níveis subordinados da estrutura de software. A abordagem geral da fatoração de segundo nível para o fluxo de dados é apresentada na Figura 10.16.

Apesar de a Figura 10.16 mostrar uma aplicação um-para-um entre as transformações do DFD e os módulos do software, freqüentemente ocorrem diferentes aplicações. Duas ou até três bolhas podem ser combinadas e representadas como um componente, ou uma única bolha pode ser expandida para

FIGURA 10.15

Fatoração de primeiro nível para *monitore sensores*

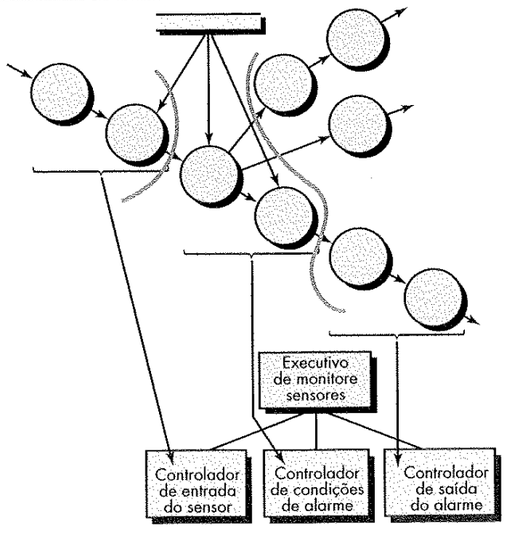
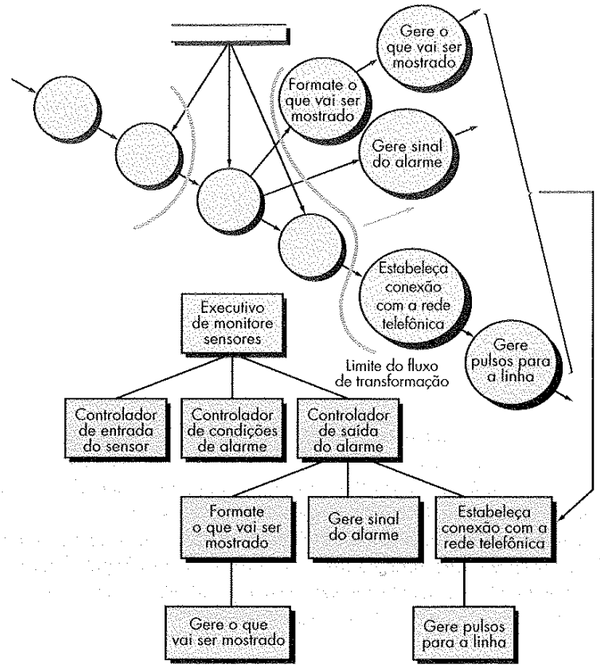


FIGURA 10.16

Fatoração de segundo nível para *monitore sensores*



AVISO

Elimine módulos de controle redundantes. Se um módulo de controle não faz nada a não ser controlar outro módulo, sua função de controle deve ser implodida para um nível mais alto.

dois ou mais componentes. Considerações práticas e medidas da qualidade de projeto determinam o resultado da fatoração de segundo nível. Revisão e refinamento podem levar a modificações nessa estrutura, mas ela pode servir como uma "primeira iteração" de projeto.

A fatoração de segundo nível para fluxo aferente segue do mesmo modo. A fatoração é novamente conseguida movendo-se para fora a partir do limite do centro de transformação ao longo do fluxo aferente. O centro de transformação do subsistema de software *monitore sensores* é aplicado de um modo um tanto diferente. Cada uma das transformações de conversão de dados ou cálculo da porção de transformação do DFD é aplicada em um módulo subordinado ao controlador de transformação. Uma primeira iteração da arquitetura completada é mostrada na Figura 10.17.

Os componentes aplicados da maneira precedente, e mostrados na Figura 10.17, representam um projeto inicial da arquitetura de software. Apesar de os componentes serem denominados de um modo que implica função, uma breve narrativa de processamento (adaptada da PSPEC, criada durante a modelagem de análise) deve ser escrita para cada um.

AVISO

Concentre-se na independência funcional dos módulos que você originou. Alta coesão e baixo acoplamento devem ser sua meta.

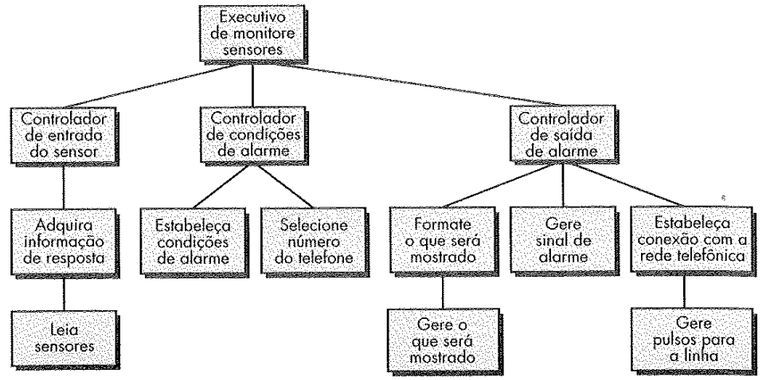
Passo 7. Refine a primeira iteração da arquitetura usando heurísticas de projeto para aperfeiçoar a qualidade do software. Uma primeira iteração de arquitetura pode sempre ser refinada pela aplicação de conceitos de independência funcional (veja o Capítulo 9). Componentes são explodidos ou implodidos para produzir fatoração sensível, boa coesão, mínimo acoplamento e, mais importante, uma estrutura que pode ser implementada sem dificuldade, testada sem confusão e mantida sem mágoa.

Refinamentos são determinados pelos métodos de análise e avaliação descritos brevemente na Seção 10.5, bem como por considerações práticas e bom senso. Há ocasiões, por exemplo, em que o controlador para o fluxo de dados aferente é totalmente desnecessário, quando algum processamento de entrada é exigido em um componente que é subordinado ao controlador de transformação, quando alto acoplamento por causa de dados globais não pode ser evitado ou quando não podem ser obtidas características estruturais ótimas. O árbitro final são requisitos de software associados ao julgamento humano.

Document shared on www.docsity.com

FIGURA 10.17

Estrutura da primeira iteração para *monitore sensores*



O objetivo dos sete passos precedentes é desenvolver uma representação arquitetural do software. Uma vez definida a estrutura, podemos avaliar e refinar a arquitetura do software examinando-o como um todo. Modificações feitas nessa ocasião demandam pouco trabalho adicional, mas podem ter um impacto profundo na qualidade do software.

O leitor deve fazer uma pausa e considerar a diferença entre a abordagem de projeto descrita e o processo de "escrever programas". Se o código é a única representação do software, o desenvolvedor terá grande dificuldade em fazer avaliação ou refinamento global ou holístico e, de fato, terá dificuldade "em enxergar a floresta pelas árvores".

CASASEGURA

Refinamento do Primeiro Esboço da Arquitetura

A cena: Sala de Jamie, à medida que a modelagem de projeto continua.

Os personagens: Jamie e Ed — membros da equipe de engenharia de software do *CasaSegura*.

A conversa:

(Ed acabou de completar o primeiro esboço do projeto do subsistema de monitoramento de sensores. Ele pára para perguntar a Jamie a sua opinião.)

Ed: Aqui está a arquitetura que eu derivei.

(Ed mostra a Jamie a Figura 10.17, que ela estuda durante alguns momentos.)

Jamie: Isso é bom, mas acho que podemos fazer algumas coisas para torná-lo mais simples... e melhor.

Ed: Tais como?

Jamie: Bem, por que você usou o componente *controlador de entrada do sensor*?

Ed: Porque você precisa de um controlador para o mapeamento.

Jamie: Na verdade, não. O controlador não faz muita coisa, pois nós estamos gerindo um único caminho de fluxo para os dados de entrada. Podemos eliminar o controlador sem efeitos desfavoráveis.

Ed: Eu posso viver com isso, vou fazer a modificação e...

Jamie (sorrindo): Espera! Nós podemos também implodir os componentes *estabeleça condições de alarme* e *selecione número de telefone*. O controlador de transformação que você mostra não é realmente necessário, e a pequena diminuição na coesão é tolerável.

Ed: Simplificação, hein?

Jamie: É. E enquanto estamos fazendo refinamentos, seria uma boa idéia implodir os componentes *formate o que será mostrado* e *gere o que será mostrado*. Formatar saída para o painel de controle é simples. Nós podemos definir um novo módulo denominado *produzir saída*.

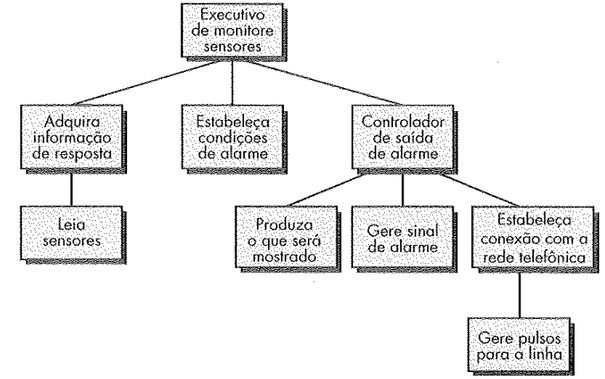
Ed (rascunhando): Então isso é o que você acha que devemos fazer?

(Ele mostra a Jamie a Figura 10.18.)

Jamie: É um começo.

FIGURA 10.18

Estrutura de programa refinada para *monitore sensores*



10.6.4 Mapeamento da Transação

Em muitas aplicações de software, um único item de dados dispara um ou alguns fluxos de informação que efetuam uma função determinada pelo item de dados disparador. O item de dados, chamado de *transação*, e suas correspondentes características de fluxo foram discutidos na Seção 10.6.2. Nesta seção consideraremos os passos de projeto usados para mapear o fluxo de transação na arquitetura de software.

O mapeamento da transação será ilustrado considerando-se o subsistema de interação com o usuário da função de segurança do *CasaSegura*. O fluxo de dados de nível 1 para esse subsistema é mostrado como parte da Figura 10.12. Refinando o fluxo, um diagrama de fluxo de dados de nível 2 é desenvolvido e mostrado na Figura 10.19. O objeto de dados **comandos do usuário** flui para dentro do sistema e resulta em fluxo de informação adicional ao longo de um dentre três caminhos de ação. Um único item de dados, **tipo de comando**, faz o fluxo de dados fluir de um centro para fora. Assim, a característica global do fluxo de dados é orientada a transações.

Deve-se notar que o fluxo de informação ao longo de dois dos três caminhos de ação acomoda fluxo aferente adicional (por exemplo, **parâmetros e dados do sistema** dão entrada no caminho de ação "configure"). Cada caminho de ação flui para uma única transformação, *mostrar mensagens e estado*.

Os passos do projeto para aplicação de transação são semelhantes e em alguns casos idênticos aos passos para aplicação de transformação (Seção 10.6.3). Uma diferença importante está no mapeamento do DFD na estrutura de software.

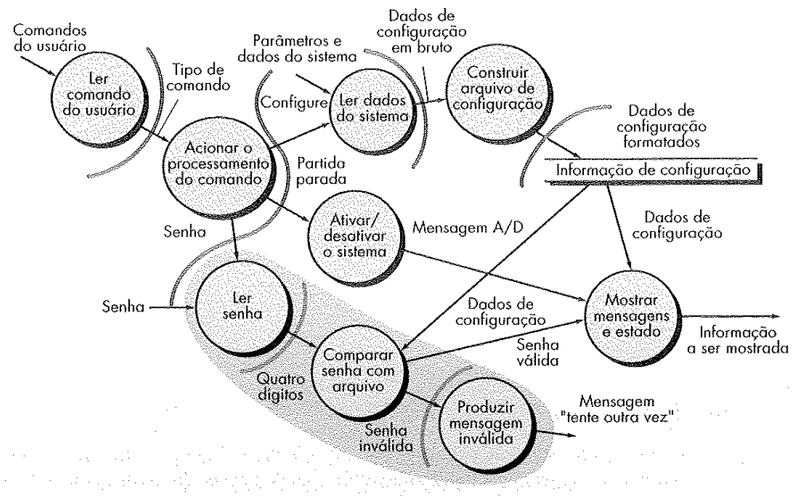
Passo 1. Revise o modelo fundamental do sistema.
Passo 2. Revise e refine os diagramas de fluxo de dados do software.
Passo 3. Determine se o DFD tem características de fluxo de transformação ou de transação.

Os passos 1, 2 e 3 são idênticos aos passos correspondentes no mapeamento da transformação. O DFD mostrado na Figura 10.19 tem uma característica clássica de fluxo de transação. No entanto, o fluxo ao longo de dois dos caminhos de ação que emanam da bolha *acionar o processamento do comando* parece ter características de fluxo de transformação. Assim, limites de fluxo devem ser estabelecidos para ambos os tipos de fluxo.

Passo 4. Identifique o centro de transação e as características de fluxo ao longo de cada um dos caminhos de ação. A localização do centro de transação pode ser imediatamente determinada pelo DFD. O centro de transação fica na origem de um conjunto de caminhos de ação que flui radialmente a partir dele. Para o fluxo mostrado na Figura 10.19, a bolha *invocar o processamento do comando* é o centro de transação.

Document shared on www.docsity.com

FIGURA 10.19 DFD de nível 2 para o subsistema de interação com o usuário



PONTO CHAVE

Fatoração de primeiro nível resulta na derivação de uma hierarquia de controle para o software. Fatoração de segundo nível distribui os módulos "trabalhadores" sob os controladores adequados.

O caminho aferente (o caminho de fluxo ao longo do qual uma transação é recebida) e todos os caminhos de ação devem também ser isolados. Cada caminho de ação deve ser avaliado quanto às suas características de fluxo individuais. Por exemplo, o caminho "senha" (destacado na Figura 10.19) tem características de transformação. Fluxos aferentes, de transformação e eferentes são indicados com limites.

Passo 5. Mapeie o DFD em uma estrutura de programa apropriada ao processamento de transação. O fluxo de transação é mapeado em uma arquitetura que contém um ramo aferente e um ramo de despacho. A estrutura do ramo aferente é desenvolvida de modo bastante semelhante ao do mapeamento de transformação. Começando pelo centro de transação, as bolhas ao longo do caminho aferente são mapeadas em módulos. A estrutura do ramo despachante contém um módulo que controla todos os módulos de ação subordinados. Cada caminho de fluxo de ação do DFD é mapeado em uma estrutura que corresponde às suas características de fluxo específicas. Esse processo é ilustrado esquematicamente na Figura 10.20.

FIGURA 10.20 Mapeamento da transação

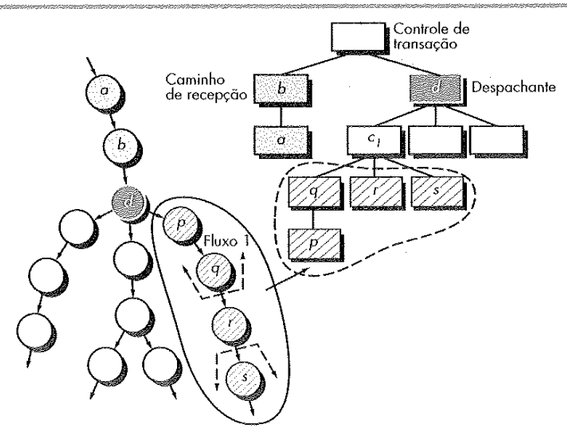
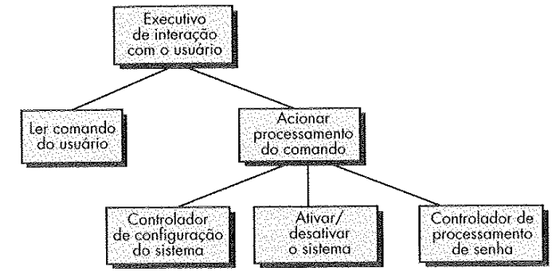


FIGURA 10.21 Fatoração de primeiro nível para o subsistema de interação com o usuário

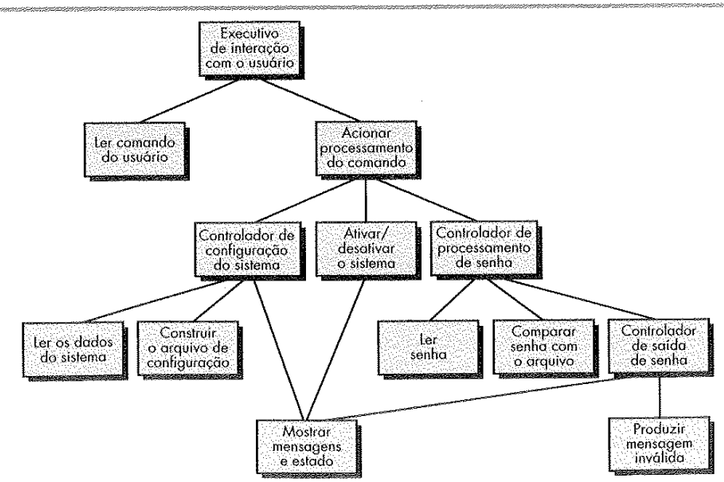


Considerando o fluxo de dados do subsistema de interação com o usuário, a fatoração de primeiro nível para o passo 5 é mostrada na Figura 10.21. As bolhas *ler comando do usuário* e *ativar/ desativar o sistema* são mapeadas diretamente na arquitetura sem a necessidade de módulos de controle intermediários. O centro de transação, *acionar o processamento do comando*, é mapeado diretamente no módulo despachante de mesmo nome. Controladores para o processamento da configuração do sistema e de senha são criados como ilustra a Figura 10.21.

Passo 6. Fatore e refine a estrutura de transação e a estrutura de cada caminho de ação. Cada caminho de ação do diagrama de fluxo de dados tem suas próprias características de fluxo de informação. Já mencionamos que fluxo de transformação ou de transação podem ser encontrados. A "subestrutura" do caminho de ação correspondente é desenvolvida usando os passos de projeto discutidos nesta seção e na anterior.

Como exemplo, considere o fluxo de informação do processamento da senha mostrado (dentro da área sombreada) na Figura 10.19. O fluxo exibe características clássicas de transformação. Uma **senha** é dada como entrada (fluxo aferente) e transmitida a um centro de transformação no qual ela é comparada com senhas armazenadas. Uma mensagem de alarme e alerta (fluxo eferente) é produzida (se não for obtida coincidência). O caminho "configure" é desenhado analogamente usando o mapeamento de transformação. A arquitetura de software resultante é mostrada na Figura 10.22.

FIGURA 10.22 Primeira iteração da arquitetura para o subsistema de interação com o usuário



Document shared on www.docsity.com

Passo 7. Refine a primeira iteração da arquitetura usando heurísticas de projeto para aperfeiçoar a qualidade do software. Este passo para o mapeamento da transação é idêntico àquele correspondente para o mapeamento da transformação. Em ambas as abordagens de projeto, critérios como independência modular, praticidade (eficácia de implementação e teste) e manutenibilidade devem ser cuidadosamente considerados à medida que modificações estruturais são propostas.

> "Faça-o tão simples quanto possível. Mas, não simplista."
>
> Albert Einstein

10.6.5 Refinamento do Projeto Arquitetural

Qualquer discussão de refinamento do projeto deve ter como prefácio o seguinte comentário: lembre-se de que um "projeto ótimo" que não funciona tem mérito questionável. O projetista de software deve estar preocupado em desenvolver uma representação do software que satisfaça a todos os requisitos funcionais e de desempenho e que mereça aceitação com base em medidas e heurísticas de projeto.

Durante os primeiros estágios do projeto, o refinamento da arquitetura de software deve ser encorajado. Como discutimos neste capítulo, estilos arquiteturais alternativos podem ser criados, refinados e avaliados para a "melhor" abordagem. Essa abordagem de otimização é um dos benefícios derivados do desenvolvimento de uma representação da arquitetura do software.

É importante notar que a simplicidade estrutural freqüentemente reflete tanto elegância quanto eficiência. O refinamento do projeto deve procurar o menor número de componentes que seja consistente com efetiva modularidade e a estrutura de dados menos complexa que sirva adequadamente aos requisitos da informação.

10.7 Resumo

A arquitetura do software fornece uma visão holística do sistema a ser construído. Mostra a estrutura e a organização dos componentes de software, suas propriedades e as conexões entre elas. Os componentes de software incluem os módulos de programa e as várias representações de dados manipuladas pelo programa. Assim, o projeto dos dados é uma parte integral da derivação da arquitetura do software. A arquitetura enfatiza as decisões iniciais de projeto e fornece um mecanismo para considerar os benefícios de estruturas alternativas do sistema.

O projeto de dados traduz os objetos de dados definidos no modelo de análise em estruturas de dados localizadas no software. Os atributos que descrevem o objeto, os relacionamentos entre os objetos de dados e seu uso no programa, todos influenciam a escolha das estruturas de dados. Em um nível mais alto de abstração, o projeto de dados pode levar à definição de uma arquitetura para um banco de dados ou para um armazém de dados.

Diversos estilos e padrões arquiteturais diferentes estão disponíveis para o engenheiro de software. Cada estilo descreve uma categoria de sistema que (1) inclui um conjunto de componentes que realizam uma função necessária para o sistema, (2) um conjunto de conectores que permitem comunicação, coordenação e cooperação entre componentes, (3) restrições que definem como os componentes podem ser integrados para formar o sistema, e (4) modelos semânticos que permitem a um projetista entender as propriedades globais de um sistema.

De modo geral, o projeto arquitetural é realizado por meio de quatro passos distintos. Primeiro, o sistema deve ser representado no contexto, isto é, o projetista deve definir as entidades externas com as quais o software interage e a natureza da interação. Uma vez especificado o contexto, ele deve identificar um conjunto de abstrações de mais alto nível chamado de arquétipos que representam os elementos-chave do comportamento ou função do sistema. Definidas as abstrações, o projetista começa a chegar mais perto do domínio de implementação. Componentes são identificados e representados no contexto de uma arquitetura que lhes dá suporte. Por fim, instanciações específicas da arquitetura são desenvolvidas para "provar" o projeto em um contexto do mundo real.

Como um exemplo simples de um projeto arquitetural, o método de mapeamento apresentado neste capítulo utiliza características de fluxo de dados para derivar o estilo arquitetural comumente usado. Um diagrama de fluxo de dados é mapeado em uma estrutura de programa por meio de uma de duas abordagens de mapeamento — mapeamento de transformação ou mapeamento de transação. Uma vez definida a arquitetura, ela é elaborada e depois analisada quanto a critérios de qualidade.

Referências Bibliográficas

[AHO83] Aho, A. V., Hopcroft; J., e Ullmann, J., *Data Structures and Algorithms*, Addison-Wesley, 1983.

[ALL97] Allen R., "A Formal Approach to Software Architecture", Tese de Ph.D., Carnegie Mellon University, Technical Report Number: CMU-CS-97-144 1997.

[BAR00] Barroca, L. e Hall L., (Eds.), *Software Architecture: Advances and Applications*, Springer-Verlag, 2000.

[BAS03] Bass, L., Clements; P. e Kazman, R., *Software Architecture in Practice*, 2ª ed., Addison-Wesley, 2003.

[BOS00] Bosch, J., *Design & Use of Software Architectures*, Addison-Wesley, 2000.

[BUS96] Buschmann, F., *Pattern-Oriented Software*, Wiley, 1996.

[DAT00] Date, *C. J., An Introduction to Database Systems*, 7ª ed., Addison-Wesley, 2000.

[DIK00] Dikel, D., Kane, D. e Wilson, J., *Software Architecture: Organizational Principles and Patterns*, Prentice-Hall, 2000.

[FRE80] Freeman, P., "The Context of Design", em *Software Design Techniques*, 3ª ed. (P. Freeman; A. Wasserman, Eds.), IEEE Computer Society Press, 1980, p. 2-4.

[GAR94] Garlan D., Allen, R., e Ockerbloom, J., "Exploiting Style in Architectural Design Environments", em *Proceedings of SIGSOFT '94 Symposium on the Foundations of Software Engineering*, 1994.

[GAR00] _____, Monroe, R. T., e Wile, D., "Acme: Architectural Description of Component-Based Systems", em *Foundations of Component-Based Systems*, G. T. Leavens e M. Sitarman (eds.), Cambridge University Press, 2000.

[HOF00] Hofmeister, C., Nord, R. e Soni, D., *Applied Software Architecture*, Addison-Wesley, 2000.

[HOF01] Hofmann, C. *et al.*, "Approaches to Software Architecture", disponível em http://citeseer.nj.nec.com/84015.html.

[KAZ98] Kazman, R. *et al.*, *The Architectural Tradeoff Analysis Method*, Software Engineering Institute, CMU/SEI-98-TR-008, jul. 1998.

[KIM98] Kimball, R., Reeves, L. e Ross, M., Thornthwaite, W., *The Data Warehouse Lifecycle Toolkit: Expert Methods for Designing, Developing, and Deploying Data Warehouses*, Wiley, 1998.

[LAN02] Land R., A Brief Survey of Software Architecture, Technical Report, Dept. of Computer Engineering, Mälardalen University, Suécia, fev. 2002.

[LUC95] Luckham, D. C. *et al.*, "Specification and Analysis of System Architecture Using Rapide", *IEEE Transactions on Software Engineering*, issue "Special Issue on Software Architecture", 1995.

[MAT96] Mattison, R., *Data Warehousing: Strategies, Technologies and Techniques*, McGraw-Hill, 1996.

[MYE78] Myers, G., *Composite Structured Design*, Van Nostrand, 1978.

[PRE98] Preiss, B. R., *Data Structures and Algorithms: With Object-Oriented Design Patterns in C++*, Wiley, 1998.

[SHA96] Shaw, M. e Garlan, D., *Software Architecture*, Prentice-Hall, 1996.

[SHA97] _____ e Clements, P., "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems", *Proc. COMPSAC*, Washington, DC, agos 1997.

[WAS80] Wasserman, A., "Principles of Systematic Data Design and Implementation", em *Software Design Techniques* (P. Freeman e A. Wasserman, Eds.), 3ª ed., IEEE Computer Society Press, 1980, p. 287-293.

[YOU79] Yourdon, E.; e Constantine, L., *Structured Design*, Prentice-Hall, 1979.

[ZHA98] Zhao, J., "On Assessing the Complexity of Software Architectures", *Proc. Intl. Software Architecture Workshop*, ACM, Orlando, FL, p. 163-167, 1998.

Problemas e Pontos a Considerar

10.1. Usando a arquitetura de uma casa ou edifício como metáfora, faça comparações com a arquitetura do software. Em que as disciplinas de arquitetura clássica e a arquitetura de software são semelhantes? Em que são diferentes?

10.2. Escreva um trabalho de três a cinco páginas que apresente diretrizes para selecionar estruturas de dados com base na natureza do problema. Comece delineando as estruturas de dados clássicas encontradas no trabalho de software e depois descreva critérios para delas selecionar tipos de problema particulares.

Document shared on www.docsity.com

10.3. Explique a diferença entre uma base de dados que serve a uma ou mais aplicações convencionais de negócios e um armazém de dados.

10.4. Apresente dois ou três exemplos de aplicações para cada um dos estilos arquiteturais mencionados na Seção 10.3.1.

10.5. Alguns dos estilos arquiteturais mencionados na Seção 10.3.1 são, por natureza, hierárquicos e outros não. Faça uma lista de cada tipo. Como os estilos arquiteturais não hierárquicos seriam implementados?

10.6. Os termos *estilo arquitetural*, *padrão arquitetural* e *framework* são freqüentemente encontrados na discussão de arquitetura de software. Faça alguma pesquisa (use a Internet) e descreva como cada um desses termos difere dos outros.

10.7. Selecione uma aplicação com a qual você está familiarizado. Responda a cada uma das questões formuladas na Seção 10.3.3 sobre controle e dados.

10.8. Pesquise o ATAM (use o site SEI) e apresente uma discussão detalhada dos seis passos apresentados na Seção 10.5.1.

10.9. Alguns projetistas argumentam que todo o fluxo de dados pode ser tratado como orientado à transformação. Discuta como esse argumento vai afetar a arquitetura do software originada quando um fluxo orientado à transação é tratado como transformação. Use um fluxo como exemplo para ilustrar pontos importantes.

10.10. Se você ainda não fez, complete o Problema 8.10. Use os métodos de projeto descritos neste capítulo para desenvolver uma arquitetura de software para o SARB.

10.11. Usando um diagrama de fluxo de dados e uma narrativa de processamento, descreva um sistema com base em computador que tenha características distintas de fluxo de transformação. Defina limites de fluxo e mapeie o DFD em uma estrutura de software usando a técnica descrita na Seção 10.6.3.

10.12. Usando um diagrama de fluxo de dados e uma narrativa de processamento, descreva um sistema com base em computador que tenha características de fluxo de transação distintas. Defina limites de fluxo e mapeie o DFD em uma estrutura de software usando a técnica descrita na Seção 10.6.4.

Leituras e Fontes de Informação Adicionais

A literatura sobre arquitetura de software explodiu durante a última década. Livros de Fowler (*Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003), Clements e seus colegas (*Documenting Software Architecture: View and Beyond*, Addison-Wesley, 2002), Schmidt e seus colegas (*Pattern-Oriented Software Architectures*, dois volumes, Wiley, 2000) Bosch [BOSOO], Dikel e seus colegas [DIK00], Hofmeister e seus colegas [HOF00], Bass, Clements e Kazman [BAS03], Shaw e Garlan [SHA96], e Buschmann *et al.* [BUS96] fornecem um tratamento profundo do assunto. Trabalho anterior de Garlan (*An Introduction to Software Architecture*, Software Engineering Institute, CMU/SEI-94-TR-021, 1994) fornece uma excelente introdução. Clements e Northrop (*Software Product Lines: Practices and Patterns*, Addison-Wesley, 2001) atendem ao projeto de arquiteturas que suportam linhas de produto de software. Clements e seus colegas (*Evaluating Software Architectures*, Addison-Wesley, 2002) consideram os tópicos associados com a avaliação de alternativas arquiteturais e a seleção da melhor arquitetura para um dado domínio de problema.

Livros específicos de implementação arquitetural tratam de projeto arquitetural dentro de um ambiente ou tecnologia de desenvolvimento específico. Wallnau e seus colegas (*Building Systems from Commercial Components*, Addison-Wesley, 2001) apresentam métodos para construir arquiteturas baseadas em componente. Pritchard (*COM and CORBA Side-by-Side*, Addison-Wesley, 1999), Mowbray (*CORBA Design Patterns*, Wiley, 1997) e Mark *et al.* (*Object Management Architecture Guide*, Wiley, 1996) fornecem diretrizes de projeto detalhadas para o framework CORBA de apoio a aplicações distribuídas. Shanley (*Protected Mode Software Architecture*, Addison-Wesley, 1996) fornece diretrizes para o projeto arquitetural para quem quer que esteja projetando sistemas operacionais de tempo real baseados em PC, sistemas operacionais multitarefa ou acionadores de dispositivos.

Pesquisa atual sobre arquitetura de software é documentada anualmente nos *Proceedings of the International Workshop on Software Architecture*, patrocinado pela ACM e outras organizações computacionais, e nos *Proceedings of the International Conference on Software Engineering*. Barroca e Hall [BAR00] apresentam um levantamento útil de pesquisas recentes.

A modelagem de dados é pré-requisito para um bom projeto de dados. Livros de Teory (*Database Modeling and Design*, Academic Press, 1998); Schmidt (*Data Modeling for Information Professionals*, Prentice-Hall, 1998); Bobak (*Data Modeling and Design for Today's Architectures*, Artech House, 1997); Silverston, Graziano, e Inmon (*The Data Model Resource Book*, Wiley, 1997); Date [DAT00], e Reingruber e Gregory (*The Data Modeling Handbook: A Best-Practice Approach to Building Quality Data Models*, Wiley, 1994) contêm apresentações detalhadas de notação

Document shared on www.docsity.com

e heurísticas da modelagem de dados e abordagens de projeto de bancos de dados. O projeto de armazém de dados têm se tornado cada vez mais importante nos últimos anos. Os livros de Humphreys, Hawkins e Dy (*Data Warehousing: Architecture and Implementation*, Prentice-Hall, 1999); Kimball *et al.* [KIM98]; e Inmon [INM95] abordam o tópico com detalhes consideráveis.

Tratamento geral para projeto de software com discussão de tópicos de projeto arquitetural e de dados pode ser encontrado na maioria dos livros dedicados à engenharia de software. Tratamento mais rigoroso do assunto pode ser encontrado em Feijs (*A Formalization of Design Methods*, Prentice-Hall, 1993), Witt *et al.* (*Software Architecture and Design Principles*, Thomson Publishing, 1994), e Budgen (*Software Design*, Addison-Wesley, 1994).

Apresentações completas de projeto orientado a fluxo de dados podem ser encontradas em [MYE78], Yourdon e Constantine [YOU79], e Page-Jones (*The Practical Guide to Structured Systems Design*, segunda edição, Prentice-Hall, 1988). Esses livros são dedicados somente ao projeto e fornecem tutorais abrangentes da abordagem orientada a fluxo de dados.

Uma ampla variedade de fontes de informação sobre projeto arquitetural está disponível na Internet. Uma lista atualizada de referências relevantes pode ser encontrada no site deste livro: **http://www.mhhe.com/pressman**.

CAPÍTULO

11 PROJETO NO NÍVEL DE COMPONENTES

CONCEITOS-CHAVE

acoplamento 248
coesão 246
componentes
convencionais 256
orientados a objetos ... 239
relacionados a processo . 243
middleware 243
OCL 255
PDL 259
programação estruturada . 257
projeto
diretrizes 246
notação gráfica 257
princípios 243
tarefas 250

O projeto no nível de componentes ocorre depois que a primeira iteração do projeto arquitetural tiver sido completada. Nesse estágio, a estrutura global dos dados e programas do software foi estabelecida. O objetivo é traduzir o modelo de projeto no software operacional. Mas o nível de abstração do modelo de projeto existente é relativamente alto e o nível de abstração do programa operacional é baixo. A tradução pode ser complicada, abrindo as portas à introdução de erros sutis, difíceis de encontrar e corrigir em estágios posteriores do processo de software. Em uma palestra famosa, Edsgar Dijkstra, um contribuinte importante para o nossa compreensão do que é projeto, declarou [DIJ72]:

> O software parece ser diferente de muitos outros produtos em que, como regra, alta qualidade implica alto preço. Os que querem realmente software confiável descobrirão que precisam encontrar meios para evitar de início a maioria dos erros e, como resultado, o processo de programação se tornará mais barato... programadores efetivos... não devem desperdiçar seu tempo na depuração — eles não devem introduzir erros já no início.

Apesar de essas palavras terem sido ditas anos atrás, elas permanecem verídicas até hoje. Quando o modelo de projeto é traduzido em código-fonte, devemos seguir um conjunto de princípios de projeto que não apenas efetua a tradução, mas também não "introduz erros desde o início".

PANORAMA

O que é? Um conjunto completo de componentes de software é definido durante o projeto arquitetural. Mas as estruturas de dados internas e detalhes de processamento de cada componente não são representados em um nível de abstração próximo ao código. Projeto no nível de componente define as estruturas de dados, algoritmos, características de interface e mecanismos de comunicação alocados a cada componente de software.

Quem faz? Um engenheiro de software realiza o projeto no nível de componentes.

Por que é importante? Você precisa poder determinar se o software vai funcionar antes de construí-lo. O projeto no nível de componentes representa o software de um modo que lhe permite revisar os detalhes do projeto quanto à correção e consistência com as primeiras representações do projeto (ou seja, os projetos de dados, arquitetural e de interface). Ele fornece meios para avaliar se as estruturas de dados, interfaces e algoritmos vão funcionar.

Quais são os passos? Representações de projeto dos dados, arquitetura e interfaces formam a fundação para o projeto no nível de componentes. A definição de classes ou narrativa de processamento para cada componente é traduzida em um projeto detalhado que faz uso de formas diagramáticas ou baseadas em texto que especificam as estruturas de dados internas, detalhes de interface local e lógica de processamento. A notação de projeto inclui diagramas UML e representações suplementares. O projeto procedural é especificado por meio de um conjunto de construções da programação estruturada.

Qual é o produto do trabalho? O projeto de cada componente, representado em notação gráfica, tabular ou baseada em texto, é o principal produto de trabalho produzido durante o projeto no nível de componente.

Como tenho certeza de que fiz corretamente? Um *walk-through*, ou inspeção de projeto, é conduzido. O projeto é examinado para determinar se as estruturas de dados, interfaces, seqüências de processamento e condições lógicas estão corretas e vão produzir as transformações adequadas dos dados ou controle atribuídas ao componente durante os primeiros passos de projeto.

É possível representar o projeto no nível de componentes usando uma linguagem de programação. Em essência, o programa é criado por meio do modelo de projeto arquitetural como diretriz. Uma abordagem alternativa é representar o projeto no nível de componente usando alguma representação intermediária (gráfica, tabular ou baseada em texto), que pode ser traduzida facilmente para código-fonte. Independentemente do mecanismo usado para representar o projeto no nível de componente, as estruturas de dados, interfaces e os algoritmos definidos devem estar de acordo com diversas diretrizes bem estabelecidas de projeto que nos ajudam a evitar erros, à medida que o projeto procedural evolui. Neste capítulo, examinaremos essas diretrizes de projeto e os métodos disponíveis para segui-las.

11.1 O QUE É UM COMPONENTE?

Definido de forma geral, *componente* é um bloco construtivo modular para software de computador. Mais formalmente, a Especificação da Linguagem de Modelagem Unificada da OMG (*OMG Unified Modeling Language Specification*) [OMG01] define um componente como "parte modular, possível de ser implantada e substituível de um sistema que encapsula implementação e exibe conjunto de interfaces".

Como discutimos no Capítulo 10, componentes preenchem a arquitetura do software e, como conseqüência, desempenham um papel para alcançar os objetivos e requisitos do sistema a ser construído. Da mesma forma que os componentes residentes na arquitetura do software, eles devem se comunicar e colaborar com outros componentes e com entidades (por exemplo, outros sistemas, dispositivos e pessoas) que existem fora do limite do software.

> "Os detalhes não são detalhes. Eles formam o projeto."
>
> **Charles Eames**

O verdadeiro significado do termo "componente" diferirá dependendo do ponto de vista do engenheiro de software que o usa. Nas próximas seções, examinaremos três importantes visões do que um componente é e de como ele é usado à medida que a modelagem de projeto prossegue.

11.1.1 Uma Visão Orientada a Objetos

PONTO CHAVE

Do ponto de vista OO, um componente é um conjunto de classes colaborativas.

No contexto da engenharia de software orientada a objetos, o componente contém um conjunto de classes colaborativas[1]. Cada classe em um componente é completamente elaborada para incluir todos os atributos e operações relevantes para sua implementação. Como parte da elaboração do projeto, todas as interfaces (mensagens) que habilitam as classes a comunicar-se e colaborar com outras classes de projeto devem ser também definidas. Para tanto, o projetista começa com o modelo de análise e elabora as classes de análise (para componentes que se relacionam ao domínio do problema) e classes de infra-estrutura (para componentes que fornecem serviços de apoio ao domínio do problema).

Para ilustrar esse processo de elaboração de projeto, considere o software a ser construído para um complexo gráfico. O objetivo global do software é coletar os requisitos do cliente em um balcão frontal, orçar o serviço de impressão e, então, passar a tarefa adiante para uma instalação de produção automatizada. Durante a engenharia de requisitos, foi derivada uma classe de análise chamada **ServiçoDeImpressão**. Os atributos e operações definidos durante a análise foram anotados na parte superior esquerda da Figura 11.1. Durante o projeto arquitetural, **ServiçoDeImpressão** é definido como um componente dentro da arquitetura de software e é representado pela notação abreviada UML mostrada no centro à direita da figura. Note que **ServiçoDeImpressão** tem duas interfaces, *cálculoDeServiço*, que fornece a capacidade de orçar um serviço, e *inícioDeServiço*, que passa o serviço adiante para a instalação de produção. Essas são representadas usando os símbolos de "pirulito" mostrados à esquerda da caixa de componente.

1 Em alguns casos, um componente pode conter uma única classe.

Document shared on www.docsity.com

AVISO

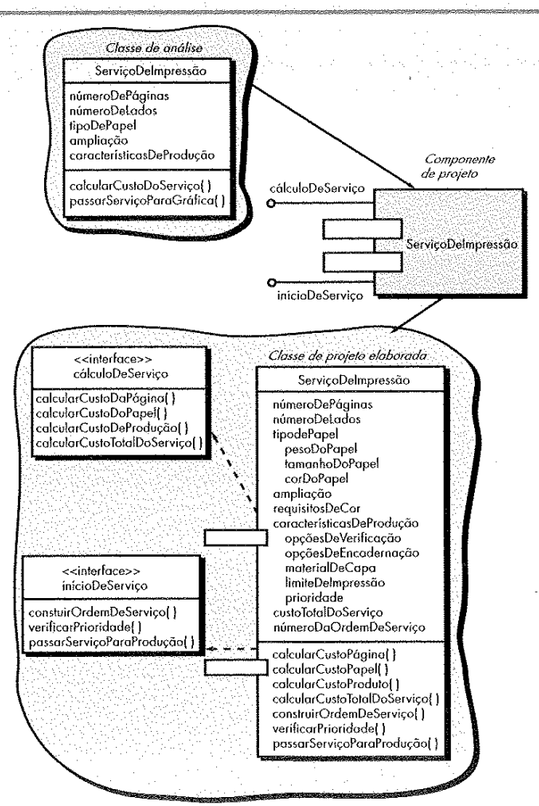
Vale lembrar que modelagem de análise e modelagem de projeto são ambas ações iterativas. Elaboração de classes originais de análise pode necessitar de passos adicionais de análise, que são então seguidos por passos de modelagem de projeto para representar a classe de projeto elaborada (os detalhes do componente).

Projeto no nível de componente começa nesse ponto. Os detalhes do componente **ServiçoDeImpressão** devem ser elaborados para fornecer informação suficiente para guiar a implementação. A classe de análise original é elaborada para evidenciar todos os atributos e operações necessários para implementar a classe como o componente **ServiçoDeImpressão**. Referindo-se à parte inferior direita da Figura 11.1, a classe de projeto elaborado **ServiçoDeImpressão** contém informação de atributo mais detalhada, bem como uma descrição expandida das operações necessárias para implementar o componente. As interfaces *cálculoDeServiço* e *inícioDeServiço* implicam comunicação e colaboração com outros componentes (não mostrados aqui). Por exemplo, a operação *calcularCustoDaPágina* (parte da interface *cálculoDeServiço*) pode colaborar com um componente **TabelaDaPreço** que contém informação de preço dos serviços. A operação *verificarPrioridade()* (parte da interface *inícioDeServiço)* pode colaborar com o componente **FilaDeServiço** para determinar os tipos e prioridades dos serviços atualmente em espera para produção.

Essa atividade de elaboração é aplicada a todo componente definido como parte do projeto arquitetural. Depois de completado, mais elaboração é aplicada para cada atributo, operação e interface. As estruturas de dados adequadas para cada atributo devem ser especificadas. Além disso, é projetado o detalhe algorítmico necessário para implementar a lógica de processamento associada a cada operação. Essa atividade de projeto procedural é discutida mais tarde neste capítulo. Por fim, são projetados os mecanismos necessários para implementar a interface. Para software OO, isso pode incluir a descrição de todas as mensagens necessárias para efetuar a comunicação entre objetos de um sistema.

FIGURA 11.1

Elaboração de um componente de projeto



11.1.2 A Visão Convencional

No contexto da engenharia de software convencional, o componente é um elemento funcional de programa que incorpora lógica de processamento, estruturas de dados internas que são necessárias para implementar a lógica de processamento e uma interface que possibilita ao componente ser chamado e dados serem passados para ele. Um componente convencional, também denominado *módulo*, reside dentro da arquitetura de software e serve a um dos três importantes papéis como (1) um *componente de controle* que coordena a chamada de todos os outros componentes do domínio do problema, (2) um *componente do domínio do problema* que implementa uma função completa ou parcial que é solicitada pelo cliente, ou (3) um *componente de infra-estrutura* responsável pelas funções que suportam o processamento necessário para o domínio do problema.

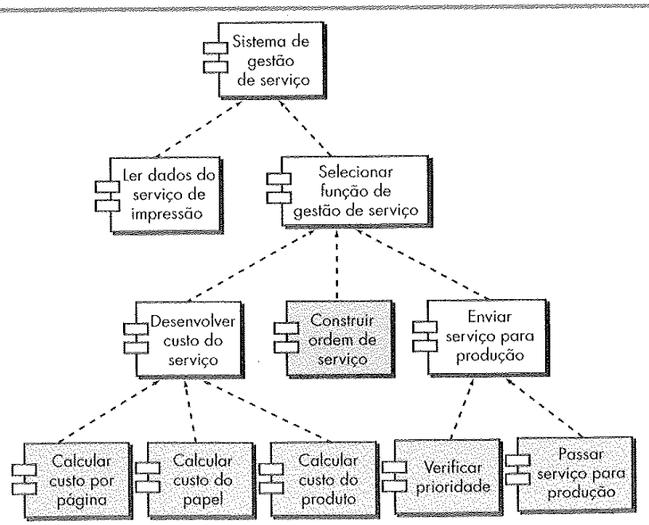
Como componentes orientados a objetos, componentes de software convencionais são derivados de modelos de análise. Neste caso, no entanto, o elemento orientado a fluxo de dados do modelo de análise serve como base para a derivação. Cada transformação (bolha) representada nos níveis mais baixos do diagrama de fluxo de dados (veja o Capítulo 8) é mapeada (veja a Seção 10.6) em uma hierarquia de módulos. Componentes de controle (módulos) residem perto do topo da hierarquia (arquitetura), e componentes do domínio do problema tendem a residir mais próximo do nível mais baixo da hierarquia. Para conseguir modularidade efetiva, conceitos de projeto como independência funcional (veja o Capítulo 9) são aplicados quando os componentes são elaborados.

> "É invariavelmente constatado que um sistema complexo que funciona evoluiu de um sistema simples que funcionava."
>
> **John Gall**

Para ilustrarmos esse processo de elaboração de projeto para componentes convencionais, consideremos outra vez o software a ser construído para um sofisticado centro gráfico. Um conjunto de diagramas de fluxo de dados teria sido derivado durante a modelagem de análise. Consideraremos que esses foram mapeados (veja a Seção 10.6) em uma arquitetura mostrada na Figura 11.2. Cada caixa representa um componente de software. Note que as caixas sombreadas são equivalentes em função às operações definidas para a classe **ServiçoDeImpressão** discutida na Seção 11.1.1. Neste caso, no entanto, cada operação é representada como um módulo separado que é chamado como mostra a figura. Outros módulos são usados para controlar o processamento e são, por essa razão, componentes controladores.

FIGURA 11.2

Diagrama de estrutura para um sistema convencional



Document shared on www.docsity.com

Durante o projeto no nível de componente, cada módulo da Figura 11.2 é elaborado. A interface do módulo é definida explicitamente. Ou seja, cada objeto de dados ou controle que flui pela interface é representado. São definidas as estruturas de dados usadas internamente ao módulo. O algoritmo que permite ao módulo realizar sua função é projetado usando a abordagem de refinamento passo a passo, discutida no Capítulo 9. O comportamento do módulo é, algumas vezes, representado por um diagrama de estado.

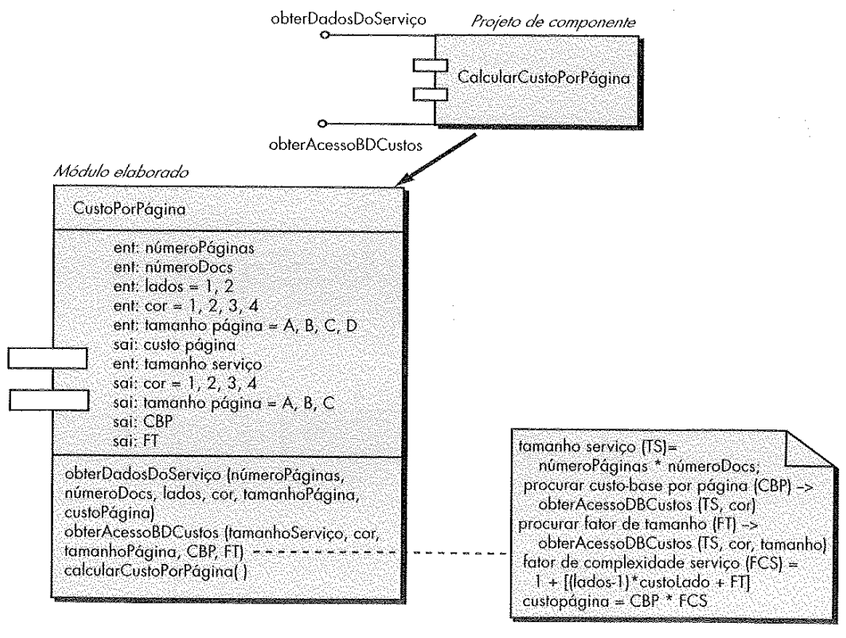
AVISO

Quando o projeto para cada componente de software é elaborado, o foco desloca-se para o projeto de estruturas de dados específicas e projeto procedural para manipular as estruturas de dados. No entanto, não esqueça a arquitetura que precisa alojar os componentes ou as estruturas de dados globais que podem servir a muitos componentes.

Para ilustrar esse processo, considere o módulo *CalcularCustoPorPágina*. O objetivo desse módulo é calcular o custo de impressão por página com base nas especificações fornecidas pelo cliente. Os dados necessários para executar essa função são: **número de páginas nesse documento, número total de documentos a ser produzido, impressão de um ou dos dois lados, requisitos de cor, requisitos de tamanho**. Esses dados são passados para *CalcularCustoPorPágina* via interface do módulo. *CalcularCustoPorPágina* usa esses dados para determinar um custo por página baseado no tamanho e na complexidade do serviço — uma função de todos os dados passados para o módulo pela interface. O custo da página é inversamente proporcional ao tamanho do serviço e diretamente proporcional à complexidade do serviço.

A Figura 11.3 representa o projeto no nível de componente usando uma notação UML modificada. O módulo *CalcularCustoPorPágina* tem acesso aos dados pela chamada ao método *obterDadosDoServiço* que permite que todos os dados relevantes sejam passados para o componente, e uma interface do banco de dados, *obterAcessoBDCustos*, que habilita o módulo a ter acesso a um banco de dados que contém todos os custos de impressão. À medida que o projeto continua, o módulo *CalcularCustoPorPágina* é elaborado para fornecer detalhe de algoritmo e interface (veja a Figura 11.3). O detalhe de um algoritmo pode ser representado por meio do texto em pseudocódigo mostrado na figura ou com um diagrama de atividade UML. As interfaces são representadas como uma coleção de objetos ou itens de dados de entrada e saída. A elaboração do projeto continua até que detalhe suficiente seja fornecido para guiar a construção do componente.

FIGURA 11.3 Projeto no nível de componente de *CalcularCustoPorPágina*



11.1.3 Uma Visão Relacionada a Processo

As visões de projeto no nível de componente orientado a objetos e convencional, apresentadas nas seções anteriores, consideram que o componente está sendo projetado a partir do zero. Isto é, o projetista precisa criar um novo componente baseado na especificação derivada do modelo de análise. Há, naturalmente, uma outra abordagem.

Durante a última década, a comunidade de engenharia de software enfatizou a necessidade de construir sistemas que façam uso de componentes de software existentes. Em resumo, um catálogo no nível de projetos ou código de componentes comprovados torna-se disponível aos engenheiros de software à medida que o trabalho prossegue. Quando a arquitetura de software está desenvolvida, componentes ou padrões de projeto são escolhidos do catálogo e usados para preencher a arquitetura. Como esses componentes têm sido criados com reusabilidade em mente, uma descrição completa de sua interface, das funções que realizam e a comunicação e colaboração que eles requerem estão todas disponíveis ao projetista. A engenharia de software baseada em componentes é discutida detalhadamente no Capítulo 30.

FERRAMENTAS DE SOFTWARE

Middleware e Engenharia de Software Baseada em Componentes

Um dos elementos-chave que conduz ao sucesso ou falha do Desenvolvimento de Software Baseado em Componentes (DSBC)[CBSE, em inglês] é a disponibilidade de middleware. *Middleware* é uma coleção de componentes de infra-estrutura que permitem aos componentes do domínio do problema comunicarem-se entre eles mesmos, em uma rede ou dentro de um sistema complexo. Três normas concorrentes estão disponíveis aos engenheiros de software que querem usar engenharia de software baseada em componentes como seu processo de software:

OMG CORBA (http://www.corba.org/)

Microsoft COM (http://www.microsoft.com/com/tech/complus.asp)

Sun JavaBeans (http://java.sun.com/products/ejb/).

Os sites mencionados apresentam grande quantidade de tutoriais, *white papers*, ferramentas e recursos gerais sobre essas importantes normas de middleware. Mais informação sobre DSBC pode ser encontrada no Capítulo 30.

11.2 PROJETO DE COMPONENTES BASEADOS EM CLASSES

Como já observamos, o projeto no nível de componente apóia-se nas informações desenvolvidas como parte do modelo de análise (veja o Capítulo 8) e representadas como parte do modelo arquitetural (veja o Capítulo 10). Quando uma abordagem de engenharia de software orientada a objetos é escolhida, o projeto no nível de componente enfoca a elaboração das classes de análise (classes específicas do domínio do problema), e a definição e refinamento das classes de infra-estrutura. A descrição detalhada dos atributos, operações e interfaces usados por essas classes é o detalhe de projeto requerido como um precursor da atividade de construção.

11.2.1 Princípios Básicos de Projeto

Quatro princípios básicos de projeto são aplicáveis ao projeto no nível de componente e têm sido amplamente adotados quando a engenharia de software orientada a objetos é aplicada. A motivação subjacente à aplicação desses princípios é criar projetos mais fáceis de modificar e reduzir a propagação dos efeitos colaterais quando as modificações ocorrem. Esses princípios podem ser usados para guiar o projetista à medida que cada componente de software é desenvolvido.

Princípio Aberto-Fechado (*Open-Closed Principle* — OCP). *"Um módulo [componente] deveria ser aberto para extensão, mas fechado para modificação"* [MAR00]. Essa afirmação parece ser uma contradição, mas ela representa uma das características mais importantes de um bom projeto no nível de componente. Dito de modo mais simples, o projetista deve especificar o componente de

Document shared on www.docsity.com

tal modo que lhe permita ser estendido (dentro do seu domínio funcional) sem a necessidade de fazer modificações internas (no nível de código ou lógico) ao próprio componente. Para tanto, o projetista cria abstrações que servem como um buffer entre a funcionalidade que é provável de ser estendida e as classes de projeto em si.

Por exemplo, considere que a função de segurança do *CasaSegura* faz uso de uma classe **Detector** que deve verificar o estado de cada tipo de sensor de segurança. É provável que com o passar do tempo, o número e tipos de sensores de segurança cresçam. Se um processador lógico interno é implementado como uma seqüência de construções se-então-senão, cada uma tratando de um tipo de sensor diferente, a adição de um novo tipo de sensor necessitará de lógica de processamento interno adicional (ainda um outro se-então-senão). Isso é uma violação de OCP.

Um modo de conseguir OCP para a classe **Detector** é apresentado na Figura 11.4. A interface *sensor* apresenta uma visão consistente de sensores para o componente **Detector**. Se um novo tipo de sensor é adicionado, nenhuma modificação é necessária na classe (componente) **Detector**. O OCP é preservado.

CASASEGURA

O OCP em Ação

A cena: sala de Vinod.

Os personagens: Vinod e Shakira — membros da equipe de engenharia de software do *CasaSegura*.

A conversa:

Vinod: Acabei de receber um telefonema do Doug [o gerente da equipe]. Ele disse que o marketing quer adicionar um novo sensor.

Shakira (com sorriso malicioso): Não, outra vez!

Vinod: É... e você não vai acreditar no que essa gente está aprontando.

Shakira: Me surpreenda.

Vinod (rindo): Eles chamam de sensor de raiva de cachorro.

Shakira: O quê?

Vinod: É para pessoas que deixam seus animais domésticos em casa, em apartamentos ou condomínios ou casas que estão próximas entre si. O cachorro começa a latir. O vizinho fica bravo e se queixa. Com esse sensor, se o cachorro late durante mais do que, digamos 1 minuto, o sensor dispara um modo especial de alarme que chama o proprietário no seu celular.

Shakira: Você está brincando comigo, certo?

Vinod: Negativo. Doug quer saber quanto tempo vai levar para adicioná-lo à função de segurança.

Shakira (pensando por um momento): Não muito... veja [ela mostra a Vinod a Figura 11.4]. Nós isolamos as classes reais de sensor atrás da interface **sensor**. Desde que a gente tenha as especificações para o sensor do cachorro, adicioná-lo vai ser sopa. A única coisa que vou ter de fazer é criar um componente adequado... humm, classe para ele. Sem qualquer modificação para o componente **Detector.**

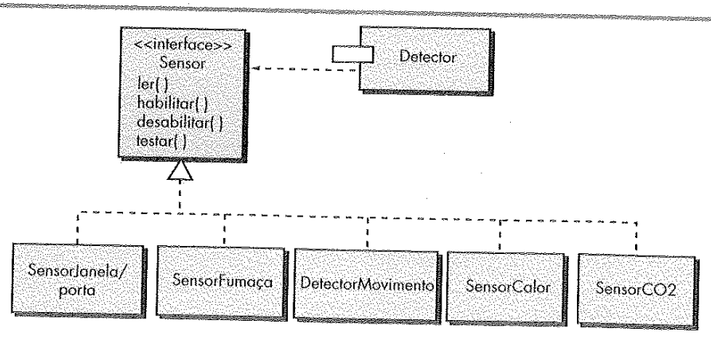
Vinod: Então eu vou dizer ao Doug que não tem problema.

Shakira: Conhecendo Doug, ele nos manterá concentrados e não vai entregar o sensor de cachorro até a próxima versão.

Vinod: Isso não é ruim, mas você pode implementar agora se ele quiser?

FIGURA 11.4

De acordo com o OCP



Shakira: Sim, o modo pelo qual nós projetamos a interface me deixa fazê-lo sem turbulência.

Vinod (pensando um momento): Você já ouviu falar do "princípio Aberto-Fechado"?

Shakira (balançando os ombros): Nunca.

Vinod (sorrindo): Sem problema.

Princípio da Substituição de Liskov (*Liskov Substitution Principle* — LSP). *"Subclasses devem ser substituíveis por suas classes-base"* [MAR00]. O princípio de projeto, originalmente proposto por Barbara Liskov [LIS88] sugere que um componente que usa uma classe-base deve continuar a funcionar adequadamente se uma classe derivada da classe-base é passada para o componente em vez dela. LSP exige que qualquer classe derivada de uma classe-base precisa honrar qualquer contrato implícito entre a classe-base e os componentes que ela utiliza. No contexto dessa discussão, um "contrato" é uma *precondição* que deve ser verdadeira antes de o componente usar uma classe-base, e uma *pós-condição* que deve ser verdadeira depois de o componente usar uma classe-base. Quando um projetista cria classes *derivadas*, elas também precisam satisfazer as pré e pós condições.

AVISO

Se você dispensa o projeto e vai direto ao código, lembre-se apenas que código é a maior "concretização". Você está violando DIP.

Princípio da Inversão de Dependência (*Dependency Inversion Principle* — DIP). *"Confie nas abstrações. Não confie nas concretizações"* [MAR00]. Como vimos na discussão do OCP, abstrações são o lugar onde um projeto pode ser estendido sem maior complicação. Quanto mais um componente depende de outros componentes concretos (em vez de em abstrações como uma interface), mais difícil é estendê-lo.

Princípio da Segregação de Interface (*Interface Segregation Principle* — ISP). *"Muitas interfaces específicas de clientes são melhores do que uma interface de propósito geral"* [MAR00]. Há muitas instâncias nas quais múltiplos componentes de cliente usam as operações fornecidas por uma classe servidora. ISP sugere que o projetista deve criar uma interface especializada para servir cada categoria importante de clientes. Somente aquelas operações relevantes a uma particular categoria de clientes devem ser especificadas na interface daquele cliente. Se múltiplos clientes solicitam as mesmas operações, elas devem ser especificadas em cada uma das interfaces especializadas.

Como exemplo, considere a classe **PlantaBaixa** para as funções de segurança e vigilância do *CasaSegura*. Para as funções de segurança, **PlantaBaixa** é usada somente durante as atividades de configuração e usa as operações *colocarDispositivo(), mostrarDispositivo(), agruparDispositivo()* e *removerDispositivo()* para colocar, mostrar, agrupar e remover sensores da planta baixa. A função de segurança do *CasaSegura* usa as quatro operações mencionadas para segurança, mas também requer operações especiais para gerir câmeras: *mostrarFOV()* e *mostrarIDDispositivo()*. Assim, ISP sugere que os componentes-clientes das duas funções do *CasaSegura* tenham interfaces especializadas definidas por eles. A interface para segurança abrangeria apenas as operações *colocarDispositivo(), mostrarDispositivo(), agruparDispositivo()* e *removerDispositivo()*. A interface para vigilância deveria incorporar as operações *colocarDispositivo(), mostrarDispositivo(), agruparDispositivo()*, e *removerDispositivo(), mostrarFOV()* e *mostrarIDDispositivo()*.

PONTO CHAVE

Projetar componentes para reuso requer mais do que bom projeto técnico. Também requer mecanismos de controle de configuração efetivos (veja o Capítulo 27).

Princípio de Equivalência de Liberação de Reuso (*Release Reuse Equivalency Principle* — REP). *"A granularidade do reuso é a granularidade da versão"* [MAR00]. Quando classes ou componentes são projetados para reuso, há um contrato implícito estabelecido entre o desenvolvedor da entidade reusável e quem irá usá-la. O desenvolvedor se compromete a estabelecer um sistema de controle de versão que suporta e mantém versões anteriores da entidade enquanto os usuários lentamente atualizam-se para a versão mais nova. Em vez de tratar cada classe individualmente, é freqüentemente aconselhável agrupar as classes reusáveis em pacotes que podem ser geridos e controlados à medida que versões mais atuais evoluem.

Princípio de Fecho Comum (*Common Closure Principle* — CCP). *"Classes que se modificam juntas devem ficar juntas"* [MAR00]. Classes devem ser empacotadas coesivamente, isto é, quando classes são empacotadas como parte de um projeto, elas devem tratar da mesma área funcional

Document shared on www.docsity.com

e comportamental. Quando algumas características daquela área precisam ser modificadas, é provável que apenas aquelas classes dentro do pacote vão precisar de modificação. Isso leva a um controle de modificação mais efetivo e gestão de versão.

Princípio Comum de Reuso (*Common Reuse Principle* — CRP). *"Classes que não são reusadas juntas não devem ser agrupadas juntas"* [MAR00]. Quando uma ou mais classes em um pacote são modificadas, o número da versão do pacote se modifica. Todas as outras classes ou pacotes que se apóiam no pacote modificado devem agora ser atualizadas para a versão mais recente do pacote e serem testadas para garantir que a nova versão opera sem incidente. Se classes não são agrupadas de maneira coesiva, é possível que uma classe sem relacionamento com outras classes do pacote seja modificada. Isso precipitará integração e teste desnecessários. Por essa razão, somente classes reusadas juntas devem ser incluídas em um pacote.

11.2.2 Diretrizes para Projeto no Nível de Componente

Em adição aos princípios discutidos na Seção 11.2.1, um conjunto de diretrizes pragmáticas de projeto pode ser aplicado à medida que o projeto no nível de componente prossegue. Essas diretrizes aplicam-se a componentes, suas interfaces e características de dependências e de herança que têm impacto sobre o projeto resultante. Ambler [AMB02] sugere as seguintes diretrizes:

? O que devemos considerar quando damos nomes a componentes?

Componentes. Convenções de nomes devem ser estabelecidas para componentes especificados como parte do modelo arquitetural e, então, refinados e elaborados como parte do modelo no nível de componente. Os nomes dos componentes arquiteturais devem ser extraídos do domínio do problema e ter significado para todos os interessados que utilizam o modelo arquitetural. Por exemplo, o nome da classe **PlantaBaixa** é significativo para todos os que a lêem independentemente do seu conhecimento técnico. Por outro lado, componentes de infra-estrutura ou classes elaboradas no nível de componentes devem ser denominados para refletir o significado específico de implementação. Se uma lista encadeada tiver de ser gerida como parte da implementação de **PlantaBaixa**, a operação *gerirLista()* é adequada, mesmo que alguém sem conhecimento técnico possa vir a interpretá-la mal.[2]

Também vale a pena usar estereótipos para ajudar a identificar a natureza dos componentes no nível de projeto detalhado. Por exemplo, **<<infra-estrutura>>** pode ser usada para identificar um componente infra-estrutura; **<<bancodedados>>** pode ser usado para identificar um banco de dados que serve a uma ou mais classes de projeto ou ao sistema todo; **<<tabela>>** pode ser usada para identificar uma tabela em um banco de dados.

Interfaces. Interfaces fornecem informação importante sobre comunicação e colaboração (bem como, para nos ajudar a seguir o OCP). No entanto, a representação livre de interfaces tende a complicar os diagramas de componentes. Ambler [AMB02] recomenda que (1) a representação pirulito de uma interface deve ser usada no lugar da abordagem UML mais formal de caixas e setas tracejadas, quando aumenta a complexidade dos diagramas; (2) por consistência, interfaces devem fluir da esquerda para a direita da caixa do componente; (3) somente aquelas interfaces relevantes para determinado componente devem ser mostradas, mesmo que outras estejam disponíveis. Essas recomendações são planejadas para simplificar a natureza visual de diagramas de componentes UML.

Dependências e herança. Para melhorar a legibilidade, é aconselhável modelar dependências da esquerda para a direita, e herança de baixo (classes derivadas) para cima (classes-base). Além disso, interdependências de componentes devem ser representadas via interfaces em vez de pela representação de dependência de componente a componente. Seguindo a filosofia do OCP, isso ajudará a tornar o sistema mais manutenível.

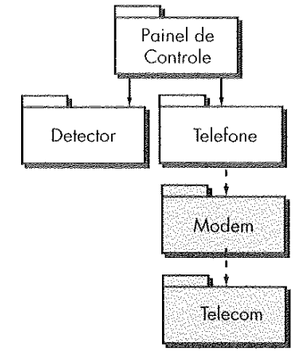
11.2.3 Coesão

No Capítulo 9, descrevemos coesão como a "exclusividade de enfoque" de um componente. No contexto de projeto no nível de componente para sistemas orientados a objetos, *coesão* implica

2 É improvável que alguém de marketing ou a organização do cliente (um tipo não técnico) venha a examinar informação de projeto detalhada.

FIGURA 11.5

Coesão em camada



que um componente ou classe encapsule somente os atributos e operações muito relacionados entre si e com a classe ou componente propriamente dito. Lethbridge e Laganiére [LET01] definem um certo número de diferentes tipos de coesão (listados em ordem do nível de coesão):[3]

AVISO

Embora um entendimento dos vários níveis de coesão seja instrutivo, é mais importante estar atento ao conceito geral quando você projeta componentes. Mantenha a coesão tão alta quanto possível.

Funcional. Exibido principalmente por operações, esse nível de coesão ocorre quando um módulo executa um e somente um cálculo, e então retorna um resultado.

Em camada. Exibido por pacotes, componentes e classes, esse tipo de coesão ocorre quando uma camada de mais alto nível tem acesso a serviços de camadas mais baixas, mas camadas mais baixas não têm acesso a camadas mais altas. Considere, por exemplo, o requisito da função de segurança do *CasaSegura* para fazer uma chamada de telefone externa se um alarme for percebido. Pode ser possível definir um conjunto de pacotes em camadas como mostra a Figura 11.5. Os pacotes sombreados contêm componentes de infra-estrutura. O acesso é do pacote de painel de controle para baixo.

Comunicacional. Todas as operações que têm acesso aos mesmos dados são definidas dentro de uma classe. Em geral, tais classes enfocam somente os dados em questão, dando acesso a eles e os armazenando.

Classes e componentes que exibem coesão funcional, em camada e comunicacional são relativamente mais fáceis de implementar, testar e manter. O projetista deve esforçar-se para alcançar esses níveis de coesão. No entanto, há muitas instâncias em que os níveis imediatamente mais baixos de coesão são encontrados:

Seqüencial. Componentes ou operações são agrupados para permitir que o primeiro forneça entrada para o próximo e assim por diante. O objetivo é implementar uma seqüência de operações.

Procedural. Componentes ou operações são agrupados para permitir que um seja invocado imediatamente depois que o anterior tiver sido invocado (acionado), mesmo quando não há passagem de dados entre eles.

Temporal. Operações executadas para refletir um comportamento ou estado específico; por exemplo, uma operação realizada no início ou todas as operações realizadas quando um erro é detectado.

Utilidade. Componentes, classes ou operações que existem dentro da mesma categoria, mas não são relacionados e estão agrupados. Por exemplo, uma classe **Estatística** exibe coesão de utilidade se ela contém todos os atributos e operações necessários para calcular seis medidas estatísticas simples.

Esses níveis de coesão são menos desejáveis e devem ser evitados quando existem alternativas de projeto. É importante notar, entretanto, que o projeto pragmático e tópicos de implementação algumas vezes forçam o projetista a optar por níveis mais baixos de coesão.

3 Em geral, quanto mais alto o nível de coesão, mais fácil implementar, testar e manter o componente.

Document shared on www.docsity.com

CASASEGURA

Coesão em Ação

A cena. Sala de Jamie.

Os personagens: Jamie e Ed — membros da equipe de engenharia de software do *CasaSegura* que estão trabalhando na função de vigilância.

A conversa:

Ed: Tenho um primeiro esboço de projeto do componente câmera.

Jamie: Quer fazer uma revisão rápida?

Ed: Suponho que sim... mas, realmente, gostaria de sua opinião sobre algo.

(Jamie gesticula para ele continuar.)

Ed: Nós originalmente definimos cinco operações para a câmera. Olhe... [Ed mostra a lista.]

determinarTipo( ) informa o tipo de câmera.

traduzirLocalização( ) permite que eu mova a câmera pela planta baixa.

exibirID( ) obtém a ID da câmera e a exibe próximo do ícone da câmera.

exibirVisão( ) mostra o campo de visão da câmera graficamente.

exibirZoom( ) mostra a amplitude da câmera graficamente.

Ed: Eu projetei cada uma separadamente e elas são operações bem simples. Então acho que pode ser uma boa idéia combinar todas as operações de exibição em somente uma que é chamada de *exibirCâmera( )* — e irá mostrar a ID, a visão e o zoom. O que você acha?

Jamie (fazendo caretas): Não acho que seja uma idéia tão boa.

Ed (carrancudo): Por quê? Todas essas operações pequenas podem dar dor de cabeça.

Jamie: O problema em combiná-las é que perdemos coesão. Você sabe, a operação *exibirCâmera( )* não vai ter exclusividade de enfoque.

Ed (levemente exasperado): E daí? A coisa toda vai ter menos do que cem linhas de código-fonte. Vai ser mais fácil de implementar, eu acho.

Jamie: E se o marketing decidir modificar o modo pelo qual nós representamos o campo de visão?

Ed: Eu iria direto para a operação *exibirCâmera( )* e faria a modificação.

Jamie: E o efeito colateral?

Ed: O que você quer dizer com isso?

Jamie: Bem, digamos que você tenha feito a modificação, mas inadvertidamente tenha criado um problema com a exibição da ID.

Ed: Eu não seria tão descuidado.

Jamie: Talvez não, mas e se alguma pessoa de manutenção daqui a dois anos tiver de fazer a modificação? Ele pode não entender a operação tão bem quanto você e, quem sabe, ele poderia ser descuidado.

Ed: Então você está contra isso?

Jamie: Você é o projetista... a decisão é sua... só tenha certeza de que você entende as conseqüências da baixa coesão.

Ed (pensando por um momento): Talvez usemos a exibição de operações separadas.

Jamie: Boa decisão.

11.2.4 Acoplamento

Em discussões anteriores de análise e projeto, observamos que comunicação e colaboração são elementos essenciais de qualquer sistema orientado a objetos. Há, no entanto, um lado negro dessa importante (e necessária) característica. À medida que a quantidade de comunicação e colaboração aumenta (à medida que o grau de "conectividade" entre classes cresce), a complexidade do sistema também aumenta. E à medida que a complexidade aumenta, a dificuldade de implementação, teste e manutenção do software também aumenta.

Acoplamento é uma medida qualitativa do grau em que as classes são conectadas entre si. À medida que as classes (e componentes) tornam-se mais interdependentes, o acoplamento cresce. Um objetivo importante em projeto no nível de componente é preservar o acoplamento tão baixo quanto possível.

Acoplamento de classes pode manifestar-se de uma variedade de modos. Lethbridge e Laganiére [LET01] definem as seguintes categorias de acoplamento:

Acoplamento por conteúdo. Ocorre quando um componente "subrepticiamente modifica dados internos a um outro componente" [LET01]. Isso viola a ocultação de informação — um conceito básico de projeto.

AVISO

À medida que o projeto de cada componente de software é elaborado, o enfoque muda para o projeto de estruturas de dados específicas e para os projetos procedurais para manipular as estruturas de dados. No entanto, não se esqueça da arquitetura que precisa alojar os componentes nas estruturas globais de dados que podem servir a vários componentes.

Acoplamento comum. Ocorre quando certo número de componentes faz uso de uma variável global. Embora isso seja algumas vezes necessário (por exemplo, para estabelecimento de valores *padrão* que são aplicáveis ao longo de toda uma aplicação), acoplamento comum pode levar à propagação descontrolada de erros e efeitos colaterais imprevisíveis quando modificações são feitas.

Acoplamento por controle. Ocorre quando *operaçãoA()* invoca *operaçãoB()* e passa um sinal de controle para B. O sinal de controle então "dirige" o fluxo lógico dentro de B. O problema com essa forma de acoplamento é que uma modificação não relacionada em B pode resultar na necessidade de modificar o significado do sinal de controle que A passa. Se isso for ignorado, resultará em erro.

Acoplamento carimbado. Ocorre quando a **ClasseB** é declarada como um tipo de argumento de uma operação da **ClasseA**. Como **ClasseB** é agora uma parte da definição da **ClasseA**, modificar o sistema torna-se mais complexo.

Acoplamento por dados. Ocorre quando operações passam longas cadeias como argumentos de dados. A "largura de banda" de comunicação entre classes e componentes aumenta e a complexidade da interface aumenta. Teste e manutenção são mais difíceis.

Acoplamento por chamada de rotina. Ocorre quando uma operação chama outra. Esse nível de acoplamento é comum e é freqüentemente necessário. No entanto, não aumenta a conectividade de um sistema.

Acoplamento por uso de tipo. Ocorre quando um componente **A** usa um tipo de dado definido em um componente **B** (isso ocorre sempre que "uma classe declara uma instância de variável ou variável local como tendo outra classe para seu tipo" [LET01]). Se a definição de tipo muda, todo componente que usa a definição deve modificar-se também.

Acoplamento por importação ou inclusão. Ocorre quando o componente **A** importa ou inclui um pacote ou o conteúdo do componente **B**.

Acoplamento externo. Ocorre quando um componente comunica ou colabora com componentes de infra-estrutura (funções de sistema operacional, facilidades de banco de dados, funções de telecomunicação). Embora esse tipo de acoplamento seja necessário, deve ser limitado a um pequeno número de componentes ou classes dentro de um sistema.

O software deve se comunicar interna e externamente. Assim, acoplamento é um fato da vida. No entanto, o projetista deve trabalhar para reduzi-lo sempre que possível e entender as ramificações do alto acoplamento quando ele não pode ser evitado.

CASASEGURA

Acoplamento em Ação

A cena: Sala de Shakira.

Os personagens: Vinod e Shakira — membros da equipe de engenharia de software do *CasaSegura* que estão trabalhando na função de segurança.

A conversa:

Shakira. Eu tive o que pensei fosse uma grande idéia... então ponderei um pouco e me pareceu uma idéia não tão boa. Por fim, a rejeitei, mas pensei que deveria mostrá-la pra você.

Vinod: Ótimo, qual é a idéia?

Shakira: Bem, cada um dos sensores reconhece uma condição de alarme de alguma espécie, certo?

Vinod (sorrindo): Essa é razão pela qual nós os chamamos de sensores, Shakira.

Shakira (exasperada): Sarcasmo, Vinod. Você deve trabalhar suas habilidades interpessoais.

Vinod: Você estava dizendo?

Shakira: Está bem, de qualquer modo, eu pensei: por que não criar uma operação dentro de cada objeto sensor denominada *fazerChamada( )* que colaboraria diretamente com o componente **ChamadaExterna**, bem, com uma interface para o componente **ChamadaExterna?**

Vinod (pensativo): Você quer dizer que em vez de ter essa colaboração ocorrendo de um componente como **PainelDeControle** ou algo desse tipo?

Shakira: É... mas, então eu disse a mim mesma, isso significa que todo objeto sensor estará conectado ao componente **ChamadaExterna**, e isso significa que ele está indiretamente acoplado ao mundo exterior e... bem, pensei que só tornei as coisas mais complicadas.

Vinod: Concordo. Neste caso, uma idéia melhor é deixar a interface de sensor passar informação para o **PainelDeControle** e deixar que ele inicie a chamada

Document shared on www.docsity.com

externa. Além disso, diferentes sensores poderiam resultar em diferentes números telefônicos. Você não quer que o sensor armazene aquela informação porque se ela muda...

Shakira: Não me pareceu certo.

Vinod: Heurísticas de projeto para acoplamento nos dizem que não está certo.

Shakira: Tudo bem...

11.3 CONDUÇÃO DO PROJETO NO NÍVEL DE COMPONENTE

Anteriormente, neste capítulo, observamos que o projeto no nível de componente é elaborativo por natureza. O projetista deve transformar informação dos modelos de análise e arquitetural em uma representação de projeto que forneça detalhe suficiente para guiar a atividade de construção (codificação e teste). Os passos seguintes representam um conjunto de tarefas típico para o projeto no nível de componente, quando é aplicado a um sistema orientado a objetos.

Passo 1. Identificar todas as classes de projeto que correspondam ao domínio do problema. Usando os modelos de análise e arquitetural, cada classe de análise e componente arquitetural é elaborada como descrito na Seção 11.1.1.

AVISO

Se você está trabalhando em um ambiente não OO, os primeiros dos três passos enfocam o refinamento dos objetos de dados e funções de processamento (transformações) identificados como parte do modelo de análise.

Passo 2. Identificar todas as classes de projeto que correspondam ao domínio de infra-estrutura. Essas classes não são descritas no modelo de análise e são freqüentemente omitidas do modelo de arquitetura, mas elas devem ser descritas neste ponto. Como já observamos anteriormente, classes e componentes dessa categoria incluem componentes IGU, componentes de sistema operacional, componentes de gestão de objetos e dados e outros.

Passo 3. Elaborar todas as classes de projeto que não são adquiridas como componentes reusáveis. Elaboração requer que todas as interfaces, atributos e operações necessárias para implementar a classe sejam descritas em detalhe. Heurísticas de projeto (por exemplo, coesão e acoplamento de componentes) devem ser consideradas quando essa tarefa é conduzida.

Passo 3a. Especificar detalhes de mensagens quando classes ou componentes colaboram. O modelo de análise faz uso de um diagrama de colaboração para mostrar como as classes de análise colaboram umas com as outras. À medida que o projeto no nível de componente prossegue, é algumas vezes útil mostrar os detalhes dessas colaborações especificando a estrutura de mensagens passadas entre os objetos de um sistema. Embora essa atividade de projeto seja opcional, ela pode ser usada como uma precursora da especificação de interface que mostra como componentes de um sistema comunicam-se e colaboram.

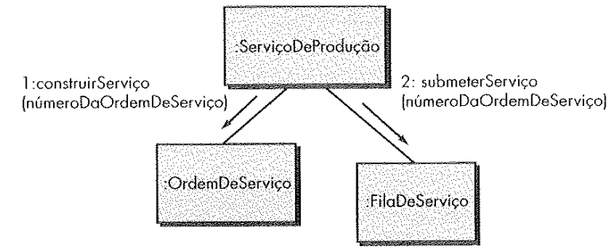
A Figura 11.6 ilustra um diagrama de colaboração simples para o sistema de gráfica discutido anteriormente. Três objetos, **ServiçoDeProdução**, **OrdemDeServiço** e **FilaDeServiço**, colaboram a fim de preparar um serviço de impressão para submissão ao fluxo de produção. Mensagens são passadas entre objetos como é ilustrado pelas setas da figura. Durante a modelagem de análise, as mensagens são especificadas como mostra a figura. Entretanto, à medida que o projeto prossegue, cada mensagem é elaborada expandindo sua sintaxe do seguinte modo [BEN02]:

[condição de guarda] expressão de seqüência (valor de retorno): =
nome da mensagem (lista de argumentos)

em que uma **[condição de guarda]** é escrita em Linguagem de Restrição de Objeto (*Object Constraint Language* — OCL)[4] e especifica qualquer conjunto de condições que devem ser satisfeitas antes que a mensagem possa ser enviada; **expressão de seqüência** é um valor inteiro (ou outro indicador de ordenação, por exemplo, 3.1.2) que indica a ordem seqüencial em que a mensagem é enviada; **(valor de retorno)** é o nome da informação retornada pela operação invocada pela mensagem; **nome da mensagem** identifica a operação que deve ser invocada e **(lista de argumentos)** é a lista de atributos passada para a operação.

4 OCL será discutida brevemente na Seção 11.4 e no Capítulo 28.

FIGURA 11.6 Diagrama de colaboração com mensagem



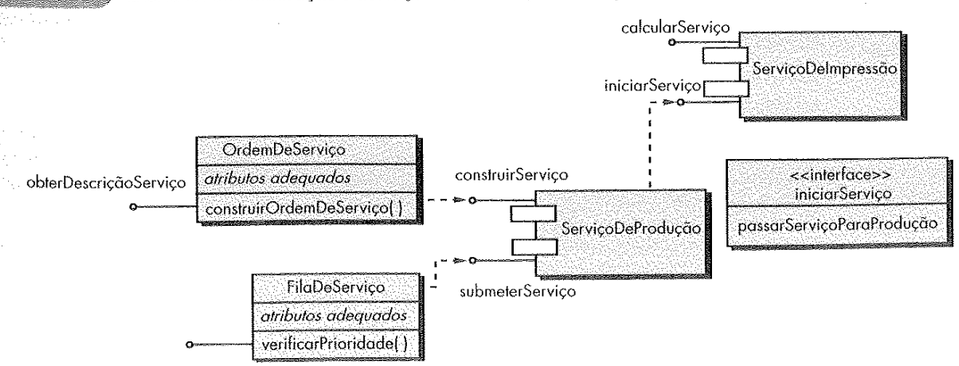
Passo 3b. Identificar interfaces adequadas para cada componente. No contexto de projeto no nível de componente, uma interface UML é "um grupo de operações de visibilidade externa (públicas). A interface não contém estrutura interna, não tem atributos, nem associações..." [BEB02]. Dito de modo mais formal, uma interface é o equivalente a uma classe abstrata que fornece conexão controlada entre classes de projeto. A elaboração de interfaces é apresentada na Figura 11.1. Na essência, operações definidas para as classes de projeto são categorizadas em uma ou mais classes abstratas. Toda operação em uma classe abstrata (a interface) deve ser coesiva; isto é, deve exibir processamento que enfoca uma função ou subfunção limitada.

Referindo-se à Figura 11.1, pode ser alegado que a interface *inícioDeServiço* não exibe coesão suficiente. Na realidade, ela executa três diferentes subfunções: construir uma ordem de serviço, verificar prioridade do serviço e passar o serviço para a produção. O projeto de interface deve ser refabricado. Uma abordagem poderia ser para reexaminar as classes de projeto e definir uma nova classe **OrdemDeServiço** que cuidaria de todas as atividades associadas com a montagem de uma ordem de serviço. A operação *constuirOrdemDeServiço()* torna-se uma parte dessa classe. Analogamente, podemos definir uma classe **FilaDeServiço** que incorporaria a operação *verificarPrioridade()*. Uma classe **ServiçoDeProdução** deveria englobar toda a informação associada com um serviço de produção a ser passado para a facilidade de produção. A interface *inícioDeServiço* deveria então tomar a forma mostrada na Figura 11.7. A interface *inícioDeServiço* é agora coesiva, enfocada em uma função. As interfaces associadas com **ServiçoDeProdução, OrdemDeServiço** e **FilaDeServiço** têm similarmente exclusividade de enfoque.

Passo 3c. Elaborar atributos e definir tipos de dados e estruturas de dados necessários para implementá-los. Em geral, estruturas e tipos de dados usados para descrever atributos são definidos dentro do contexto da linguagem de programação que será usada para a implementação. UML define um tipo de dados de atributo por meio da seguinte sintaxe:

nome : tipo-expressão = valor inicial {cadeia de propriedade}

FIGURA 11.7 Interfaces de refatoração e definições de classe para ServiçoDeImpressão



Document shared on www.docsity.com

em que **nome** é o nome do atributo e **tipo-expressão** é o tipo de dado; **valor inicial** é o valor que o atributo toma quando um objeto é criado e **cadeia de propriedade** define uma propriedade ou característica do atributo.

Durante a primeira iteração do projeto no nível de componente, atributos são normalmente descritos por nome. Referindo-se outra vez à Figura 11.1, a lista de atributos para **ServiçoDeImpressão** lista apenas os nomes dos atributos. Entretanto, à medida que a elaboração do projeto prossegue, cada atributo é definido pelo formato mencionado de atributo UML. Por exemplo, **peso-TipoDopapel** é definido da seguinte maneira:

peso-TipoDopapel: string = "A" {contém 1 de 4 valores — A, B, C ou D}

que define **peso-TipoDopapel** como uma variável string que recebe o valor inicial A que pode tomar um dos quatro valores do conjunto {A,B,C,D}.

Se um atributo aparece repetidamente ao longo de um certo número de classes de projeto e ele tem uma estrutura relativamente complexa, é melhor criar uma classe separada para acomodar o atributo.

Passo 3d. Descrever fluxo de processamento em cada operação em detalhe. Isso pode ser conseguido usando uma linguagem de programação baseada em pseudocódigo (veja a Seção 11.5.5) ou com um diagrama de atividade UML. Cada componente de software é elaborado por meio de certo número de iterações que aplicam o conceito de refinamento passo a passo (veja o Capítulo 9).

A primeira iteração define cada operação como parte da classe de projeto. Em cada caso, a operação deve ser caracterizada de modo que assegure alta coesão; a operação deve realizar uma única função ou subfunção-alvo. A iteração seguinte faz pouco mais do que expandir o nome da operação. Por exemplo, a operação *calcularCustoPapel()* observada na Figura 11.1 pode ser expandida do seguinte modo:

calcularCustoPapel (peso, tamanho, cor): numérico

isso indica que *calcularCustoPapel()* requer os atributos **peso**, **tamanho** e **cor** como entrada e retorna um valor que é numérico (na verdade, um valor em dólar) como saída.

> "Se eu tivesse mais tempo, teria escrito uma carta mais curta."
>
> **Blaise Pascal**

AVISO

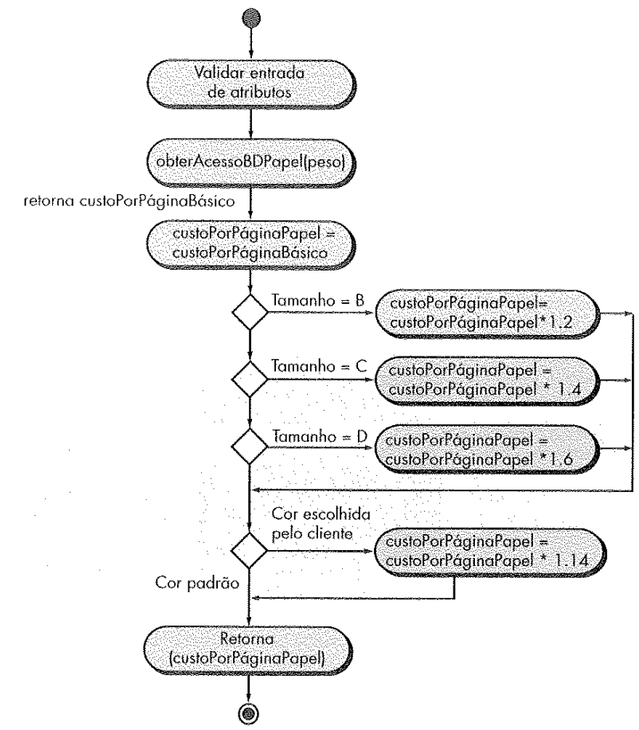
Use elaboração passo a passo quando refinar um projeto de componente. Sempre pergunte: "Há um modo pelo qual isso possa ser simplificado e ainda conseguir o mesmo resultado?".

Se o algoritmo necessário para implementar *calcularCustoPapel()* é simples e totalmente compreendido, nenhuma elaboração adicional de projeto precisa ser necessária. O engenheiro de software que faz a codificação vai providenciar o detalhe necessário para implementar a operação. No entanto, se o algoritmo é mais complexo ou enigmático, mais elaboração de projeto é necessária nesse estágio. A Figura 11.8 representa um diagrama de atividade UML para *calcularCustoPapel()*. Quando diagramas de atividade são usados para especificação de projeto no nível de componente, eles são geralmente representados em um nível de abstração um pouco mais alto do que código-fonte. Uma abordagem alternativa — o uso de pseudocódigo para especificação de projeto — é discutida mais adiante neste capítulo.

Passo 4. Descrever fontes de dados persistentes (bancos de dados e arquivos) e identificar as classes necessárias para geri-los. Bancos de dados e arquivos normalmente transcendem a descrição de projeto de um componente individual. Na maioria dos casos, esses depósitos de dados persistentes são inicialmente especificados como parte do projeto arquitetural. No entanto, à medida que a elaboração do projeto prossegue, é freqüentemente útil fornecer detalhe adicional sobre a estrutura e organização dessas fontes de dados persistentes.

Passo 5. Desenvolver e elaborar representações comportamentais para uma classe ou componente. Diagramas de estado UML foram usados como parte do modelo de análise para representar o comportamento externamente observável do sistema e o comportamento mais localizado das classes de análise individuais. Durante o projeto no nível de componente, é algumas vezes necessário modelar o comportamento de uma classe de projeto.

FIGURA 11.8 Diagrama de atividade de *calcularCustoDePapel()*



O comportamento dinâmico de um objeto (uma instanciação de uma classe de projeto quando o programa é executado) é afetado por eventos externos a ele e pelo estado atual (modo de comportamento) do objeto. Para entender o comportamento dinâmico de um objeto, o projetista deve examinar todos os casos de uso relevantes para a classe de projeto durante a sua vida. Esses fornecem informação que ajuda o projetista a delinear os eventos que afetam o objeto e os estados pelos quais o objeto passa à medida que o tempo transcorre e eventos acontecem. As transições entre estados (dirigidas por eventos) são representadas pelo diagrama de estados (*statechart*) UML [BEN02] como mostra a Figura 11.9.

A transição de um estado (representado por um retângulo com cantos arredondados) para outro ocorre como conseqüência de um evento que toma a forma:

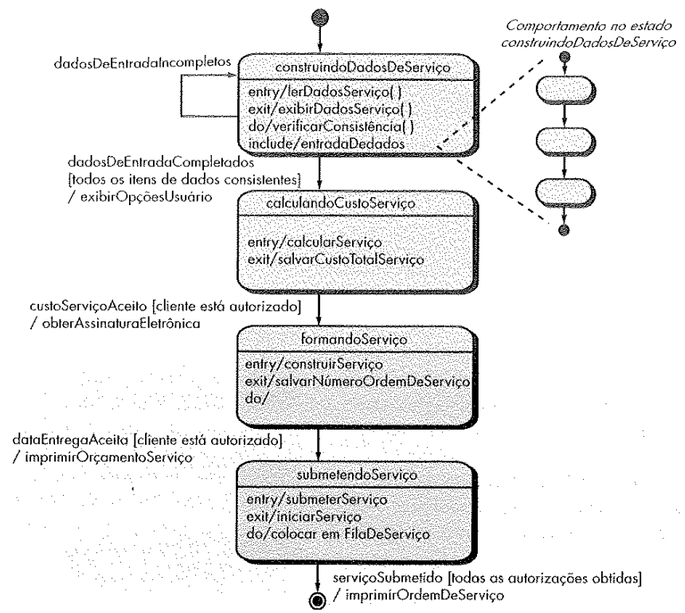
nome-evento (lista-parâmetros) [condição-de-guarda] / expressão de ação

em que **nome-evento** identifica o evento; **lista-parâmetros** incorpora dados associados ao evento; **condição-de-guarda** é escrita em Linguagem de Restrição de Objetos (*Object Constraint Language* — OCL) e especifica uma condição que deve ser satisfeita antes que o evento possa ocorrer, e **expressão de ação** define uma ação que ocorre quando a transição tem lugar.

De acordo com a Figura 11.9, cada estado pode definir ações *entry/* e *exit/* que ocorrem à medida que transições para e do estado ocorrem. Na maioria dos casos, essas ações correspondem a operações relevantes à classe que está sendo modelada. O indicador *do/* fornece um mecanismo para indicar atividades que ocorrem enquanto no estado e o indicador *include/* fornece um meio para elaborar o comportamento embutindo mais detalhe na definição de estado do diagrama de estado.

Document shared on www.docsity.com

FIGURA 11.9 Fragmento do diagrama de estado para a classe ServiçoDeImpressão



É importante notar que o modelo comportamental freqüentemente contém informação que não é óbvia em outros modelos de projeto. Por exemplo, um exame cuidadoso do diagrama de estado da Figura 11.9 indica que o comportamento dinâmico da classe **ServiçoDeImpressão** depende de duas aprovações pelo cliente à medida que os dados de custos e cronograma para o serviço de impressão são derivados. Sem aprovações (a condição de guarda garante que o cliente é autorizado para aprovar) o serviço de impressão não pode ser submetido, porque não há um modo de atingir o estado *submetendoServiço*.

Passo 6. Elaborar diagramas de implantação para fornecer detalhe adicional de implementação. Diagramas de implantação (veja o Capítulo 9) são usados como parte do projeto arquitetural e representados na forma de descritor. Nessa forma, as funções principais do sistema (freqüentemente representadas como subsistemas) são representadas no contexto do ambiente computacional que vai alojá-las.

Durante o projeto no nível de componente, diagramas de implantação podem ser elaborados para representar a localização de pacotes-chave de componentes. No entanto, componentes em geral não são representados individualmente em um diagrama de componentes. A razão para isso é evitar a complexidade diagramática. Em alguns casos, diagramas de implantação são elaborados na forma de instâncias nesse momento. Isso significa que o(s) ambiente(s) específico(s) de hardware e sistema operacional a ser usado(s) é (são) especificado(s) e a localização dos pacotes de componentes nesse ambiente é indicada.

Passo 7. Fabricar toda a representação do projeto no nível de componente e sempre considerar alternativas. Ao longo deste livro, temos enfatizado que projeto é um processo iterativo. O primeiro modelo no nível de componente que você cria não é tão completo, consistente ou preciso quanto na enésima iteração que aplicar ao modelo. É essencial refabricar à medida que o trabalho de projeto é conduzido.

Além disso, um projetista não deve ter visão restrita. Há sempre soluções alternativas de projeto, e os melhores projetistas consideram todas (ou a maioria) delas antes de se decidir pelo modelo final de projeto. Desenvolva alternativas e considere cada uma cuidadosamente, usando os princípios e conceitos de projeto apresentados nos Capítulos 5, 9 e neste capítulo.

11.4 LINGUAGEM DE RESTRIÇÃO DE OBJETO

PONTO CHAVE

OCL fornece uma gramática e sintaxe formal para descrever elementos de projeto no nível de componentes.

A grande variedade de diagramas disponíveis como parte da UML fornece ao projetista um rico conjunto de formas representacionais para o modelo de projeto. No entanto, representações gráficas são freqüentemente insuficientes. O projetista precisa de um mecanismo para representar explícita e formalmente informação que restrinja algum elemento do modelo de projeto. É possível, naturalmente, descrever restrições em linguagem natural como português, mas essa abordagem invariavelmente leva à inconsistência e ambigüidade. Por essa razão, uma linguagem mais formal — que se apóie em teoria dos conjuntos e linguagens de especificação formal (veja o Capítulo 28), mas tenha a sintaxe um tanto menos matemática que uma linguagem de programação — parece apropriada.

A *Linguagem de Restrição de Objetos* (OCL) complementa a UML permitindo ao engenheiro de software usar uma gramática e sintaxe formais para construir declarações não ambíguas sobre vários elementos do modelo de projeto (classes e objetos, eventos, mensagens, interfaces). As declarações mais simples da linguagem OCL são construídas em quatro partes: (1) um *contexto* que define a situação limitada na qual a declaração é válida; (2) uma *propriedade* que representa algumas características do contexto (se o contexto é uma classe, uma propriedade pode ser um atributo); (3) uma *operação* (por exemplo, aritmética, orientada a conjunto) que manipula ou qualifica uma propriedade; (4) *palavras-chave* (**se, então, senão, e, ou, não, implica**) que são usadas para especificar expressões condicionais.

Como um simples exemplo de expressão OCL, considere a condição de guarda colocada no evento *custoServicoAceito* que causa uma transição entre os estados **calculandoCustoServiço** e **formandoServiço** no diagrama de estados para a classe **ServiçoDeImpressão** (veja a Figura 11.9). No diagrama, a condição de guarda é expressa em linguagem natural e implica que a autorização somente pode ocorrer se o cliente está autorizado a aprovar o custo do serviço. Em OCL, a expressão pode tomar a forma:

```
cliente
    self. autoridadeAutorização = 'sim'
```

em que um atributo booleano, **autoridadeAutorização**, da classe **Cliente** (na verdade uma instância específica da classe) deve ser fixado com **sim** para a condição de guarda ser satisfeita.

Veja na Web

A especificação OCL completa pode ser encontrada em www.omg.org.

À medida que o modelo de projeto é criado, há freqüentemente instâncias (veja a Seção 11.2.1) em que pré ou pós-condições devem ser satisfeitas antes de completar alguma ação especificada pelo projeto. OCL fornece uma ferramenta poderosa para especificar pré e pós-condições de modo formal. Como exemplo, considere uma extensão do sistema de gráfica (discutido ao longo deste capítulo) em que o cliente fornece um limite superior de custo para o serviço de impressão e uma data de entrega "inadiável", ao mesmo tempo em que outras características do serviço de impressão são especificadas. Se as estimativas de custo e de entrega excedem esses limites, o serviço não é submetido e o cliente deve ser notificado. Em OCL um conjunto de pré e pós-condições pode ser especificado da seguinte maneira:

```
Context ServiçoDeImpressão::validar (limiteSuperiorDeCusto : Integer, requisitosDeEntregaCliente :
    Integer)
    pre: limiteSuperiorDeCusto > 0
      and requisitosDeEntregaCliente > 0
      and self.autorizaçãoServiço = 'não'
    pos: if self.custoTotalDoServiço < = limiteSuperiorDeCusto
            and self.dataDeEntrega < = requisitosDeEntregaCliente
      then
            self.autorizaçãoServiço = 'sim'
      enfif
```

Document shared on www.docsity.com

Essa declaração OCL define uma *invariante* — condições que precisam existir antes de **(pre)** e depois de **(pos)** algum comportamento. Inicialmente, uma precondição estabelece que limitantes de custo e data de entrega devem ser especificados pelo cliente, e a autorização deve ser fixada em 'não'. Depois que os custos e a entrega são determinados, a pós-condição é aplicada. Deve também ser notado que a expressão **self.autorizaçãoServiço = 'sim'** não está associando ao valor "sim", mas está declarando que a **autorizaçãoServiço** deve ter sido fixada em "sim" quando a operação termina.

Uma descrição completa de OCL está além do escopo deste livro.[5] Leitores interessados devem ver [WAR98] e [OMG01] para informação adicional.

FERRAMENTAS DE SOFTWARE

UML/OCL

Objetivo: uma grande variedade de ferramentas UML está disponível para apoiar o projetista em todos os níveis de projeto. Algumas delas fornecem suporte OCL.

Mecânica: Ferramentas dessa categoria habilitam um projetista a criar todos os diagramas UML que são necessários para construir um modelo de projeto completo. Mais importante, muitas ferramentas fornecem sólida verificação sintática e semântica e gestão de controle de versão e de modificação (veja o Capítulo 27). Quando a habilidade OCL é fornecida, ferramentas possibilitam ao projetista criar expressões OCL e, em alguns casos, "compilá-las" para vários tipos de avaliação e análise.

Ferramentas Representativas[6]

ArgoUML, distribuída por Tigress.org (http://argouml.tigris.org/), suporta UML completa e OCL e inclui uma variedade de ferramentas de apoio a projetos que vai além da geração de diagramas UML e expressões OCL.

Dresden OCL toolkit, desenvolvida por Frank Finger da Universidade de Tecnologia de Dresden (http://dresden-ocl.sourceforge.net/), é um conjunto de ferramentas baseado em um compilador OCL que abrange vários módulos, os quais interpretam, verificam tipo e normalizam as restrições OCL.

OCL parser, desenvolvido por IBM (http://www-3.ibm.com/software/ad/library/standards/ocl-download.html), é escrita em Java e está disponível livremente para a comunidade orientada a objetos, para encorajar o uso de OCL com modeladores UML.

11.5 PROJETO DE COMPONENTES CONVENCIONAIS

Os fundamentos do projeto no nível de componentes para componentes convencionais de software[7] foram formados no início dos anos de 1960 e solidificados com o trabalho de Edsgar Dijkstra e seus colegas ([BOH66], [DIJ65], [DIJ76]). No fim dos anos de 1960, Dijkstra e outros propuseram o uso de um conjunto de construções lógicas restritas por meio das quais qualquer programa poderia ser construído. As construções enfatizam "a manutenção do domínio funcional". Cada construção tem uma estrutura lógica previsível, entra-se nela pelo topo e sai-se pela parte inferior, permitindo ao leitor seguir o fluxo procedural mais facilmente.

As construções são seqüência, condição e repetição. *Seqüência* implementa passos de processamento que são essenciais na especificação de qualquer algoritmo. *Condição* fornece facilidades para processamento seletivo baseado em alguma ocorrência lógica, e *repetição* provê ciclos. Essas

5 No entanto, mais discussão sobre OCL (apresentada no contexto de métodos formais) é apresentada no Capítulo 28.

6 Os produtos mencionados aqui não representam uma recomendação, mas, em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

7 Um componente convencional de software implementa um elemento de processamento que trata funções ou subfunções no domínio do problema ou alguma capacidade no domínio de infra-estrutura. Freqüentemente chamados de *módulos*, *procedimentos* ou *subrotinas*, componentes convencionais não encapsulam dados da maneira que os componentes OO fazem.

PONTO CHAVE

Programação estruturada é uma técnica de projeto que restringe a lógica do fluxo a três construções: seqüência, condição e repetição.

três construções são fundamentais para a *programação estruturada* — uma importante técnica de projeto no nível de componentes.

As construções estruturadas foram propostas para limitar o projeto procedural do software a um pequeno número de operações previsíveis. Métricas de complexidade (veja o Capítulo 15) indicam que o uso de construções estruturadas reduz a complexidade do programa e conseqüentemente melhora a legibilidade, testabilidade e manutenibilidade. O uso de um número limitado de construções lógicas também contribui para um processo de entendimento humano, o que os psicólogos chamam de *fatiamento*. Para entender esse processo, considere o modo pelo qual você está lendo esta página. Você não lê letras individuais, e sim reconhece padrões ou reuniões de letras que formam palavras ou frases. As construções estruturadas são reuniões lógicas que permitem ao leitor reconhecer os elementos procedurais de um módulo, em vez de ler o projeto ou código linha a linha. O entendimento é melhorado quando são encontrados padrões lógicos prontamente reconhecíveis.

11.5.1 Notação Gráfica de Projeto

Discutimos anteriormente o diagrama de atividade UML neste capítulo e nos Capítulos 7 e 8. O diagrama de atividade permite a um projetista representar seqüência, condição e repetição — todos os elementos da programação estruturada — e é o descendente de uma antiga representação pictórica de projeto (ainda usada amplamente) chamada de *fluxograma*.

Um fluxograma, como um diagrama de atividade, é muito simples pictoricamente. Uma caixa é usada para indicar um passo de processamento. Um losango representa uma condição lógica e setas mostram o fluxo de controle. A Figura 11.10 ilustra as três construções estruturadas. A *seqüência* é representada como duas caixas de processamento conectadas por uma linha (seta) de controle. A *condição*, também chamada de *se-então-senão*, é mostrada como um losango de decisão que, se verdadeiro, provoca a ocorrência da parte de processamento *então*, e se falso, invoca a parte de processamento *senão*. *Repetição* é representada de duas formas ligeiramente diferentes. O *faça-enquanto* testa uma condição e executa um ciclo de tarefa repetidamente, enquanto a condição permanece verdadeira. Um *repita-até* executa o ciclo de tarefas primeiro, depois testa uma condição e repete a tarefa até que a condição falhe. A construção de *seleção* (ou *seleção de caso*) mostrada na figura é na verdade uma extensão do *se-então-senão*. Um parâmetro é testado por decisões sucessivas até que uma condição verdadeira ocorra e uma *parte* de processamento correspondente a um caminho do caso seja executada.

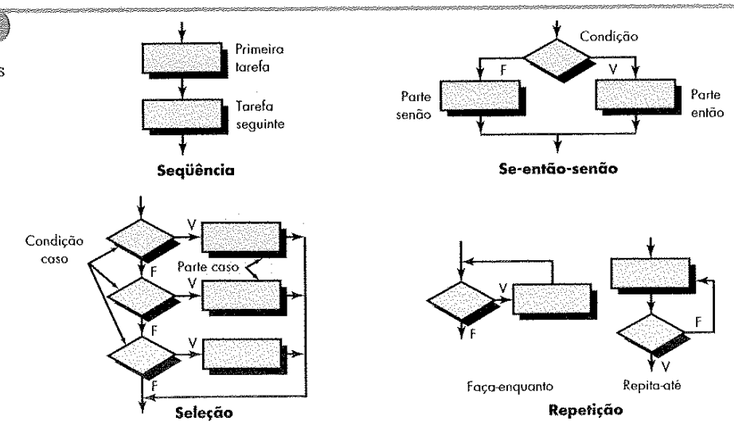
AVISO

Construções de programação estruturada devem tornar mais fácil o entendimento do projeto. Se seu uso sem "violação" resultar em complexidade desnecessária, é aceitável violar.

Em geral, o uso dogmático apenas das construções estruturadas pode introduzir ineficiência quando o escape de um conjunto de ciclos aninhados ou condições aninhadas é necessário. Mais importante, uma complicação adicional de todos os testes lógicos, ao longo do caminho de escape, pode obscurecer o fluxo de controle do software, aumentar a possibilidade de erros e ter um impacto negativo na legibilidade e manutenibilidade. O que podemos fazer?

FIGURA 11.10

Fluxograma das construções



Document shared on www.docsity.com

O projetista é deixado com duas opções: (1) a representação procedural é reprojetada de modo que o "ramo de escape" não seja necessário em uma posição aninhada do fluxo de controle, ou (2) as construções estruturadas são violadas de modo controlado; isto é, é projetado um ramo restrito de saída do fluxo aninhado. A opção 1 é obviamente a abordagem ideal, mas a opção 2 pode ser acomodada sem violar o espírito da programação estruturada.

11.5.2 Notação Tabular de Projeto

AVISO

Use uma tabela de decisão quando um conjunto complexo de condições e ações for encontrado em um componente.

Em muitas aplicações de software, um módulo pode ter de avaliar uma combinação complexa de condições e selecionar ações apropriadas com base nessas condições. *Tabelas de decisão* [HUR83] fornecem uma notação que traduz ações e condições (descritas em uma narrativa de processamento) em uma forma tabular. A tabela é difícil de interpretar erroneamente e pode até ser usada como entrada legível por máquina para um algoritmo guiado por tabelas.

Uma tabela de decisão é dividida em quatro quadrantes. O quadrante superior esquerdo contém uma lista de todas as condições. O quadrante inferior esquerdo contém uma lista de todas as ações possíveis, baseadas em combinações de condições. Os quadrantes à direita formam uma matriz que indica as combinações de condições e as ações correspondentes que vão ocorrer para uma combinação específica. Assim, cada coluna da matriz pode ser interpretada como uma *regra de processamento*. Os seguintes passos são aplicados para desenvolver uma tabela de decisão:

? Como construo uma tabela de decisão?

1. Liste todas as ações que podem ser associadas a um procedimento (ou módulo) específico.
2. Liste todas as condições (ou decisões tomadas) durante a execução do procedimento.
3. Associe conjuntos específicos de condições com ações específicas, eliminando combinações impossíveis de condições; como alternativa, desenvolva todas as possíveis permutações de condições.
4. Defina as regras indicando que ação(ões) ocorre(m) para um conjunto de condições.

Para ilustrar o uso de uma tabela de decisão, considere o seguinte extrato de um caso de uso informal que foi proposto para o sistema de gráfica:

> Três tipos de clientes são definidos: cliente normal, cliente prata e cliente ouro (esses tipos são atribuídos de acordo com o volume de negócios realizado com a gráfica durante um período de 12 meses). Um cliente normal recebe taxas de impressão e entrega normais. Um cliente prata tem 8% de desconto em todas as cotações e é colocado na frente de todos os clientes normais na fila de serviço. Um cliente ouro ganha 15% de redução no preço cotado e é colocado na frente dos clientes normal e prata na fila de serviço. Um desconto especial de x% além dos outros descontos pode ser aplicado para qualquer cotação do cliente a critério da gerência.

A Figura 11.11 mostra uma representação em tabela de decisão do caso de uso informal precedente. Cada uma das seis regras indica uma das seis condições viáveis. Como regra geral, a tabela de decisão pode ser usada efetivamente para suplementar outra notação de projeto procedural.

FIGURA 11.11

Tabela de decisão resultante

	Regras					
Condições	**1**	**2**	**3**	**4**	**5**	**6**
Cliente normal	T	T				
Cliente prata			T	T		
Cliente ouro					T	T
Desconto especial	F	T	F	T	F	T
Ações						
Sem desconto	✓					
Aplicar 8% de desconto			✓	✓		
Aplicar 15% de desconto					✓	✓
Aplicar x% adicional de desconto		✓		✓		✓

11.5.3 Linguagem de Projeto de Programas

Linguagem de projeto de programas (*Program Design Language* — PDL), também chamada de *português estruturado* ou *pseudocódigo* é "um jargão no sentido do uso do vocabulário de uma língua (por exemplo, português) e a sintaxe geral de outra (uma linguagem de programação estruturada)" [CAI75]. Neste capítulo, PDL é usada como referência genérica a uma linguagem de projeto.

À primeira vista, a PDL pode parecer uma linguagem de programação. A diferença entre PDL e uma linguagem real de programação está no uso de texto narrativo (por exemplo, português) embutido diretamente nas declarações PDL. Em razão do uso de texto narrativo embutido diretamente em uma estrutura sintática, a PDL não pode ser compilada. No entanto, ferramentas PDL já existem para traduzir PDL "esqueleto" em linguagem de programação e/ou numa representação gráfica (por exemplo, fluxograma) de projeto. Essas ferramentas também produzem mapeamentos aninhados, um índice de operações de projeto, tabelas de referência cruzada e uma variedade de outras informações.

Uma linguagem de projeto de programa pode ser uma simples transposição de linguagem como Ada, C ou Java. Uma sintaxe básica de PDL deve incluir construções para definição de componentes, descrição de interface, declaração de dados, estruturação de blocos, construções de condição, construções de repetição e construções de E/S. Deve-se notar que a PDL pode ser estendida para incluir palavras-chave para processamento multitarefa e/ou concorrente, manipulação de interrupções, sincronização entre processos e muitas outras características. O projeto da aplicação para o qual a PDL deve ser usada deve determinar a forma final da linguagem de projeto. O formato e a semântica para algumas dessas construções PDL são apresentados no exemplo que se segue.

AVISO

É uma boa idéia usar sua linguagem de programação como base da PDL. Você pode então gerar um esqueleto de código misturado com texto narrativo durante o desenvolvimento do projeto.

Para ilustrar o uso de PDL, apresentamos um exemplo de projeto procedural para a função de segurança do *CasaSegura*, discutida nos capítulos anteriores. O sistema monitora alarmes de fogo, fumaça, roubo, água e temperatura (por exemplo, a caldeira quebra enquanto o proprietário está fora durante o inverno), produzindo o disparo de um alarme e a comunicação com um serviço de monitoração, gerando uma mensagem de voz sintetizada. Na PDL que se segue, ilustramos algumas das construções importantes mencionadas nas seções anteriores.

Lembre-se de que PDL *não* é uma linguagem de programação. O projetista pode adaptá-la conforme necessário, sem medo de erro de sintaxe. No entanto, o projeto do software de monitoração teria de ser revisado (você vê algum problema?) e refinado adicionalmente antes que o código possa ser escrito. A PDL[8] apresentada a seguir fornece uma elaboração do projeto procedural para uma versão anterior do componente gerir alarme.

```
componente gerirAlarme:
O objetivo deste componente é gerir as chaves do painel de controle e as entradas dos sensores por
tipo, e para atuar em qualquer condição de alarme que for encontrada.
    fixar valores default para estadoDoSistema (valor retornado), todos os itens de dados
    iniciar todas as portas do sistema e refixar todo o hardware
    verificar chavesDoPainelDeControle(cpc)
        se cpc = "teste", então chamar fixar alarme em "ligado"
        se cpc = "alarmeDesligado", então chamar fixar alarme em "desligado"
        .
        .
        .
        default para cpc = nada
    refixar todos os valoresDeSinais e chaves
    fazer para todos os sensores
        chamar procedimento verificarSensor retornando valorDoSinal
        se valorDoSinal > limite [tipoDeAlarme]
            então mensagem.telefone = mensagem [tipoDeAlarme]
```

8 O nível de detalhe representado pela PDL é definido localmente. Algumas pessoas preferem uma descrição orientada mais para linguagem natural, enquanto outras preferem algo que seja mais próximo do código.

Document shared on www.docsity.com

```
                fixar campainhaDoAlarme em "ligado" para tempoAlarmeEmSegundos
                    fixar estado sistema = "condiçãoAlarme"
                    começopar
                        chamar procedimento alarme em "ligado", tempoAlarmeEmSegundos;
                        chamar procedimento fixar telefone para tipoDeAlarme, númeroDoTelefone
                    fimpar
            senão pule
        fim se----------
    fim fazer para
fim gerirAlarme
```

Note que o projetista do componente gerir alarme usou uma nova construção **começopar... fimpar** que especifica um bloco paralelo. Todas as tarefas especificadas dentro do bloco **começopar** são executadas em paralelo. Neste caso, os detalhes de implementação não são considerados.

FERRAMENTAS DE SOFTWARE

Linguagem de Projeto de Programa

Objetivo: Apesar da grande maioria dos engenheiros de software que usam PDL ou pseudocódigo desenvolver uma versão adaptada da linguagem de programação que eles pretendem usar para implementação, algumas ferramentas PDL existem efetivamente.

Mecânica: Em alguns casos, as ferramentas fazem a engenharia reversa de código-fonte existente (uma triste realidade em um mundo em que alguns programas não têm absolutamente nenhuma documentação). Outras permitem a um projetista criar PDL com apoio automático.

Ferramentas Representativas[9]

PDL/81, desenvolvida por Caine, Farber e Gordon (http://www.cfg.com/pdl81/lpd.html), suporta a criação de projetos usando uma versão definida de PDL.

DocGen, distribuída por Software Improvement Group (http://www.software-improvers.com/DocGen.htm), é uma ferramenta de engenharia reversa que gera documentação tipo PDL para códigos Ada e C.

PowerPDL, desenvolvida por Iconix (http://www.iconixsw.com/SpecSheets/PowerPDL.html), permite a um projetista criar projetos basedos em PDL e depois traduzir o pseudocódigo em formas que podem gerar outras representações de projeto.

11.5.4 Comparação da Notação de Projeto

A notação de projeto deve levar a uma representação procedural fácil de entender e de revisar. Além disso, a notação deve melhorar a habilidade de "codificar", de modo que a codificação fique de fato um subproduto natural do projeto. Por fim, a representação de projeto deve ser fácil de manter, para que o projeto sempre represente corretamente o programa.

Uma questão natural que surge em qualquer discussão de notação de projeto é: Que notação é realmente a melhor, dados os atributos mencionados? Qualquer resposta a essa questão é subjetiva e aberta a debate. No entanto, parece que a linguagem de projeto de programas oferece a melhor combinação de características. A PDL pode ser embutida diretamente em listagens, melhorando a documentação e tornando menos difícil a manutenção de projeto. A edição pode ser realizada com qualquer editor de texto ou sistema de processamento de texto, pois já existem processadores automáticos, e o potencial para "geração automática de código" é bom.

No entanto, não significa que outras notações de projeto sejam necessariamente inferiores à PDL ou que "não sejam boas" em atributos específicos. A natureza pictórica de fluxogramas e diagramas de atividade fornece uma perspectiva do fluxo de controle preferida por muitos projetistas. O conteúdo tabular preciso das tabelas de decisão é uma excelente ferramenta para aplicações

9 Os produtos mencionados aqui não representam uma recomendação, mas, em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

guiadas por tabelas. E muitas outras representações de projeto (por exemplo, redes de Petri), não apresentadas neste livro, oferecem seus próprios benefícios específicos. Em última análise, a escolha da ferramenta de projeto pode estar relacionada mais de perto a fatores humanos do que a atributos técnicos.

11.6 RESUMO

A ação de projeto no nível de componentes inclui uma seqüência de tarefas que reduz lentamente o nível de abstração no qual o software é representado. O projeto no nível de componentes mostra, em última análise, o software em um nível de abstração bem próximo do código.

Duas diferentes visões de projeto no nível de componentes podem ser consideradas, dependendo da natureza do software a ser desenvolvido. A visão orientada a objetos enfoca elaboração de classes de projeto que vêm de ambos os domínios do problema e da infra-estrutura. A visão convencional refina três diferentes tipos de componentes ou módulos: módulos de controle, módulos do domínio do problema e módulos de infra-estrutura. Em ambos os casos, são aplicados princípios e conceitos básicos de projeto que levam a software de alta qualidade. Quando considerado sob o ponto de vista do processo, projeto no nível de componente refere-se a componentes reusáveis de software e padrões de projeto que são elementos centrais da engenharia de software baseada em componentes.

O projeto orientado a objetos no nível de componentes é baseado em classes. Um número de princípios e conceitos importantes guia o projetista à medida que as classes são elaboradas. Princípios como o Aberto-Fechado e o de Inversão de Dependência, e conceitos como acoplamento e coesão, guiam o engenheiro de software na construção de componentes de software testáveis, implementáveis e manuteníveis. Para conduzir o projeto no nível de componente nesse contexto, classes são elaboradas pela especificação de detalhes de mensagens, identificando adequadamente as interfaces, elaborando atributos e definindo estruturas de dados para implementá-las, descrevendo o fluxo de processamento de cada operação e representando o comportamento no nível de classe ou componente. Em cada caso, iteração de projeto (refabricação) é uma atividade essencial.

Projeto convencional no nível de componentes requer a representação de estruturas de dados, interfaces e algoritmos para um módulo de programa em detalhe suficiente para guiar a geração de código-fonte em linguagem de programação. Para tanto, o projetista usa um certo número de notações de projeto que representam detalhes em nível de componentes, em formato gráfico, tabular ou baseado em texto.

A programação estruturada é uma filosofia de projeto procedural que restringe o número e o tipo de construções lógicas usadas para representar o detalhe algorítmico. O objetivo da programação estruturada é assistir o projetista na definição de algoritmos que são menos complexos e, conseqüentemente, mais fáceis de ler, testar e manter.

REFERÊNCIAS BIBLIOGRÁFICAS

[AMB02] Ambler, S., "UML Component Diagramming Guidelines", disponível em http://www.modelingstyle.info/, 2002.

[BEN02] Bennett, S., McRobb, S., e Farmer, R., *Object-Oriented Analysis and Design*, 2ª ed., McGraw-Hill, 2002.

[BOH66] Bohm, C.; e Jacopini, G., "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules", *CACM*, v. 9, n. 5, p. 366-371, maio 1966.

[CAI75] Caine, S.; e Gordon, K., "PDL — A Tool for Software Design", em *Proc. National Computer Conference*, AFIPS Press, 1975, p. 271-276.

[DIJ65] Dijkstra, E., "Programming Considered as a Human Activity", em *Proc. 1965IFIP Congress*, North-Holland Publishing Co., 1965.

[DIJ72] _____, "The Humble Programmer," 1972 ACM Turing Award Lecture, *CACM*, v. 15, n. 10, p. 859-866, out. 1972.

[DIJ76] _____, "Structured Programming", em *Software Engineering, Concepts and Techniques*, (J. Buxton *et al*, eds.), Van Nostrand-Reinhold, 1976.

[HUR83] Hurley, R. B., *Decision Tables in Software Engineering*, Van Nostrand-Reinhold, 1983.

Document shared on www.docsity.com

[LET01] Lethbridge, T. e Laganiere, R., *Object-Oriented Software Engineering: Practical Software Development Using UML and Java*, McGraw-Hill, 2001.
[LIS88] Liskov, B., "Data Abstraction and Hierarchy", S/GPLAN notices, v. 23, n. 5, maio 1988.
[MAR00] Martin, R., "Design Principles and Design Patterns", disponível em http://www.objectmentor.com, 2000.
[OMG01] *OMG Unified Modeling Specification*, Object Management Group, versão 1.4, set. 2001.
[WAR98] Warmer, J. e Klepp, A., *Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1998.

Problemas e Pontos a Considerar

11.1. O termo *componente* é algumas vezes difícil de definir. Primeiro forneça uma definição genérica e, então, definições mais explícitas para software OO e convencional. Por fim, selecione três linguagens de programação com as quais você tem familiaridade e mostre como cada uma define um componente.

11.2. Por que componentes de controle são necessários em software convencional e geralmente não necessários em software orientado a objetos?

11.3. Descreva o OCP com suas próprias palavras. Por que é importante criar abstrações que servem como uma interface entre componentes?

11.4. Descreva DIP com suas próprias palavras. O que pode acontecer se um projetista confiar excessivamente em concretizações?

11.5. Selecione três componentes que você desenvolveu recentemente e avalie os tipos de coesão que cada um tem. Se tivesse de definir o benefício principal da alta coesão, qual seria?

11.6. Selecione três componentes que você desenvolveu recentemente e avalie os tipos de acoplamento que cada um tem. Se tivesse de definir o benefício principal de baixo acoplamento, qual seria?

11.7. É razoável dizer que os componentes do domínio do problema nunca devem exibir acoplamento externo? Em caso afirmativo, que tipos de componentes devem exibir acoplamento externo?

11.8. Faça alguma pesquisa e desenvolva uma lista de categorias típicas para componentes de infra-estrutura.

11.9. O que é condição de guarda e quando é usada?

11.10. Qual o papel de interfaces em um projeto no nível de componente baseado em classe?

11.11. Os termos *atributos públicos* e *privados* são freqüentemente usados em trabalho de projeto no nível de componente. O que você acha que cada um significa e que conceitos de projeto eles tentam impor?

11.12. O que é uma fonte de dados persistente?

11.13. Desenvolva (1) uma classe de projeto elaborada; (2) descrições de interface; (3) um diagrama de atividade para uma das operações de uma classe; e (4) um diagrama de estados detalhado para uma das classes do *CasaSegura* que discutimos nos capítulos anteriores.

11.14. O refinamento passo a passo e a refabricação são a mesma coisa? Se não, como eles diferem?

11.15. Faça um pouco de pesquisa e descreva três ou quatro construções ou operadores OCL que não tenham sido discutidos na Seção 11.4.

11.16. Selecione uma pequena parte de um programa existente (cerca de 50 a 75 linhas-fonte). Isole as construções de programação estruturada desenhando caixas em volta do código-fonte. O trecho de programa tem construções que violam a filosofia de programação estruturada? Em caso afirmativo, reprojete o código para torná-lo de acordo com as construções de programação estruturada. Em caso negativo, o que você nota sobre as caixas que desenhou?

Leituras e Fontes de Informação Adicionais

Princípios de projeto, conceitos, diretrizes e técnicas para projeto de classes orientadas a objetos e componentes são discutidos em muitos livros de engenharia de software orientada a objetos e análise e projeto OO. Entre as muitas fontes de informação estão: Bennett e seus colegas [BEN02], Larman (*Applying UML and Patterns*, Prentice-Hall, 2001), Lethridge e Laganiere [LET01]; e Nicola e seus colegas (*Streamlined Object Modeling: Patterns, Rules and Implementation*, Prentice-Hall, 2001), Schach (*Object-Oriented and Classical Software Engineering*, quinta edição, McGraw-Hill, 2001), Dennis e seus colegas (*Systems Analysis and Design: An Object-Oriented Approach with UML*, Wiley, 2001), Graham (*Object-Oriented Methods: Principles and Practice*, Addison-Wesley, 2000), Richter (*Designing Flexible Object-Oriented Systems with UML*, Macmillan, 1999), Stevens e Pooley (*Using UML Software Engineering with Objects and Components*, edição revisada, Addison-Wesley, 1999), e Riel (*Object-Oriented Design Heuristics*, Addison-Wesley, 1996).

O projeto por conceito de contrato é um paradigma útil de projeto. Livros de Mitchell e McKim (*Design by Contract by Example*, Addison-Wesley, 2001) e Jezequel e seus colegas (*Design Patterns and Contracts*, Addison-Wesley, 1999) cobrem esse tópico com algum detalhe. Metsker (*Design Patterns Java Workbook*, Addison-Wesley, 2002) e Shalloway e Trott (*Design Patterns Explained: A New Perspective on Object-Oriented Design*, Addison-Wesley, 2001) consideram o impacto de padrões no projeto de componentes de software. Iteração de projeto é essencial para a criação de projetos de alta qualidade. Fowler (*Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999) fornece um guia útil que pode ser aplicado à medida que o projeto evolui.

O trabalho de Linger, Mills e Witt (*Structured Programming Theory and Practice*, Addison-Wesley, 1979) permanece um tratamento definitivo do assunto. O texto contém uma boa PDL bem como discussão detalhada das ramificações de programação estruturada. Outros livros que enfocam tópicos de projeto procedural para sistemas tradicionais incluem os de Robertson (*Simple Program Design*, terceira edição, Course Technology, 2000), Farrell (*A Guide to Programming Logic and Design*, Course Technology, 1999), Bentley (*Programming Pearls*, segunda edição, Addison-Wesley, 1999), e Dahl (*Structured Programming*, Academic Press, 1997).

Relativamente, poucos livros recentes têm sido dedicados somente a projeto no nível de componente. Em geral, obras sobre linguagem de programação tratam de projeto procedural com algum detalhe, mas sempre no contexto da linguagem introduzida pelo livro. Centenas de títulos estão disponíveis.

Uma ampla variedade de fontes de informação sobre projeto no nível de componente está disponível na Internet. Uma lista atualizada de referências que são relevantes ao projeto no nível de componente pode ser encontrada no site deste livro: **http://www.mhhe.com/pressman**.

Document shared on www.docsity.com

CAPÍTULO

12 PROJETO DE INTERFACE COM O USUÁRIO

CONCEITOS-CHAVE

acessibilidade 283
análise do fluxo de trabalho 275
análise de tarefas 272
refinamento de objetos ... 274
facilidade de ajuda 281
interface
análise 271
avaliação 284
consistência 266
modelos 268
internacionalização 283
passos de projeto 277
padrões 280
regras de ouro 265
usabilidade 268

Como garanto a correção do trabalho? O protótipo é "direcionado por testes" pelos usuários e a realimentação por meio do teste de direcionamento é usada para a modificação iterativa seguinte do protótipo.

A planta de execução de uma casa (seu projeto arquitetural) não fica completa sem uma representação de portas, janelas e ligações com a rede de água, elétrica e telefônica (para não mencionar TV a cabo). As "portas, janelas e ligações com as redes" de um software para computador formam o projeto de interface de um sistema.

O projeto de interface concentra-se em três áreas de preocupação: (1) o projeto de interfaces entre componentes do software, (2) o projeto de interfaces entre o software e outros produtores e consumidores de informação não-humanos (outras entidades externas), e (3) o projeto de interface entre um ser humano (o usuário) e o computador. Neste capítulo focalizaremos exclusivamente a terceira categoria de projeto de interface — *projeto de interface com o usuário (IU)*.

No prefácio de seu livro clássico sobre projeto de interface com o usuário, Ben Shneiderman [SHN90] declara:

> Frustração e ansiedade são parte da vida diária para muitos usuários de sistemas de informação computadorizados. Eles lutam para aprender a linguagem de comando ou sistemas de seleção de menus que se destinam a ajudá-los a fazer seu serviço. Algumas pessoas se deparam com casos tão sérios de susto com computador, terror de terminal ou neurose de rede que evitam usar sistemas computadorizados.

Os problemas a que Shneiderman se refere são reais. É verdade que interfaces gráficas com o usuário, janelas, ícones e dispositivos de mouse têm eliminado muitos dos mais horripilantes problemas de interface. Mas mesmo em um "mundo de janelas", todos nós já encontramos interfaces com o usuário difíceis de aprender e de usar, confusas, implacáveis e, em muitos casos, totalmente frustrantes. No entanto, alguém gastou tempo e energia construindo cada uma dessas interfaces, e não é provável que o construtor tenha criado esses problemas de propósito.

PANORAMA

O que é? O projeto de interface com o usuário cria um meio efetivo de comunicação entre o ser humano e o computador. Seguindo um conjunto de princípios de projeto de interface, o projeto identifica objetos e ações de interface e depois cria um layout de tela que forma a base para um protótipo de interface com o usuário.

Quem faz? Um engenheiro de software projeta a interface com o usuário pela aplicação de um processo iterativo que se apóia sobre princípios de projeto amplamente aceitos.

Por que é importante? Se o software é difícil de usar, se o leva a cometer erros ou se frustra seus esforços de alcançar suas metas, você não gostará dele, independentemente do poder computacional que exibe ou da funcionalidade que oferece. A interface tem de ser correta porque ela molda a percepção do software pelo usuário.

Quais são os passos? O projeto de interface com o usuário começa com a identificação dos requisitos do usuário, das tarefas e do ambiente. Uma vez identificadas as tarefas do usuário, cenários do usuário são criados e analisados para definir um conjunto de objetos e ações de interface. Os cenários formam a base para a criação do layout de tela que ilustra o projeto gráfico e a colocação de ícones, definição de texto descritivo na tela, especificação e títulos para janelas e especificação de itens de menu principais e secundários. Ferramentas são usadas para prototipar e, por fim, implementar o modelo de projeto; e o resultado é avaliado quanto à qualidade.

Qual é o produto do trabalho? Cenários do usuário são criados e layouts de telas, gerados. Um protótipo de interface é desenvolvido e modificado de modo iterativo.

Como tenho certeza de que fiz corretamente? O protótipo é "direcionado por testes" pelos usuários e a realimentação por meio do teste de direcionamento é usada para a modificação iterativa seguinte do protótipo.

O projeto de interface com o usuário tem tanto a ver com o estudo de pessoas quanto com aspectos tecnológicos. Quem é o usuário? Como ele aprende a interagir com um novo sistema baseado em computador? Como interpreta a informação produzida pelo sistema? O que espera do sistema? Essas são apenas algumas das muitas questões que precisam ser formuladas e respondidas como parte do projeto de interface com o usuário.

12.1 AS REGRAS DE OURO

Em seu livro sobre projeto de interface, Theo Mandel [MAN97] cunha três "regras de ouro":

1. Coloque o usuário no controle.
2. Reduza a carga de memória do usuário.
3. Faça a interface consistente.

Essas regras de ouro, na verdade, formam a base para um conjunto de princípios de projeto da interface com o usuário e pautam essa importante atividade de projeto de software.

12.1.1 Coloque o Usuário no Controle

Durante uma seção de elicitação de requisitos para um novo sistema de informação importante, um usuário-chave foi inquirido sobre os atributos da interface gráfica orientada a janelas. "O que eu realmente gostaria", disse o usuário solenemente, "é de um sistema que leia minha mente. Que saiba o que quero fazer antes que eu precise fazê-lo e torne muito fácil para eu conseguir que seja feito. Isso é tudo, apenas isso."

Minha primeira reação foi sacudir a cabeça e sorrir, mas esperei um momento. Não havia absolutamente nada errado com a solicitação do usuário. Ele queria um sistema que reagisse às suas necessidades e o ajudasse a conseguir que as coisas fossem feitas. Queria controlar o computador, não o oposto.

A maioria das limitações e restrições de interface impostas por um projetista é destinada a simplificar o modo de interação. Mas para quem? Em muitos casos o projetista pode introduzir restrições e limitações para simplificar a implementação da interface. O resultado pode ser uma interface fácil de construir, mas frustrante de usar.

Mandel [MAN97] define alguns princípios de projeto que permitem ao usuário manter o controle:

Defina os modos de interação de uma forma que não force o usuário a ações desnecessárias ou indesejadas. Um modo de interação é o estado atual da interface. Por exemplo, se *verificar ortografia* é selecionado em um menu de processador de texto, o software passa para um modo de verificação da ortografia. Não há razão para forçar o usuário a permanecer no modo de verificação de ortografia se ele desejar fazer uma pequena edição de texto no meio do caminho. O usuário deve entrar e sair do modo com pouco ou nenhum esforço.

Proporcione interação flexível. Como diferentes usuários têm diferentes preferências de interação, a escolha deve ser oferecida. Por exemplo, o software deve permitir a um usuário interagir por meio de comandos do teclado, movimento do mouse, caneta digitalizadora ou comandos com reconhecimento de voz. Mas nem toda ação é adequada a todos os mecanismos de interação. Considere, por exemplo, a dificuldade de usar comandos de teclado (ou entrada por voz) para desenhar uma forma complexa.

Permita que a interação com o usuário possa ser interrompida e desfeita. Mesmo quando envolvido em uma seqüência de ações, o usuário poderá interromper a seqüência para fazer alguma outra coisa (sem perder o trabalho que já tinha sido feito). O usuário poderá também "desfazer" qualquer ação.

Simplifique a interação à medida que os níveis de competência progridem e permita que a interação seja personalizada. Os usuários freqüentemente descobrem que realizam a mesma seqüência de interações repetidamente. Vale a pena projetar um mecanismo "macro" que permita a um usuário avançado personalizar a interface para facilitar a interação.

Document shared on www.docsity.com

Esconda detalhes técnicos internos do usuário esporádico. A interface com o usuário deve levá-lo ao mundo virtual da aplicação. Ele não deve perceber o sistema operacional, funções de gerenciamento de arquivos ou outra tecnologia obsoleta de computação. Em essência, a interface nunca deve exigir que o usuário interaja em um nível que seja "interno" à máquina (um usuário nunca precisa teclar comandos do sistema operacional de dentro de um software de aplicação).

Projete a interação direta com objetos que aparecem na tela. O usuário tem uma sensação de controle quando pode manipular os objetos necessários para realizar uma tarefa de modo semelhante ao que ocorreria se o objeto fosse uma coisa física. Por exemplo, uma interface de aplicação que permite a um usuário "esticar" um objeto (aumentá-lo no tamanho) é uma implementação de manipulação direta.

> "Eu tenho sempre desejado que meu computador seja tão fácil de usar quanto o meu telefone. Meu desejo realizou-se. Eu não sei mais como usar o meu telefone."
>
> Bjarne Stronstrup (criador do C++)

12.1.2 Reduza a Carga de Memória do Usuário

Quanto mais um usuário tiver de lembrar, mais sujeita a erros será a interação com o sistema. É por isso que uma interface bem-projetada não sobrecarrega a memória do usuário. Sempre que possível, o sistema deve "lembrar" informação pertinente e assistir o usuário com um cenário de interação que ajude a lembrar. Mandel [MAN97] define os princípios de projeto que permitem a uma interface reduzir a carga de memória do usuário:

Reduza a demanda da memória de curto prazo. Quando os usuários estão envolvidos em tarefas complexas, a demanda da memória de curto prazo pode ser significativa. A interface deve ser projetada para reduzir a necessidade de lembrar ações e resultados anteriores. Pode-se conseguir isso por meio do fornecimento de indicações visuais que permitem ao usuário reconhecer ações anteriores, em vez de ter de se lembrar delas.

Estabeleça *defaults* significativos. O conjunto inicial de parâmetros (*defaults*) deve fazer sentido para o usuário médio, mas o usuário poderá especificar preferências individuais. No entanto, uma opção "*reset*" (refazer os parâmetros iniciais) deve estar disponível, permitindo a redefinição dos valores originais.

Defina atalhos intuitivos. Quando são usados mnemônicos para realizar uma função do sistema (por exemplo, alt-P para invocar a função de impressão), o mnemônico deve ser ligado à ação de um modo que seja fácil de lembrar (por exemplo, primeira letra da tarefa a ser invocada).

O leiaute visual da interface deve ser baseado em uma metáfora do mundo real. Por exemplo, um sistema de pagamento de contas deve usar uma metáfora de talão e canhoto de cheques para guiar o usuário durante o processo de pagamento de contas. Isso permite ao usuário se apoiar em indicações visuais bem-entendidas, em vez de memorizar uma seqüência de interação esotérica.

Revele informação de um modo progressivo. A interface deve ser organizada hierarquicamente. A informação sobre uma tarefa, um objeto ou algum comportamento deve ser inicialmente apresentada em um alto nível de abstração. Mais detalhes devem ser apresentados, após o usuário indicar interesse com um clique do mouse. Um exemplo, comum a várias aplicações de processamento de texto, é a função de sublinhar. A função em si é uma de várias funções que constam de um menu de estilo de texto. No entanto, não são listados todos os modos de sublinhar. O usuário deve escolher sublinhar, depois são apresentadas todas as opções para sublinhar (sublinhar com linha simples, sublinhar com linha dupla e sublinhar com linha tracejada).

12.1.3 Faça a Interface Consistente

A interface deve apresentar e receber informação de modo consistente. Isso implica que (1) toda a informação visual seja organizada de acordo com um padrão de projeto mantido ao longo de todos os mostradores de tela, (2) mecanismos de entrada são restritos a um conjunto limitado,

CASASEGURA

Violação de uma "Regra de Ouro" da Interface com o Usuário

A cena: Sala de Vinod, à medida que o projeto de interface começa.

Os personagens: Vinod e Jamie, membros da equipe de software do *CasaSegura*.

A conversa:

Jamie: Estive pensando sobre a interface da função de vigilância.

Vinod (sorrindo): Pensar é bom.

Jamie: Penso que talvez possamos simplificar um pouco as coisas.

Vinod: O que você quer dizer com isso?

Jamie: Bem, que tal se eliminássemos a planta baixa completamente? Ela é atraente, mas vai consumir muito esforço de desenvolvimento. Em vez disso, podemos pedir ao usuário para especificar a câmera que ele deseja ver e depois mostrar o vídeo em uma janela de vídeo.

Vinod: Como o proprietário se lembra de quantas câmeras estão instaladas e onde elas estão?

Jamie (ligeiramente irritada): Ele é o proprietário, ele deveria saber.

Vinod: Mas e se ele não souber?

Jamie: Ele deveria.

Vinod: Esse não é o ponto... e se ele se esquecer?

Jamie: Humm, poderíamos fornecer uma lista das câmeras em operação e suas localizações.

Vinod: Isso é possível, mas por que ele teria que pedir a lista?

Jamie: Está bem, nós fornecemos a lista se ele pedir ou não.

Vinod: Melhor. Pelo menos ele não tem de se lembrar de algo que nós podemos lhe fornecer.

Jamie (pensando por um momento): Mas você gosta da planta baixa, não é?

Vinod: Sim.

Jamie: Qual você acha que marketing vai preferir?

Vinod: Você está brincando, certo?

Jamie: Não.

Vinod: Bem... a atrativa... eles adoram características de produtos sexy... eles não estão interessados em qual é mais fácil de construir.

Jamie (suspirando): Está bem, talvez eu faça um protótipo de ambos.

Vinod: Boa idéia ... depois nós deixamos o cliente decidir.

usado consistentemente ao longo de toda a aplicação, e (3) mecanismos para navegar de tarefa a tarefa são consistentemente definidos e implementados. Mandel [MAN97] define um conjunto de princípios de projeto que ajuda a fazer a interface consistente:

Permita ao usuário situar a tarefa atual em um contexto significativo. Muitas interfaces implementam camadas de interação complexas com muitas imagens de tela. É importante fornecer indicadores (títulos de janela, ícones gráficos, código de cores consistente) que permitam ao usuário saber o contexto do trabalho em mãos. Além disso, o usuário pode determinar de onde veio e que alternativas existem de transição para uma nova tarefa.

Mantenha consistência ao longo de uma família de aplicações. Um conjunto de aplicações (ou produtos) deve implementar as mesmas regras de projeto, de modo que seja mantida a consistência para toda a interação.

Se modelos interativos anteriores criaram expectativas para o usuário, não faça modificações, a menos que haja forte razão para isso. Quando uma seqüência interativa particular torna-se uma norma de fato (por exemplo, o uso de alt-S para salvar um arquivo), o usuário espera isso em toda a aplicação que encontra. Uma modificação (por exemplo, usar alt-S para invocar aumento de escala [*scaling*]) causará confusão.

Os princípios de projeto de interface discutidos nesta e nas seções anteriores fornecem diretrizes básicas para um engenheiro de software. Nas seções seguintes examinaremos o processo de projeto de interface propriamente dito.

> "Coisas que parecem diferentes devem agir diferentemente. Coisas que parecem iguais devem agir da mesma forma."
>
> Larry Marine

Document shared on www.docsity.com

INFO

Usabilidade

Em um artigo significativo sobre usabilidade, Larry Constantine [CON95] formula uma questão que tem significativa importância no assunto: "Afinal de contas, o que os usuários querem?". Ele responde desse modo: "O que os usuários realmente querem são boas ferramentas. Todos os sistemas de software, desde sistemas operacionais e linguagem a aplicações de entrada de dados e apoio à decisão, são apenas ferramentas. Os usuários finais querem das ferramentas que nós construímos para eles muito das mesmas coisas que esperamos das ferramentas que usamos. Eles querem sistemas que sejam fáceis de aprender e que os ajudem em seu trabalho. Eles querem software que não os atrase, que não os enganem ou confundam, que não lhes facilite cometer erros ou dificulte terminar o trabalho".

Constantine alega que usabilidade não é derivada da estética, de mecanismos atuais de interação, ou de interface com inteligência incorporada. Em vez disso, ela ocorre quando a arquitetura da interface atende às necessidades das pessoas que irão usá-la.

Uma definição formal de usabilidade é algo ilusório. Donahue e seus colegas [DON99] a definem da seguinte maneira: "Usabilidade é uma medida de quanto um sistema de computador... facilita o aprendizado; ajuda os aprendizes a lembrar o que aprenderam; reduz a probabilidade de erros; habilita-os a ser eficientes e faz com que fiquem satisfeitos com o sistema".

O único modo de determinar se "usabilidade" existe em um sistema que você está construindo é conduzir avaliação ou teste de usabilidade. Observe usuários interagindo com o sistema e responda às seguintes questões [CON95]:

- O sistema é utilizável sem ajuda ou instrução contínua?
- As regras de interação ajudam um usuário competente a trabalhar eficientemente?
- Os mecanismos de interação tornam-se mais flexíveis à medida que os usuários tornam-se mais competentes?
- O sistema tem estado sintonizado com o ambiente físico e social no qual ele será usado?
- O usuário está consciente do estado do sistema? O usuário sabe sempre onde está?
- A interface está estruturada de modo lógico e consistente?
- Mecanismos de interação, ícones e procedimentos são consistentes em toda a interface?
- A interação prevê erros e ajuda os usuários a corrigi-los?
- A interface é tolerante a erros que são cometidos?
- A interação é simples?

Se cada uma dessas questões for respondida afirmativamente, provavelmente a usabilidade foi atingida.

Entre os muitos benefícios mensuráveis derivados de um sistema usável [DON99] estão o aumento de vendas e a satisfação do usuário, vantagem competitiva, melhores opiniões na mídia, melhores comentários, custos de manutenção reduzidos, produtividade do usuário final melhorada, custos de treinamento reduzidos, custos de documentação reduzidos, probabilidade reduzida de litígio por clientes insatisfeitos.

12.2 Análise e Projeto de Interface com o Usuário

Veja na Web

Uma excelente fonte de informação de projeto de IU pode ser encontrada em www.useit.com.

O processo para análise e projeto de uma interface com o usuário tem início com a criação de diferentes modelos da função do sistema (tal como é percebida externamente). As tarefas humanas e orientadas a computador, necessárias para realizar a função do sistema, são então delineadas; os aspectos de projeto que se aplicam a todos os projetos de interfaces são considerados; as ferramentas são usadas para prototipar e, por fim, implementar o modelo de projeto; e o resultado é avaliado quanto à qualidade.

12.2.1 Modelos de Análise e de Projeto de Interface

Quatro diferentes modelos entram em jogo quando uma interface com o usuário deve ser analisada e projetada. Um engenheiro humano (ou engenheiro de software) estabelece um *modelo de usuário*, o engenheiro de software cria um *modelo de projeto*, o usuário final desenvolve uma imagem mental freqüentemente chamada *modelo mental* ou *percepção do sistema* e os implementadores do sistema criam um *modelo de implementação*. Infelizmente, cada um desses modelos pode ser significativamente diferente. O papel do projetista de interface é reconciliar essas diferenças e derivar uma representação consistente da interface.

> "Se há um 'truque' nelo, a IU está com problema."
>
> Douglas Anderson

O modelo de usuário estabelece o perfil dos usuários finais do sistema. Para construir uma interface efetiva com o usuário, "todo o projeto deveria começar com uma compreensão do usuário pretendido, incluindo perfis de sua idade, sexo, habilidades físicas, educação, *background* étnico ou cultural, motivação, metas e personalidade" [SHN90]. Além disso, os usuários podem ser categorizados como:

Novatos. Nenhum conhecimento sintático[1] do sistema e pouco conhecimento semântico[2] da aplicação ou do uso de computadores em geral.

Usuários esporádicos e conhecedores. Razoável conhecimento semântico da aplicação, mas lembrança relativamente pequena da informação sintática necessária para usar a interface.

Usuários freqüentes e conhecedores. Bom conhecimento semântico e sintático que freqüentemente leva à "síndrome do usuário possante"; isto é, indivíduos que buscam atalhos e modos de interação abreviados.

Um modelo de projeto do sistema inteiro incorpora representações de dados, arquitetural, de interface e procedural do software. A especificação de requisitos pode estabelecer certas restrições que ajudam a definir o usuário do sistema, mas o projeto da interface é em geral secundário para o modelo de projeto[3].

AVISO

Mesmo um usuário novato quer atalhos; mesmo usuários freqüentes e competentes algumas vezes precisam de guia. Dê-lhes então o que eles necessitam.

PONTO CHAVE

O modelo mental do usuário molda como ele percebe a interface e se a IU atende às suas necessidades.

O *modelo mental* do usuário (percepção do sistema) é a imagem do sistema que os usuários finais têm em suas mentes. Por exemplo, se o usuário de um sistema especial de formatação de página fosse solicitado a descrever sua operação, a percepção do sistema guiaria a resposta. A precisão da descrição vai depender do perfil do usuário (novatos forneceriam uma resposta esquemática, na melhor das hipóteses) e da familiaridade geral com software no domínio de aplicação. Um usuário que entende plenamente de formatação de página, mas trabalhou com o sistema específico apenas uma vez, pode na verdade ser capaz de fornecer uma descrição mais completa de sua função do que o novato que gastou semanas tentando aprender o sistema.

> "[P]reste atenção no que os usuários fazem, não no que eles dizem."
>
> Jakob Nielsen

O modelo de implementação combina a manifestação exterior do sistema baseado em computador (a aparência e sentido da interface) acoplada a toda informação de apoio (livros, manuais, videotapes, arquivos de ajuda) que descreve a sintaxe e a semântica do sistema. Quando o modelo de implementação e o modelo mental do usuário são coincidentes, os usuários geralmente se sentem confortáveis com o software e o usam efetivamente. Para conseguir essa "coincidência" dos modelos, o modelo de projeto deve ter sido desenvolvido para acomodar a informação contida no modelo do usuário, e o modelo de implementação deve refletir precisamente a informação sintática e semântica sobre a interface.

Os modelos descritos nesta seção são "abstrações do que o usuário está fazendo, ou pensa que está fazendo, ou que mais alguém pensa que deveria estar fazendo, quando usa um sistema interativo" [MON84]. Em essência, esses modelos permitem ao projetista de interface satisfazer o elemento-chave do princípio mais importante de projeto de interface com o usuário: *Conheça o usuário, conheça as tarefas*.

---

1 Nesse contexto, *conhecimento sintático* se refere à mecânica de interação que é necessária para usar a interface efetivamente.

2 *Conhecimento semântico* se refere a um sentido subjacente da aplicação — compreensão das funções que são realizadas, significado da entrada e da saída, e metas e objetivos do sistema.

3 Essa não é a maneira que as coisas deveriam ser. Em muitos casos, o projeto da interface com o usuário é tão importante quanto o projeto arquitetural e no nível de componentes.

Document shared on www.docsity.com

12.2.2 O Processo

O processo de análise e projeto das interfaces com o usuário é iterativo e pode ser representado usando-se um modelo espiral semelhante ao discutido no Capítulo 3. Observando-se a Figura 12.1, o processo de projeto da interface com o usuário abrange quatro atividades distintas de arcabouço [MAN97]:

1. Análise e modelagem do usuário, tarefa e ambiente.
2. Projeto da interface.
3. Construção da interface.
4. Validação da interface.

A espiral mostrada na Figura 12.1 implica que cada uma dessas tarefas vai ocorrer mais de uma vez, com cada passo em volta da espiral representando refinamento adicional dos requisitos e do projeto resultante. Na maioria dos casos, a atividade de construção envolve prototipagem — o único modo prático de validar o que foi projetado.

A análise de interface concentra-se no perfil do usuário que vai interagir com o sistema. Nível de aptidão, entendimento do negócio e receptividade geral ao novo sistema são registrados e diferentes categorias de usuários são definidas. Para cada categoria de usuário são levantados requisitos. Em essência, o engenheiro de software tenta entender a percepção do sistema (veja a Seção 12.2.1) de cada classe de usuário.

> "É melhor projetar a experiência do usuário do que retificá-la."
>
> **Jon Meads**

Uma vez definidos os requisitos gerais, é conduzida uma tarefa de análise mais detalhada. As tarefas que o usuário realiza para alcançar as metas do sistema são identificadas, descritas e elaboradas (ao longo de vários passos iterativos pela espiral). A análise de tarefa é discutida em mais detalhes na Seção 12.3.

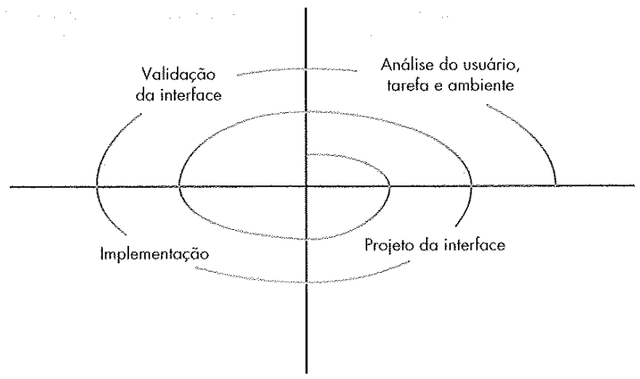
A análise do ambiente do usuário concentra-se no ambiente de trabalho físico. Entre as questões a serem formuladas estão:

? O que precisamos saber sobre o ambiente quando começamos o projeto de IU?

- Onde a interface será localizada fisicamente?
- O usuário vai estar sentado, em pé, ou realizando outras tarefas não relacionadas à interface?
- O hardware da interface acomoda as restrições de espaço, luz ou ruído?
- Há considerações especiais de fatores humanos determinados por fatores ambientais?

FIGURA 12.1

O processo de projeto da interface com o usuário



A informação recolhida como parte da atividade de análise é usada para criar um modelo de análise da interface. A atividade de projeto tem início usando-se esse modelo como base.

A meta do projeto de interface é definir um conjunto de objetos e ações de interface (e suas representações de tela) que permitam ao usuário realizar todas as tarefas estabelecidas, de um modo que satisfaça todas as metas de usabilidade definidas para o sistema. O projeto de interface é discutido com mais detalhes na Seção 12.4.

A atividade de construção normalmente começa com a criação de um protótipo que permite a avaliação de cenários de uso. À medida que o processo de projeto iterativo continua, ferramentas de desenvolvimento de interface com o usuário (veja o quadro da Seção 12.4) podem ser usadas para completar a construção da interface.

A validação concentra-se (1) na capacidade da interface de implementar todas as tarefas do usuário corretamente, de acomodar todas as variantes de tarefa e de atingir todos os requisitos gerais do usuário; (2) no grau em que a interface é fácil de usar e de aprender; e (3) na aceitação dos usuários da interface como ferramenta útil ao seu trabalho.

Como já mencionamos antes, as atividades descritas nesta seção ocorrem iterativamente. Assim, não há necessidade de tentar especificar cada detalhe (para o modelo de análise ou de projeto) no primeiro passo. Os passos subseqüentes ao longo do processo aperfeiçoam os detalhes de tarefa, informação de projeto e características operacionais da interface.

12.3 ANÁLISE DE INTERFACE[4]

Um princípio-chave de todos os modelos de engenharia de software é descrito a seguir: *entenda melhor o problema antes de tentar projetar uma solução*. No caso de projeto de interface com o usuário, entender o problema significa entender (1) as pessoas (usuários finais) que vão interagir com o sistema por meio da interface; (2) as tarefas que os usuários finais devem realizar para fazer seu trabalho; (3) o conteúdo apresentado como parte da interface; e (4) o ambiente em que essas tarefas serão conduzidas. Nas seções seguintes, examinaremos cada um desses elementos da análise de interface com o objetivo de estabelecer uma fundação sólida para as tarefas de projeto que se seguem.

12.3.1 Análise do Usuário

Anteriormente notamos que cada usuário tem uma imagem mental ou percepção do sistema do software que pode ser diferente da imagem mental desenvolvida por outros usuários. Além disso, a imagem mental do usuário pode ser amplamente diferente do modelo de projeto do engenheiro de software. O único modo pelo qual um projetista pode fazer a imagem mental e o modelo de projeto convergirem é trabalhar para entender os usuários em si, bem como o modo pelo qual essas pessoas vão usar o sistema. Informação de um amplo conjunto de fontes pode ser usada para conseguir isso:

? Como aprendemos o que o usuário quer da IU?

Entrevistas com usuário. Com abordagem mais direta, entrevistas envolvem representantes da equipe de software que se encontram com usuários finais para melhor entender as suas necessidades, motivações, cultura de trabalho e grande quantidade de outros tópicos. Isso pode ser conseguido com encontros individuais ou por grupos focados.

Entrada de vendas. O pessoal de vendas se encontra com clientes e usuários regularmente e pode obter informação que ajudará a equipe de software a categorizar os usuários e a melhor entender seus requisitos.

Entrada de marketing. Análise de mercado pode ser inestimável na definição de segmentos do mercado que fornecem um entendimento de como cada segmento pode usar o software de modos sutilmente diferentes.

4 Esta seção poderia ter sido colocada no Capítulo 8, pois nele são discutidos os tópicos de análise de requisitos. Foi colocada aqui, porque análise e projeto de interface estão intimamente ligados, e a fronteira entre os dois é freqüentemente confusa.

Document shared on www.docsity.com

AVISO

Acima de tudo, gaste tempo conversando com os usuários reais, mas seja cuidadoso. Uma opinião forte não necessariamente significa que a maioria dos usuários vai concordar.

Entrada de suporte. A equipe de suporte conversa com os usuários diariamente, fazendo deles a mais provável fonte de informação sobre o que funciona e o que não funciona, do que os usuários gostam e do que não gostam, quais características geram dúvidas e quais são fáceis de usar.

O seguinte conjunto de questões (adaptadas de [HAC98]) ajudará o projetista de interface a melhor entender os usuários de um sistema:

PONTO CHAVE

Como aprendemos sobre a demografia e as características dos usuários finais?

- Os usuários são profissionais treinados, técnicos, funcionários burocráticos ou operários de fábrica?
- Que nível de educação formal o usuário médio tem?
- Os usuários são capazes de aprender com materiais escritos ou têm expressado desejo de treinamento em sala de aula?
- Os usuários são digitadores experientes ou têm fobia a teclado?
- Qual o intervalo de idade da comunidade de usuários?
- Os usuários serão representados predominantemente por um gênero?
- Como os usuários são recompensados pelo trabalho que realizam?
- Os usuários trabalham no horário normal de escritório ou trabalham até que o serviço acabe?
- O software será parte integral do trabalho que o usuário faz, ou será usado apenas ocasionalmente?
- Qual a língua principal falada entre os usuários?
- Quais as conseqüências se um usuário cometer um erro ao usar o sistema?
- Os usuários são especialistas no assunto tratado pelo sistema?
- Os usuários querem saber sobre a tecnologia que está por trás da interface?

A resposta a essas e a questões similares permitirão que o projetista entenda quem são os usuários finais, o que é provável para motivá-los ou satisfazê-los, como podem ser agrupados em diferentes classes ou perfis de usuário, quais os seus modelos mentais do sistema e como a interface com o usuário precisa ser caracterizada para satisfazer suas necessidades.

12.3.2 Modelagem e Análise de Tarefas

PONTO CHAVE

O objetivo do usuário é realizar uma ou mais tarefas pela IU. Para tanto, a IU deve fornecer mecanismos que permitam ao usuário alcançar seu objetivo.

O objetivo da análise de tarefa é responder às seguintes questões:

- Qual trabalho será realizado pelo usuário em circunstâncias específicas?
- Quais tarefas e subtarefas serão realizadas quando o usuário faz o trabalho?
- Quais objetos específicos do domínio do problema serão manipulados pelo usuário à medida que o trabalho é realizado?
- Qual a seqüência de tarefas de trabalho — o fluxo de trabalho?
- Qual a hierarquia das tarefas?

Para responder a essas questões, o engenheiro de software deve se apoiar nas técnicas de análise discutidas nos capítulos 7 e 8, mas, nessa instância, as técnicas são aplicadas à interface com o usuário.

Casos de uso. Em capítulos anteriores observamos que o caso de uso descreve o modo pelo qual um ator (no contexto de projeto de interface com o usuário, um ator é sempre uma pessoa) interage com um sistema. Quando usado como parte da análise de tarefa, o caso de uso é desenvolvido para mostrar como um usuário final realiza alguma tarefa específica relativa ao trabalho. Na maioria das vezes, o caso de uso é escrito em um estilo informal (um único parágrafo) na primeira pessoa.

Por exemplo, considere que uma pequena empresa de software quer construir um sistema de projeto apoiado por computador explicitamente para projetistas de interiores. Para obter melhor compreensão de como eles fazem seu trabalho, projetistas de interior reais são questionados para descrever funções específicas de projeto. Quando perguntado "Como você decide onde colocar um móvel em um cômodo?", um projetista de interiores escreve o seguinte caso de uso informal:

> Eu começo esboçando a planta baixa do cômodo, as dimensões e localização das janelas e portas. Estou muito preocupado com a luz que entra na sala, com a vista das janelas (se for bonita, quero chamar atenção para ela), com o comprimento das paredes livres, com o fluxo de movimentação pelo cômodo. Olho então a lista de móveis que meu cliente e eu escolhemos — mesas, cadeiras, sofá, armários; a lista de destaques — lâmpadas, tapetes, pinturas, esculturas, plantas, peças pequenas; e minhas anotações sobre qualquer desejo de meu cliente a respeito de colocação. Desenho então cada item da minha lista usando um gabarito que está em escala com a planta baixa. Rotulo cada item e uso lápis porque sempre movo as coisas. Considero um certo número de alternativas de colocação e decido por uma da qual gosto mais. Depois, desenho uma perspectiva (um quadro 3D) do cômodo para dar ao meu cliente uma sensação de como ele vai ficar.

Esse caso de uso fornece uma descrição básica de uma tarefa de trabalho importante para o sistema de projeto apoiado por computador. Com base nele, o engenheiro de software pode extrair tarefas, objeto e o fluxo global da interação. Além disso, características adicionais do sistema que vão agradar o projetista de interiores podem também ser concebidas. Por exemplo, uma foto digital da parte externa pode ser tirada de cada janela do cômodo. Quando o cômodo for apresentado, a vista externa real poderia ser representada através de cada janela.

CASASEGURA

Casos de Uso para o Projeto de IU

A cena: Sala de Vinod, à medida que o projeto de interface continua.

Os personagens: Vinod e Jamie, membros da equipe de software do *CasaSegura*.

A conversa:

Jamie: Consegui nosso contato com marketing e pedi que ela escrevesse um caso de uso para a interface de vigilância.

Vinod: Do ponto de vista de quem?

Jamie: Do proprietário, de quem mais seria?

Vinod: Há também o papel do administrador do sistema. Mesmo que o proprietário esteja desempenhando esse papel, é um ponto de vista diferente. O "administrador" coloca o sistema em ação, configura o material, dispõe a planta baixa, localiza as câmeras...

Jamie: Tudo o que eu fiz marketing fazer foi desempenhar o papel de um proprietário que deseja ver vídeo.

Vinod: Está certo. É um dos comportamentos principais da interface da função de vigilância. Mas, vamos ter de examinar também o comportamento do administrador do sistema.

Jamie (irritada): Você está certo.

(Jamie retira-se para procurar a pessoa de marketing. Ela retorna poucas horas depois.)

Jamie: Eu tive sorte. Encontrei nosso contato do marketing e trabalhamos juntos no caso de uso do administrador. Basicamente, vamos definir "administração" como uma função aplicável a todas as outras funções do *CasaSegura*. Eis a que chegamos.

(Jamie mostra o caso de uso informal a Vinod.)

Caso de Uso Informal: Quero ser capaz de ajustar e editar o layout do sistema em qualquer momento. Quando ajusto o sistema, seleciono uma função de administração. Ele me pergunta se quero fazer um novo ajuste ou se quero editar um ajuste existente. Se eu selecionar um novo ajuste, o sistema exibe uma tela de desenho que me permite desenhar na planta baixa em uma grade, com ícones para paredes, janelas e portas de modo que é fácil desenhar. Eu esboço os ícones com os seus comprimentos adequados. O sistema exibirá os comprimentos em pés ou metros (posso selecionar o sistema de medida). Posso escolher em uma biblioteca de câmeras e sensores e colocá-los na planta baixa. Eu obtenho um rótulo para cada um, ou o sistema rotulará automaticamente. Posso estabelecer ajustes de sensores e câmeras por meio de menus adequados. Se eu selecionar editar, posso mover sensores ou câmeras, adicionar novos ou remover existentes, editar a planta baixa e editar o ajuste de câmeras e sensores. Em cada caso, espero que o sistema faça verificação de consistência e me ajude a evitar erros.

Vinod (depois de ler o cenário): Certo, há provavelmente alguns padrões de projeto ou componentes reusáveis também úteis de IGU (Interface Gráfica com o Usuário) para programas de desenho. Aposto 50 pratas como podemos implementar parte ou a maioria da interface do administrador usando-os.

Jamie: Concordo. Vou verificar isso.

Document shared on www.docsity.com

Refinamento das tarefas. No Capítulo 9 discutimos o refinamento passo a passo (também chamado *decomposição funcional* ou *refinamento passo a passo*) como mecanismo para refinar as tarefas de processamento, necessárias para que o software consiga algumas funções desejadas. A análise de tarefas para projeto de interface usa uma abordagem de refinamento ou de orientação a objetos, mas aplica essa abordagem a atividades humanas.

AVISO

O refinamento de tarefas é bastante útil, mas também pode ser perigoso. Só porque você tem de aperfeiçoar uma tarefa, não considere que não exista outro modo para fazê-la, e que o outro modo será experimentado quando a IU estiver implementada.

A análise de tarefas pode ser aplicada de dois modos. Como já mencionamos anteriormente, um sistema baseado em computador interativo é em geral usado para substituir uma atividade manual ou semimanual. Para entender as tarefas que precisam ser realizadas para atingir a meta da atividade, um engenheiro humano[5] precisa entender as tarefas que os seres humanos realizam correntemente (quando usam uma abordagem manual) e depois mapear essas tarefas em um conjunto de tarefas semelhantes (mas não necessariamente idênticas), que são implementadas no contexto da interface com o usuário. Como alternativa, o engenheiro de negócios pode estudar uma especificação existente para uma solução baseada em computador e obter um conjunto de tarefas do usuário que vai acomodar o modelo do usuário, o modelo do projeto e a percepção do sistema.

Independentemente da abordagem geral para a análise de tarefas, um engenheiro de negócios deve primeiro definir e classificar as tarefas. Já mencionamos anteriormente que uma abordagem é o refinamento passo a passo. Por exemplo, considere que uma pequena empresa de software queira construir um sistema de projeto apoiado por computador explicitamente para projetistas de interiores. Observando um projetista de interiores trabalhando, o engenheiro nota que o projeto de interiores abrange várias atividades principais: leiaute do mobiliário (observe o caso de uso discutido anteriormente), seleção de tecidos e materiais, seleção de revestimento de parede e janelas, apresentação (para o cliente), orçamento e aquisição. Cada uma dessas tarefas principais pode ser detalhada em subtarefas. Por exemplo, usando informação contida no caso de uso, o leiaute do mobiliário pode ser refinado nas seguintes tarefas: (1) desenho de uma planta baixa com base nas dimensões dos cômodos; (2) colocação de portas e janelas nas posições adequadas; (3a) uso de gabaritos de mobiliário para desenhar o contorno do mobiliário, em escala, na planta baixa; (3b) uso de gabaritos de destaques para desenhar os destaques em escala na planta baixa; (4) deslocamento do contorno do mobiliário e contorno dos destaques para obter melhor localização; (5) rotulação de todos os contornos dos móveis e dos destaques; (6) desenho de dimensões para mostrar a localização; (7) desenho de uma vista em perspectiva para o cliente. Uma abordagem semelhante pode ser usada para cada uma das outras tarefas principais.

As subtarefas de 1 a 7 podem, cada uma, ser adicionalmente refinadas. As subtarefas de 1 a 6 podem ser realizadas manipulando-se informação e realizando ações dentro da interface com o usuário. Por outro lado, a subtarefa 7 pode ser realizada automaticamente por software, resultando em pouca interação direta com o usuário[6]. O modelo de projeto da interface deve acomodar cada uma dessas tarefas de um modo que seja consistente com o modelo de usuário (o perfil de um projetista de interiores "típico") e com a percepção do sistema (o que o projetista de interiores espera de um sistema automatizado).

AVISO

Embora o refinamento de objetos seja útil, não deve ser usado como uma abordagem isolada. A opinião do usuário deve ser considerada durante a análise de tarefas.

Refinamento de objetos. Em vez de focalizar as tarefas que o usuário deve realizar, o engenheiro de software examina o caso de uso e outras informações obtidas do usuário e extrai os objetos físicos usados pelo projetista de interiores. Esses objetos são classificados em classes. Atributos de cada classe são definidos, e uma avaliação das ações aplicadas a cada objeto fornece ao projetista uma lista de operações. Por exemplo, o gabarito para mobiliário pode traduzir para uma classe chamada de **Mobiliário** com atributos que poderiam incluir **tamanho**, **forma**, **localização** e outros. O projetista de interiores *selecionaria* o objeto da classe **Mobiliário**, o *moveria* para uma posição na planta baixa (outro objeto nesse contexto), *desenharia* o contorno do mobiliário etc. As tarefas *selecionar, mover* e *desenhar* são operações. O modelo de análise da interface com o usuário não iria fornecer uma implementação literal para cada uma dessas operações. No entanto, à medida que o projeto é refinado, os detalhes de cada operação são definidos.

5 Em muitos casos, as tarefas descritas nesta seção são realizadas por um engenheiro de software. O ideal seria que o profissional tivesse treinamento em engenharia humana e projeto de interface com o usuário.

6 No entanto, esse pode não ser o caso. O projetista de interior desejará especificar a perspectiva para ser desenhada, a escala, o uso de cores e outras informações. O caso de uso relativo ao desenho de vistas em perspectiva forneceria a informação necessária para tratar dessa tarefa.

Análise de fluxo de trabalho. Quando um certo número de diferentes usuários, cada um desempenhando papéis diferentes, faz uso de uma interface com o usuário, é algumas vezes necessário ir além da análise de tarefa e do refinamento de objetos e aplicar *análise de fluxo de trabalho*. Essa técnica permite ao engenheiro de software entender como um processo de trabalho é completado quando muitas pessoas (e papéis) estão envolvidas. Considere uma empresa que pretenda automatizar completamente o processo de receituário e entrega de medicamentos receitados. O processo completo[7] gira em torno de uma aplicação baseada na Web que está acessível a médicos (ou seus assistentes), farmacêuticos e pacientes. O fluxo de trabalho pode ser representado efetivamente com um diagrama de raias UML (uma variação do diagrama de atividade).

Consideramos somente uma pequena parte do processo de trabalho: a situação que ocorre quando um paciente solicita reaviamento de receita. A Figura 12.2 apresenta um diagrama de raias que indica as tarefas e decisões para cada um dos três papéis mencionados. Essa informação pode ter sido elicitada por entrevista ou de casos de uso escritos por cada ator. Independentemente disso, o fluxo de eventos (mostrado na figura) habilita o projetista de interface a reconhecer três características-chave da interface:

1. Cada usuário implementa diferentes tarefas pela interface; assim, a aparência da interface projetada para o paciente será diferente daquela definida para farmacêuticos ou médicos.
2. O projeto de interface para farmacêuticos e médicos deve exibir informação de fontes secundárias de informação e acomodar acesso a essas informações (por exemplo, acesso ao estoque para o farmacêutico e acesso à informação sobre medicamentos alternativos para o médico).
3. Muitas das atividades observadas no diagrama de raias podem ser mais bem refinadas usando análise de tarefas e/ou refinamento de objetos (por exemplo, aviar receita pode implicar uma entrega pelo correio, uma visita a uma farmácia, ou uma visita a um centro especial de distribuição de medicamentos).

Representação hierárquica. À medida que a interface é analisada, um processo de refinamento ocorre. Uma vez que o fluxo de trabalho foi estabelecido, uma hierarquia de tarefa pode ser definida para cada tipo de usuário. A hierarquia é derivada por um refinamento passo a passo de cada tarefa identificada pelo usuário. Por exemplo, considere a tarefa de usuário *solicita que uma receita seja reaviada*. A seguinte hierarquia de tarefa é desenvolvida:

Solicita que uma receita seja reaviada

- *Fornece informação de identificação*
  - *Especifica nome*
  - *Especifica ID de usuário*
  - *Especifica PIN e senha*
- *Especifica número da receita*
- *Especifica data em que o reaviamento é solicitado*

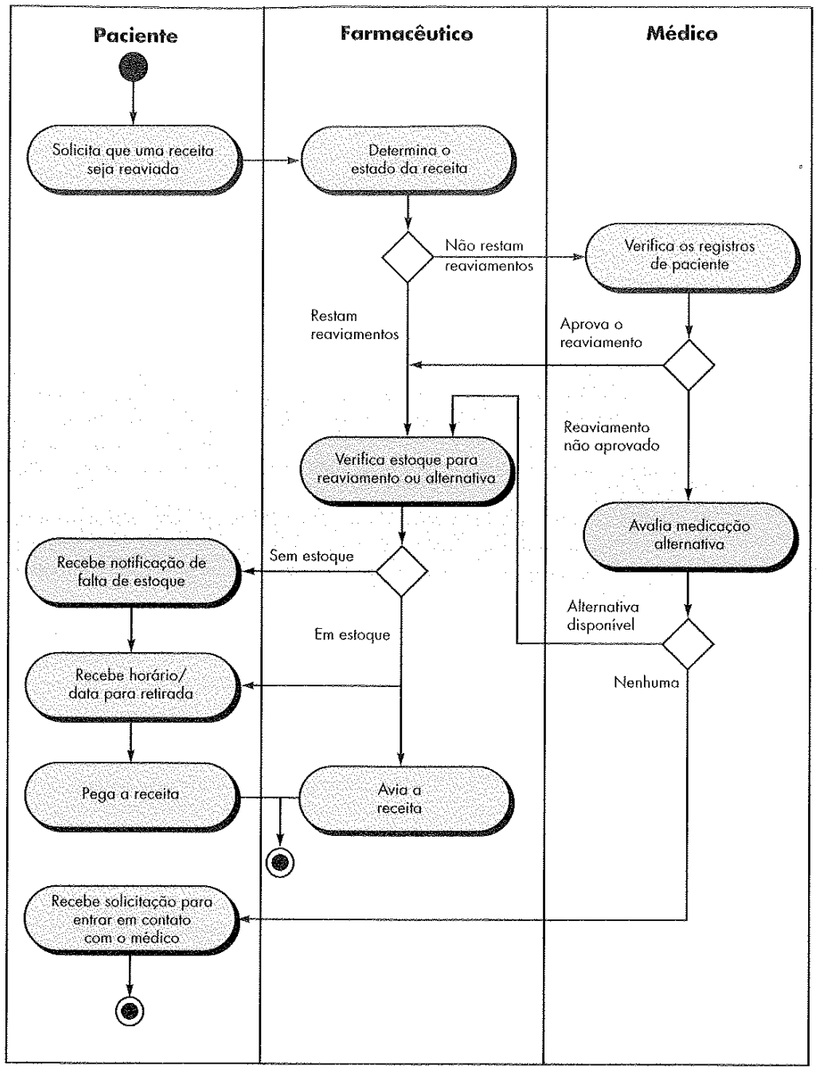
Para completar a tarefa *solicita que uma receita seja reaviada*, três subtarefas são definidas. Uma delas, *fornece informação de identificação*, é mais refinada em três subtarefas adicionais.

> "É muito melhor adaptar a tecnologia para o usuário do que forçar o usuário a se adaptar à tecnologia."
>
> **Larry Marine**

7 Esse exemplo foi adaptado de [HAC98].

Document shared on www.docsity.com

FIGURA 12.2 Diagrama de raias para a função de reaviamento de receita



12.3.3 Análise de Conteúdo de Mostrador

As tarefas do usuário identificadas na seção anterior levam à apresentação de uma variedade de diferentes tipos de conteúdo. Para aplicações modernas, o conteúdo do mostrador pode variar de relatórios baseados em caracteres (por exemplo, uma planilha), mostradores gráficos (por exemplo, um histograma, um modelo 3D, uma fotografia de uma pessoa), ou informação especializada (por exemplo, arquivos de áudio e vídeo). As técnicas de modelagem de análise discutidas no Capítulo 8 identificam objetos de dados de saída produzidos por uma aplicação. Esses objetos de dados podem

ser (1) gerados por componentes (não relacionados à interface) em outras partes da aplicação; (2) adquiridos de dados armazenados em um banco de dados acessível para a aplicação; ou (3) transmitidos de sistemas externos à aplicação em questão.

Durante esse passo de análise de interface são considerados o formato e a estética do conteúdo (como é mostrado pela interface). Entre as questões formuladas e respondidas estão:

? Como determinamos o formato e a estética do conteúdo exibido como parte da IU?

- Os diferentes tipos de dados são atribuídos a localizações geográficas consistentes na tela (por exemplo, fotos sempre aparecem no canto superior do lado direito)?
- O usuário pode personalizar a localização do conteúdo da tela?
- É apropriada a identificação de tela associada a todo o conteúdo?
- Como um relatório grande é dividido para facilitar seu entendimento?
- Haverá mecanismos disponíveis para mover-se diretamente para informação do sumário relativo a grandes coleções de dados?
- A saída gráfica será escalável para caber dentro dos limites do dispositivo de exibição que é usado?
- Como a cor será usada para melhorar o entendimento?
- Como as mensagens de erro e alertas serão apresentadas aos usuários?

À medida que cada uma dessas (e outras) questões é respondida, são estabelecidos os requisitos para a apresentação do conteúdo.

12.3.4 Análise do Ambiente de Trabalho

Hackos e Redish [HAC98] discutem a importância da análise do ambiente de trabalho quando afirmam:

> As pessoas não realizam seu trabalho isoladamente. Elas são influenciadas pelas atividades ao seu redor, pelas características físicas do local de trabalho, pelo tipo de equipamento que estão usando e pelos relacionamentos de trabalho que têm com outras pessoas. Se os produtos que você projetar não se encaixarem no ambiente, eles podem ser de uso difícil ou frustrante.

Em algumas aplicações, a interface com o usuário para um sistema baseado em computador é colocada em uma "posição amigável ao usuário" (iluminação adequada, boa altura de monitor, acesso fácil ao teclado), mas em outros (um chão de fábrica ou a cabine de comando de um avião) a iluminação pode ficar abaixo do ótimo, o barulho pode ser um problema, um teclado ou um mouse podem não ser uma opção, a colocação do monitor pode ser pior que a ideal. O projetista de interface pode ser restringido por fatores que se opõem à facilidade de uso.

Em adição aos fatores ambientais físicos, a cultura do lugar de trabalho também entra em jogo. A interação com o sistema será medida de algum modo (tempo por transação ou precisão de uma transação)? Duas ou mais pessoas terão de compartilhar informação antes que uma entrada possa ser fornecida? Como será fornecido suporte aos usuários do sistema? Essas e muitas questões relacionadas devem ser respondidas antes de iniciar o projeto de interface.

12.4 PASSOS DO PROJETO DA INTERFACE

Uma vez completada a análise da interface, todas as tarefas (ou objetos e ações) requeridas pelo usuário final foram identificadas em detalhes e tem início a atividade de projeto da interface. Projeto de interface, como todo projeto de engenharia de software, é um processo iterativo. Cada passo do projeto de interface com o usuário ocorre um certo número de vezes, cada qual elaborando e refinando informação desenvolvida no passo anterior.

Embora muitos diferentes modelos de projeto de interface com o usuário tenham sido propostos (por exemplo, [NOR86], [NIE00]), todos sugerem alguma combinação dos seguinte passos:

Document shared on www.docsity.com

1. Usando informação desenvolvida durante a análise de interface (veja a Seção 12.3), defina os objetos e ações (operações) de interface.
2. Defina os eventos (ações dos usuários) que vão causar modificação no estado da interface com o usuário. Modele esse comportamento.
3. Represente cada estado da interface como ela realmente será vista pelo usuário final.
4. Indique como o usuário interpreta o estado do sistema por meio da informação fornecida pela interface.

Em alguns casos, o projetista de interface pode começar com esboços de cada estado da interface (isto é, o que o usuário da interface vê em várias circunstâncias) e depois voltar para definir objetos, ações e outras informações importantes de projeto. Indiferentemente da seqüência de tarefas de projeto, o projetista deve (1) sempre seguir as regras de ouro discutidas na Seção 12.1; (2) modelar como a interface será implementada; e (3) considerar o ambiente (tecnologia de exibição, sistema operacional, ferramentas de desenvolvimento) a ser usado.

> "Projeto interativo [é] uma mistura sem separação de artes gráficas, tecnologia e psicologia."
>
> **Brad Wieners**

12.4.1 Aplicação dos Passos do Projeto de Interface

Um importante passo no projeto da interface é a definição dos objetos de interface e das ações que a eles são aplicadas. Para tanto, os casos de uso são analisados de forma bastante semelhante àquela descrita no Capítulo 8. Isto é, é escrita uma descrição do caso de uso. Substantivos (objetos) e verbos (ações) são isolados para criar uma lista de objetos e ações.

Uma vez que os objetos e ações foram definidos e refinados iterativamente, são categorizados por tipo. São identificados objetos-alvo, objetos-fonte e objetos de aplicação. Um *objeto-fonte* (por exemplo, um ícone de relatório) é arrastado ou solto sobre um *objeto-alvo* (por exemplo, um ícone de impressora). A implicação dessa ação é criar um relatório impresso. Um *objeto de aplicação* representa dados específicos da aplicação que não são diretamente manipulados como parte da interação de tela. Por exemplo, uma lista de endereços é usada para guardar nomes para endereçamento. A lista propriamente dita pode ser ordenada, intercalada ou depurada (ações baseadas em menu), mas não é arrastada e solta por intermédio de interação com o usuário.

Quando o projetista está certo de que todos os objetos e ações importantes foram definidos (para uma iteração de projeto), o leiaute de tela é realizado. Como outras atividades de projeto de interface, o *leiaute de tela* é um processo interativo no qual são conduzidos o projeto gráfico e a colocação de ícones, a definição de texto descritivo de tela, a especificação e intitulação de janelas, e a definição dos itens principais e secundários de menu. Se uma metáfora do mundo real é adequada à aplicação, ela é especificada nesse instante e um leiaute é organizado de modo que complemente a metáfora.

Para fornecer uma breve ilustração dos passos de projeto mencionados anteriormente, consideramos um cenário de usuário para o sistema *CasaSegura* (discutido nos capítulos anteriores). Segue um caso de uso preliminar (escrito pelo proprietário) para a interface:

> **Caso de uso preliminar:** Desejo ter acesso ao meu sistema *CasaSegura* de qualquer local remoto, via Internet. Usando um navegador de software operando em meu computador notebook (enquanto estou trabalhando ou viajando), eu posso determinar o estado do sistema de alarme, armar ou desarmar o sistema, reconfigurar zonas de segurança e observar diferentes cômodos da casa pelas câmeras de vídeo pré-instaladas.
>
> Para ter acesso ao *CasaSegura* de um local remoto, forneço uma identificação e uma senha que definem níveis de acesso (por exemplo, nem todos os usuários podem estar aptos a reconfigurar o sistema) e fornecem segurança. Uma vez validado, posso verificar o estado do sistema e modificá-lo, armando ou desarmando o *CasaSegura*. Posso reconfigurar o sistema exibindo uma planta baixa da casa, observando cada um dos sensores de segurança, exibindo cada zona presentemente configurada e modificando-as conforme achar necessário. Posso observar o interior da casa por câmeras de vídeo estrategicamente colocadas. Posso direcionar, afastar e aproximar o foco de cada câmera para obter diferentes visões do interior.

Com base nesse caso de uso, as seguintes tarefas, objetos e itens de dados do proprietário são identificados:

- obter *acesso* ao sistema *CasaSegura*
- *dar entrada* a uma **ID** e **senha** para conseguir acesso remoto
- *verificar* **estado do sistema**
- *armar* ou *desarmar* o sistema *CasaSegura*
- *exibir* **planta baixa** e **localização dos sensores**
- *exibir* **zonas** na planta baixa
- *modificar* **<>zonas** na planta baixa
- *exibir* **localização das câmeras de vídeo** na planta baixa
- *selecionar* **câmera de vídeo** para observação
- *observar* **imagens de vídeo**
- *direcionar* ou *aproximar* **o foco da câmera de vídeo**

AVISO

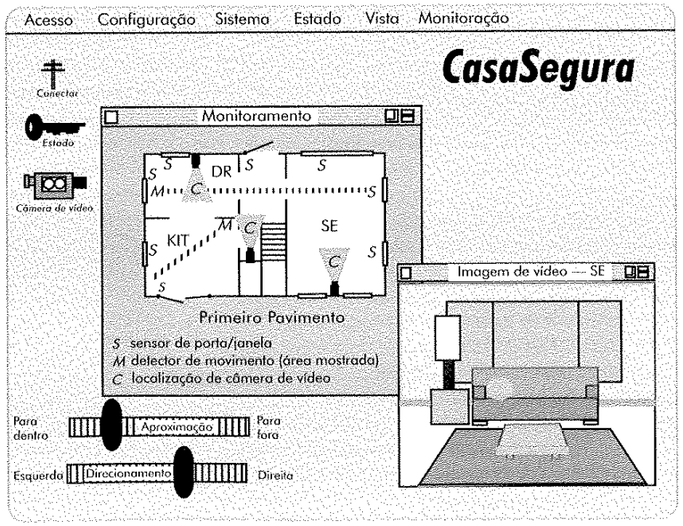
Embora ferramentas automatizadas possam ser úteis no desenvolvimento de protótipos de leiaute, algumas vezes um lápis e papel é tudo o que é necessário.

Objetos (em negrito) e ações (em itálico) são extraídos dessa lista de tarefas do proprietário. A maioria dos objetos mencionados é objeto de aplicação. No entanto, a **localização de câmera de vídeo** (objeto-fonte) é arrastada e solta sobre a **câmera de vídeo** (objeto-alvo) para criar uma **imagem** (uma janela que mostra o vídeo).

Um esboço preliminar do leiaute de tela para o monitoramento por vídeo é criado (Figura 12.3).[8] Para acionar a imagem de vídeo, é selecionado um ícone de localização de câmera de vídeo, C, localizado na planta baixa exibida na janela de monitoramento. Nesse caso, uma localização de câmera na sala de estar, SE, é então arrastada e solta sobre o ícone câmera de vídeo na porção superior esquerda da tela. A janela de imagem de vídeo aparece, exibindo imagens oriundas da câmera localizada na sala de estar (SE). As alavancas de controle de aproximação (*zoom*) e direcionamento

FIGURA 12.3

Leiaute de tela preliminar



8 Observe que difere um pouco da implementação dessas características nos capítulos anteriores. Isso pode ser considerado um primeiro esboço de projeto e representa uma alternativa que poderia ser considerada.

Document shared on www.docsity.com

(*pan*) são usadas para controlar a ampliação e direção da imagem de vídeo. Para selecionar uma vista de outra câmera, o usuário simplesmente arrasta e solta um ícone diferente de localização de câmera sobre o ícone câmera no canto superior esquerdo da tela.

O esboço de leiaute mostrado teria de ser suplementado com uma expansão de cada item de menu, dentro da barra de menu, indicando que ações estão disponíveis para o modo de monitoração de vídeo (estado). Um conjunto completo de esboços para cada tarefa do proprietário, mencionado no cenário de usuário, seria criado durante o projeto da interface.

Veja na Web

Uma ampla variedade de padrões de projeto de IU tem sido proposta. Para obter informação sobre um grande número de sites de padrões, visite www.hicpatterns.org.

12.4.2 Padrões de Projeto de Interface com o Usuário

Interfaces gráficas com o usuário sofisticadas têm-se tornado tão comuns que uma grande variedade de padrões de projeto de interface com o usuário tem surgido. Como observamos anteriormente neste livro, um padrão de projeto é uma abstração que prescreve uma solução de projeto para um problema de projeto específico e bem delimitado. Cada um dos exemplos de padrão (e todos os padrões dentro de cada categoria) apresentados na moldura poderia também ter um projeto completo no nível de componente, incluindo classes de projeto, atributos, operações e interfaces.

INFO

Padrões de Interface com o Usuário

Centenas de padrões de IU têm sido propostos na última década. Tidwell [TID02] e vanWelie [WEL01] fornecem taxinomias[9] de padrões de projeto de interface com o usuário que podem ser organizadas em dez categorias. Exemplos de padrões dentro de cada uma dessas categorias são apresentados neste quadro.

IU Inteira (*Whole UI*). Fornece diretriz de projeto para estrutura e navegação de mais alto nível.

Padrão: *navegação em alto nível (top-level navigation)*

Descrição sucinta: Fornece um menu de alto nível, freqüentemente acoplado como um logo ou gráfico identificador, que possibilita a navegação direta para qualquer das principais funções do sistema.

Layout de página (*Page layout*). Trata da organização geral das páginas (para sites Web) ou exibições de telas distintas (para aplicações interativas).

Padrão: *pilha de cartão (card stack)*

Descrição sucinta: Tem a aparência de uma pilha de cartões com separadores, cada um selecionável por um clique de mouse e cada um representando subfunções específicas ou categorias de conteúdo.

Formulários e entrada (*Forms and inputs*). Considera uma variedade de técnicas de projeto para entrada completa no nível de formulário.

Padrão: *preencher-os-espaços (fill-in-the-blanks)*

Descrição sucinta: Permite a entrada de dados alfanuméricos em uma "caixa de texto".

Tabelas (*Tables*). Fornece diretriz de projeto para criar e manipular dados tabulares de todas as espécies.

Padrão: *tabela ordenável (sortable table)*

Descrição sucinta: Exibe uma longa lista de registros que podem ser ordenados pela seleção de um mecanismo de comutação para qualquer rótulo de coluna.

Manipulação direta de dados (*Direct data manipulation*). Trata da edição, modificação e transformação de dados.

Padrão: *migalhas de pão (bread crumbs)*

Descrição sucinta: Fornece um caminho de navegação completo quando o usuário está trabalhando com uma hierarquia complexa de páginas ou telas de exibição.

Navegação (*Navigation*). Apóia o usuário na navegação por meio de menus hierárquicos, páginas Web e telas de exibição interativas.

Padrão: *edite-no-lugar (edit-in-place)*

Descrição sucinta: Fornece capacidade simples de edição de texto para certos tipos de conteúdos na localização que é exibida.

Busca (*Searching*). Habilita as buscas por conteúdo específico por meio da informação mantida em um site Web ou contido em depósitos de dados persistentes aos quais se tem acesso via aplicação interativa.

Padrão: *busca simples (simple search)*

Descrição sucinta: Fornece capacidade de pesquisa em um site Web ou em fonte de dados persistentes para um item de dados simples descrito por uma cadeia alfanumérica.

Elementos de página (*Page elements*). Implementa elementos específicos de uma página Web ou tela de exibição.

Padrão: *perito (wizard)*

9 Descrições completas de padrões (com muitos outros padrões) podem ser encontradas em [TID02] e [WEL01].

Descrição sucinta: Conduz o usuário por uma tarefa complexa um passo de cada vez, fornecendo diretriz para o término da tarefa por meio de uma série de exibições de janelas simples.

Comércio eletrônico (*e-commerce*). Específico de site Web, esses padrões implementam elementos recorrentes de aplicações de comércio eletrônico.

Padrão. *carrinho de compras (shopping cart)*

Descrição sucinta: Fornece uma lista de itens selecionados para compra.

Miscelânea (*Miscellaneous*). Padrões que não são fáceis de se encaixar em uma das categorias anteriores. Em alguns casos, esses padrões são dependentes do domínio ou ocorrem somente para classes de usuários específicas.

Padrão: *indicador de progresso (progress indicator)*

Descrição sucinta: Fornece uma indicação de progresso quando uma operação está em curso.

Uma discussão abrangente de padrões de interface com o usuário pode ser encontrada em [DUY02], [BOR01], [WEL01] e [TID02] para mais informações.

12.4.3 Questões de Projeto

À medida que o projeto de uma interface com o usuário evolui, quatro questões comuns de projeto quase sempre aparecem: tempo de resposta do sistema, facilidades de ajuda a usuário, manipulação da informação de erro, e rotulação de comandos. Infelizmente, muitos projetistas não cuidam logo dessas questões no processo de projeto (algumas vezes o primeiro indício de um problema não ocorre até que um protótipo operacional esteja disponível). Iteração desnecessária, atrasos de projeto e frustração do cliente freqüentemente resultam disso. É bem melhor estabelecer cada uma como uma questão do projeto a ser considerada no início do projeto de software, quando as modificações são fáceis e os custos são baixos.

> "Um erro comum que as pessoas fazem quando tentam projetar algo completamente a prova de tudo é subestimar a engenhosidade dos totalmente loucos."
>
> Douglas Adams

Tempo de resposta. Tempo de resposta do sistema é a principal queixa para muitas aplicações interativas. Em geral, o tempo de resposta do sistema é medido a partir do ponto no qual o usuário realiza alguma ação de controle (por exemplo, pressiona a tecla de retorno ou clica no mouse) até que o software responda com a saída ou a ação desejada.

O tempo de resposta do sistema tem duas características importantes: duração e variabilidade. Se a resposta do sistema é muito longa, a frustração e a tensão do usuário são o resultado inevitável. *A variabilidade* refere-se ao desvio do tempo médio de resposta, e de vários modos é a característica mais importante do tempo de resposta. Variabilidade baixa permite ao usuário estabelecer um ritmo de interação, mesmo que o tempo de resposta seja relativamente longo. Por exemplo, é preferível uma resposta de 1 segundo para um comando do que uma resposta que varia de 0,1 a 2,5 segundos. Quando a variabilidade é significativa, o usuário fica sempre em desequilíbrio, pensando sempre que algo "diferente" ocorreu fora de cena.

Facilidades de ajuda. Quase todo usuário de um sistema interativo baseado em computador requer ajuda de vez em quando. Em alguns casos, uma simples questão dirigida a um colega conhecedor pode ser satisfatória. Em outros, uma pesquisa detalhada, em um conjunto de vários volumes de "manuais do usuário", pode ser a única opção. Em muitas situações, no entanto, um software moderno fornece facilidades de ajuda on-line que permitem a um usuário ter uma questão respondida ou resolver um problema sem deixar a interface.

Algumas questões de projeto [RUB88] devem ser tratadas quando uma facilidade de ajuda é considerada:

- A ajuda vai estar disponível para todas as funções do sistema em todos os momentos durante a interação com o sistema? As opções de resposta incluem ajuda para apenas um subconjunto de todas as funções e ações ou ajuda para todas as funções.

Document shared on www.docsity.com

- Como o usuário solicitará ajuda? As opções incluem um menu de ajuda, uma chave especial de função ou um comando de ajuda.
- Como a ajuda será representada? As opções incluem uma janela separada, uma referência a um documento impresso (menos aconselhável), ou uma sugestão de uma ou duas linhas produzidas em uma localização fixa da tela.
- Como o usuário voltará à interação normal? As opções incluem um botão de retorno mostrado na tela, uma chave de função ou uma seqüência de controle.
- Como a informação de ajuda será estruturada? As opções incluem uma estrutura "plana" em que se tem acesso a toda a informação através de uma palavra-chave, uma hierarquia de informação em camadas, que fornece detalhes crescentes à medida que o usuário progride na estrutura, ou o uso de hipertexto.

Manipulação de Erros. Mensagens e alertas de erro são "más notícias" dadas aos usuários de sistemas interativos quando alguma coisa está indo mal. Na pior hipótese, mensagens e alertas de erro levam informação inútil ou confusa e servem apenas para aumentar a frustração do usuário. Há poucos usuários de computador que não tenham encontrado um erro da forma: *"Aplicação XXX está sendo forçada a ser abandonada porque um erro do tipo 1023 foi encontrado"*. Em algum lugar, uma explicação do erro 1023 deve existir; caso contrário, por que os projetistas teriam adicionado a identificação? No entanto, a mensagem de erro não fornece indicação real do que está errado ou de onde pesquisar para obter informação adicional. Uma mensagem de erro apresentada desse modo não faz nada para amenizar a ansiedade do usuário ou ajudar a corrigir o problema.

> "A interface do inferno — 'para corrigir este erro e continuar, digite qualquer número primo de 11 dígitos...'"
>
> **Autor desconhecido**

Em geral, toda mensagem ou alerta de erro produzida por um sistema interativo deveria ter as seguintes características:

? Que características deve ter uma "boa" mensagem de erro?

- A mensagem deve descrever o problema em um jargão que o usuário possa entender.
- A mensagem deve fornecer sugestão construtiva para recuperação do erro.
- A mensagem deve indicar quaisquer conseqüências negativas do erro (por exemplo, arquivos de dados potencialmente corrompidos) de modo que o usuário possa verificar para assegurar-se de que elas não ocorreram (ou corrigi-las se tiverem ocorrido).
- A mensagem deve ser acompanhada por uma indicação audível ou visual. Isto é, um bip pode ser gerado para acompanhar a exibição da mensagem ou a mensagem pode piscar momentaneamente ou ser exibida em uma cor que seja facilmente reconhecível como "cor de erro".
- A mensagem deve ser "não opinativa". Isto é, a redação nunca deve colocar a culpa no usuário.

Como ninguém realmente gosta de más notícias, poucos usuários gostarão de mensagens de erro independentemente de quão bem projetadas elas tenham sido. Mas uma filosofia de mensagens de erro efetiva pode fazer muito para melhorar a qualidade de um sistema interativo e vai reduzir significativamente a frustração do usuário quando problemas efetivamente ocorrerem.

Rotular menu e comando. O comando digitado foi em certo momento o modo mais comum de interação entre o usuário e o sistema de software e era comumente usado para aplicações de todos os tipos. Hoje, o uso de interfaces orientadas a janelas e a apontar e clicar reduziu a necessidade de comandos digitados, mas muitos usuários potenciais continuam a preferir um modo de interação orientado a comandos. Diversos aspectos de projeto surgem quando comandos digitados são previstos como um modo de interação:

- Cada opção de menu terá um comando correspondente?

- Que forma os comandos vão tomar? As opções incluem uma seqüência de controle (por exemplo, alt-P), teclas de função ou uma palavra digitada.
- Quão difícil será aprender e lembrar os comandos? O que pode ser feito se um comando for esquecido?
- Os comandos podem ser personalizados ou abreviados pelo usuário?

Como mencionamos anteriormente, devem ser estabelecidas convenções para o uso de comandos ao longo de todas as aplicações. É confuso e induz a erros um usuário digitar alt-D quando um objeto gráfico deve ser duplicado, em uma aplicação, e alt-D quando um objeto gráfico deve ser removido, em outra. O potencial de erro é óbvio.

Veja na Web

Diretrizes para desenvolvimento de software acessível podem ser encontradas em www-3.ibm.com/able/guidelines/software/accesssoftware.html.

Acessibilidade da aplicação. À medida que as aplicações de computador tornam-se comuns, engenheiros de software devem garantir que o projeto de interface abranja mecanismos que possibilitam fácil acesso para aqueles com necessidades especiais. *Acessibilidade* para os usuários (e engenheiros de software) que podem ser desafiados fisicamente é um imperativo por razões morais, legais e de negócio. Uma variedade de diretrizes de acessibilidade (por exemplo [W3C03]) — muitas projetadas para aplicações Web, mas freqüentemente aplicáveis a todos os tipos de software — fornecem sugestões detalhadas para projetar interfaces que alcançam vários níveis de acessibilidade. Outras (por exemplo, [APP03], [MIC03]) fornecem diretrizes específicas para "tecnologia assistencial" que trata das necessidades dos portadores de deficiência visual, auditiva, motora, de fala e de aprendizado.

Internacionalização. Engenheiros de software e seus gerentes invariavelmente subestimam o esforço e as habilidades necessárias para criar interfaces de usuário que acomodam as necessidades de diferentes locais e línguas. Freqüentemente, interfaces são projetadas para um local e uma língua e, então, deslocadas para a utilização em outros países. O desafio para os projetistas de interface é criar um software "globalizado". Interfaces de usuário devem ser projetadas para acomodar um núcleo genérico de funcionalidade que pode ser entregue para todos aqueles que

FERRAMENTAS DE SOFTWARE

Desenvolvimento de Interface com o Usuário

Objetivo: Essas ferramentas habilitam o engenheiro de software a criar uma IGU com relativamente pouco desenvolvimento de software personalizado. As ferramentas fornecem acesso a componentes reusáveis e tornam a criação de uma interface uma questão de selecionar dentre capacidades predefinidas, montadas usando-se a ferramenta.

Mecânica: Interfaces modernas com o usuário são construídas com um conjunto de componentes reusáveis acopladas com alguns componentes personalizados desenvolvidos para fornecer características especializadas. A maioria das ferramentas de desenvolvimento de interface com o usuário habilita um engenheiro de software a criá-la por meio da capacidade de "arrastar e soltar". O desenvolvedor seleciona dentre várias capacidades predefinidas (construtores de formulários, mecanismos de interação, capacidade de processamento de comando) e coloca essas capacidades no contexto da interface a ser criada.

Ferramentas Representativas[10]

Macromedia Authorware, desenvolvida por Macromedia Inc. (www.macromedia.com/software/), foi projetada para a criação de interfaces e ambientes de ensino a distância. Faz uso de capacidades sofisticadas de construção.

Motif Common Desktop Environment, desenvolvida por The Open Group (www.osf.org/tech/desktop/cde/), é uma interface gráfica com o usuário integrada para sistemas abertos de computadores pessoais. Ela entrega uma única interface gráfica padronizada para a gestão de dados, arquivos e aplicações.

PowerDesigner/PowerBuilder, desenvolvida por Sybase (www.sybase.com/products/internetappdevtools), é um conjunto abrangente de ferramentas CASE que incluem muitas capacidades para projetar e construir IGUs.

10 Os produtos mencionados aqui não representam uma recomendação, mas, em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

Document shared on www.docsity.com

usam o software. As características de localização possibilitam à interface ser personalizada para um mercado específico. Uma variedade de diretrizes de internacionalização (por exemplo, [IBM03]) está disponível para engenheiros de software. Essas diretrizes tratam amplas questões de projeto (leiautes de tela podem ser diferentes em vários mercados) e tópicos discretos de implementação (diferentes alfabetos podem criar requisitos especializados de rotulação e espaçamento). A norma *Unicode* [UNI03] foi desenvolvida para tratar do desafio desanimador de gerir dezenas de linguagens naturais com centenas de símbolos e caracteres.

12.5 Avaliação de Projeto

Uma vez criado um protótipo operacional da interface com o usuário, ele deve ser avaliado para determinar se satisfaz as necessidades do usuário. A avaliação pode se estender por um espectro de formalidade que varia de um teste de uso informal, no qual o usuário fornece realimentação imediata, até um estudo formalmente projetado, que usa métodos estatísticos para a avaliação de questionários preenchidos por uma população de usuários finais.

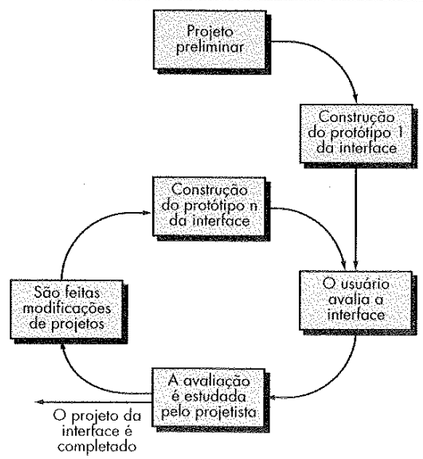
O ciclo de avaliação da interface com o usuário tem a forma mostrada na Figura 12.4. Depois de o modelo de projeto ter sido completado, um protótipo de primeiro nível é criado. O protótipo é avaliado pelo usuário[11], que fornece comentários diretos ao projetista sobre a eficácia da interface. Além disso, se técnicas formais de avaliação são usadas (questionários, folhas de classificação), o projetista pode extrair informação desses dados (por exemplo, 80% de todos os usuários não gostaram do mecanismo para salvar arquivos de dados). Modificações de projetos são feitas com base na entrada fornecida pelo usuário, e é criado o protótipo do próximo nível. O ciclo de avaliação continua até que não sejam mais necessárias modificações no projeto da interface.

A abordagem de prototipagem é efetiva, mas é possível avaliar a qualidade de uma interface com o usuário antes que um protótipo seja construído? Se problemas potenciais puderem ser descobertos e corrigidos logo, o número de voltas pelo ciclo de avaliação e o tempo de desenvolvimento encurtado serão reduzidos. Se um modelo de projeto da interface tiver sido criado, alguns critérios de avaliação [MOR81] podem ser aplicados durante as primeiras revisões de projeto:

1. O tamanho e a complexidade da especificação escrita do sistema e sua interface fornecem uma indicação da quantidade de aprendizado necessária aos usuários do sistema.

Figura 12.4

O ciclo de avaliação do projeto da interface



11 É importante observar que especialistas em ergonomia e projeto de interface podem também conduzir revisões da interface chamadas *avaliações heurísticas ou ensaios cognitivos*.

2. O número especificado de tarefas dos usuários e o número médio de ações por tarefa fornecem uma indicação do tempo de interação e da eficiência global do sistema.
3. O número de ações, tarefas e estados do sistema, apontado pelo modelo de projeto, indica a carga de memória sobre os usuários do sistema.
4. O estilo da interface, facilidades de ajuda e protocolo de manipulação de erros fornecem uma indicação geral da complexidade da interface e do grau em que ela será aceita pelo usuário.

Uma vez construído o primeiro protótipo, o projetista pode coletar uma variedade de dados qualitativos e quantitativos que vai ajudar na avaliação da interface. Para coletar dados qualitativos, questionários podem ser distribuídos aos usuários do protótipo. As questões podem ser todas (1) simples respostas sim/não, (2) respostas numéricas e (3) respostas em escala (subjetiva), (4) escalas de Likert (concordar fortemente, concordar um pouco), (5) respostas em porcentagem (subjetiva), ou (6) em aberto.

Se são desejados dados quantitativos, uma forma de análise de estudo de tempo pode ser conduzida. Os usuários são observados durante a interação e dados — tais como número de tarefas corretamente completadas durante um período de tempo padrão, freqüência de ações, seqüência de ações, tempo gasto "olhando" no monitor de vídeo, número de erros, tempo de recuperação do erro, tempo gasto usando ajuda e número de referências à ajuda por período de tempo padrão — são coletados e usados como diretriz para modificação da interface.

Uma discussão completa de métodos de avaliação de interface com o usuário está além do objetivo deste livro. Para mais informações, consulte [LEA88], [MAN97] e [HAC98].

12.6 Resumo

A interface com o usuário pode ser considerada o elemento mais importante de um sistema ou produto baseado em computador. Se a interface é malprojetada, a habilidade do usuário de extrair todo o poder computacional de uma aplicação pode ficar severamente comprometida. De fato, uma interface fraca pode provocar a falha de uma aplicação que, por outro lado, tenha sido bem projetada e solidamente implementada.

Três importantes princípios pautam o projeto de interfaces efetivas com o usuário: (1) coloque o usuário no controle, (2) reduza a carga de memória do usuário, e (3) torne a interface consistente. Para conseguir uma interface que atenda a esses princípios, um processo de projeto organizado deve ser conduzido.

O desenvolvimento de uma interface com o usuário começa com uma série de tarefas de análise. Essas incluem a identificação do usuário, tarefa e modelagem/análise ambiental. A análise do usuário define o perfil de vários usuários finais e aplica a informação obtida de uma variedade de fontes de negócio e técnicas. Análise de tarefas define as tarefas e ações do usuário por meio de uma abordagem elaborativa ou orientada a objetos, aplicando casos de uso, refinamento de tarefa e objeto, análise de fluxo de trabalho e representações hierárquicas de tarefas para entender completamente a interação humano/computador. Análise ambiental identifica as estruturas físicas e sociais nas quais a interface deve operar.

Uma vez identificadas as tarefas, cenários de uso são criados e analisados para definir um conjunto de objetos e ações da interface. Isso fornece a base para a criação de leiautes de tela, que mostram o projeto gráfico e a colocação de ícones, a definição de texto descritivo de tela, a especificação e títulos de janelas, e a especificação dos itens principais e secundários do menu. Os aspectos de projeto, como tempo de resposta, estrutura de comando e ação, manipulação de erros e facilidades de ajuda, são considerados à medida que o modelo de projeto é refinado. Uma variedade de ferramentas de implementação é usada para construir um protótipo para avaliação pelo usuário.

Document shared on www.docsity.com

Referências Bibliográficas

[APP03] Apple Computer, *People with Special Needs*, 2003, disponível em: http://www.apple.com/disability/.

[BOR01] Borchers, J., *A Pattern Approach to Interaction Design*, Wiley, 2001.

[CON95] Constantine, L, "What DO Users Want? Engineering Usability in Software". *Windows Tech Journal*, dez. 1995, disponível em: http://www.forUse.com.

[DON99] Donahue, G., Weinschenck, S. e Nowicki, J., "Usability Is Good Business". Compuware Corp., jul. 1999, disponível em: http://www.compuware.com.

[DUY02] vanDuyne, D., Landay, J., Hong J., *The Design of Sites*, Addison-Wesley, 2002.

[HAC98] Hackos, J., Redish, J., *User and Task Analysis for Interface Design*, Wiley, 1998.

[IBM03] IBM, *Overview of Software Globalization*, 2003, disponível em: http://oss.software.ibm.com/icu/userguide/i 18n.html.

[LEA88] Lea, M., "Evaluating User Interface Designs". User *Interface Design for Computer Systems*, Halstead Press (Wiley), 1988.

[MAN97] Mandel, T., *The Elements of User Interface Design*, Wiley, 1997.

[MIC03] *Microsoft Accessibility Technology for Everyone*, 2003, disponível em: http://www.microsoft.com/enable/.

[MON84] Monk, A., *Fundamentals of Human-Computer Interaction*, Academic Press, (ed.), 1984.

[MOR81] Moran, T. P., "The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems". *Intl. Journal of Man-Machine Studies*, v. 15, p. 3-50.

[NIE00] Nielsen, J., *Designing Web Usability*, New Riders Publishing, 2000.

[NOR86] Norman, D. A., "Cognitive Engineering". In: User *Centered Systems Design*, Lawrence Earlbaum Associates, 1986.

[RUB88] Rubin, T., *User Interface Design for Computer Systems*, Halstead Press (Wiley), 1988.

[SHN90] Shneiderman, B., *Designing the User Interface*, 3. ed., Addison-Wesley, 1990.

[TID99] Tidwell, J., "Common Ground: A Pattern Language for HC1 Design". Disponível em: http://www.mit.edu/~jtidwell/interaction_patterns.html, maio 1999.

[TID02] Tidwell, J., "IU Patterns and Techniques". Disponível em: http://time-tripper.com/uipatterns/index.html, maio 2002.

[UNI03] Unicode, Inc., *The Unicode Home Page*, 2003, disponível em: http://www.unicode.org/.

[W3C03] World Wide Web Consortium, *Web Content Accessibility Guidelines*, 2003, disponível em: http://www.w3.org/TR/2003/WD-WCAG20-20030624/.

[WEL01] vanWelie, M., "Interaction Design Patterns", Disponível em: http://www.welie.com/patterns/, 2001.

Problemas e Pontos a Considerar

12.1. Descreva a melhor e a pior interface com que você já trabalhou e critique-as relativamente aos conceitos introduzidos neste capítulo.

12.2. Desenvolva dois princípios de projeto adicionais que "colocam o usuário no controle".

12.3. Desenvolva dois princípios de projeto adicionais que "reduzem a carga de memória do usuário".

12.4. Desenvolva dois princípios de projeto adicionais que "tornem a interface consistente".

12.5. Você foi solicitado a desenvolver um sistema bancário residencial baseado na Web. Desenvolva o modelo de usuário, o modelo de projeto, o modelo mental e um modelo de implementação.

12.6. Realize uma análise de tarefa detalhada para qualquer um dos sistemas listados no Problema 12.5. Use uma abordagem detalhista ou orientada a objetos.

12.7. Acrescente pelo menos cinco questões adicionais à lista desenvolvida para análise de conteúdo da Seção 12.3.3.

12.8. Continuando o Problema 12.6, defina objetos e ações de interface para a aplicação. Identifique cada tipo de objeto.

12.9. Desenvolva um conjunto de leiautes de tela, bem como uma definição dos itens de menu principais e secundários para o sistema que você escolheu no Problema 12.5.

12.10. Desenvolva um conjunto de leiautes de tela com uma definição dos itens de menu principais e secundários para o sistema *CasaSegura*. Você pode escolher uma abordagem diferente daquela mostrada no leiaute de tela da Figura 12.3.

12.11. Descreva sua abordagem para as facilidades de ajuda ao usuário para análise de tarefas que você realizou como parte do Problema 12.5.

12.12. Dê alguns exemplos que ilustram por que a variabilidade do tempo de resposta pode ser um problema.

12.13. Desenvolva uma abordagem que integre automaticamente mensagens de erro e facilidades de ajuda ao usuário. Ou seja, o sistema reconhece automaticamente o tipo de erro e oferece uma janela de ajuda com sugestões para corrigi-lo. Desenvolva um projeto de software razoavelmente completo que considere estruturas de dados e algoritmos adequados.

12.14. Desenvolva um questionário de avaliação de interface que contenha 20 questões genéricas que se aplicariam à maioria das interfaces. Peça a dez colegas de classe que preencham o questionário para um sistema interativo que todos vocês usam. Resuma os resultados e relate-os para sua turma.

Leituras e Fontes de Informação Adicionais

Apesar de seu livro não ser especificamente sobre interfaces humano/computador, muito do que Donald Norman (*The Design of Everyday Things*, reedição, Currency/Double-day, 1990) tem a dizer sobre a psicologia de projeto efetivo se aplica à interface com o usuário. Trata-se de leitura recomendada para quem estiver fazendo profissionalmente projeto de interface de alta qualidade.

Interfaces gráficas com o usuário são onipresentes no mundo moderno da computação. Se a interface é usada para uma ATM, um telefone móvel, um PDA, um site Web ou uma aplicação de negócio, a interface com o usuário fornece uma janela para o software. É por essa razão que livros dedicados ao projeto de interface aumentam. Galitz (*The Essential Guide to User Interface Design*, Wiley, 2002), Cooper (*About Face 2.0: The Essentials of User Interface Design*, IDG Books, 2003), Beyer e Holtzblatt (*Contextual Design: A Customer Centered Approach to Systems Design*, Morgan-Kaufmann, 2002), Raskin (*The Humane Interface*, Addison-Wesley, 2000), Constantine e Lockwood (*Software for Use*, ACM Press, 1999), Mayhew (*The Usability Engineering Lifecycle*, Morgan-Kaufmann, 1999) discutem usabilidade, conceitos, princípios e técnicas de projeto de interface com o usuário e apresentam muitos exemplos úteis.

Johnson (*GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers*, Morgan-Kaufmann, 2000) fornece diretrizes úteis para aqueles que aprendem mais efetivamente pelo exame de contra-exemplos. Um livro muito agradável de Cooper (*The Inmates Are Running the Asylum*, Sams Publishing, 1999) discute por que produtos de alta tecnologia nos deixam loucos e como projetar os que não nos deixem.

Análise e modelagem de tarefas são atividades centrais de projeto. Hackos e Redish [HAC98] escreveram um livro dedicado a esses assuntos e fornecem um método detalhado para a abordagem de análise de tarefas. Wood (*User Interface Design: Bridging the Gap from User Requirements to Design*, CRC Press, 1997) considera a atividade de análise de interfaces e a transição para as tarefas de projeto.

A atividade de avaliação concentra-se na usabilidade. Livros de Rubin (*Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests*, Wiley, 1994) e Nielson (*Usability Inspection Methods*, Wiley, 1994) cuidam do assunto com considerável detalhe.

Em um livro especial que pode ser de considerável interesse para projetistas de produtos, Murphy (*Front Panel: Designing Software for Embedded User Interfaces*, R&D Books, 1998) fornece diretrizes detalhadas para o projeto de interfaces de sistemas embutidos e trata dos riscos de segurança inerentes a controle, manipulação de maquinaria pesada e interfaces para sistemas médicos ou de transporte. O projeto de interface para produtos embutidos é também discutido por Garrett (*Advanced Instrumentation and Computer I/O Design: Real-Time System Computer Interface Engineering*, IEEE, 1994).

Uma grande variedade de fontes de informação sobre projeto de interface com o usuário está disponível na Internet. Uma lista atualizada de referências relevantes aos assuntos de projeto de interfaces pode ser encontrada no site deste livro: **http://www.mhhe.com/pressman**.

Document shared on www.docsity.com

CAPÍTULO 13

ESTRATÉGIAS DE TESTE DE SOFTWARE

CONCEITOS-CHAVE

critérios de completamento 293
depuração 308
especificação de teste 301
estratégia convencional ... 294
estratégia OO 302
ITG 290
testes alfa e beta 304
teste de integração 297
teste de regressão 300
teste de sistema 305
teste de unidade 295
teste de validação 303
teste fumaça 300
V&V 289

Uma estratégia de teste de software integra métodos de projeto de casos de teste em uma série bem planejada de passos, que resultam na construção bem-sucedida de software. A estratégia fornece um roteiro que descreve os passos a ser conduzidos como parte do teste, quando esses passos são planejados e depois executados, e quanto de esforço, tempo e recursos serão necessários. Assim, qualquer estratégia de teste deve incorporar planejamento de teste, projeto de casos de teste, execução de teste e a resultante coleta e avaliação de dados.

Uma estratégia de testes de software deve ser suficientemente flexível para promover uma abordagem de teste sob medida. Ao mesmo tempo, deve ser suficientemente rígida para promover planejamento razoável e acompanhamento gerencial, à medida que o projeto progride. Shooman [SHO83] discute esses aspectos:

> Sob vários aspectos, o teste é um processo independente e o número de tipos diferentes de teste varia tanto quanto as diferentes abordagens de desenvolvimento. Durante muitos anos, nossa única defesa contra erros de programação era o projeto cuidadoso e a inteligência própria do programador. Estamos agora em uma era em que técnicas modernas de projeto [e revisões técnicas formais] estão nos ajudando a reduzir o número de erros iniciais que são inerentes ao código. Analogamente, diferentes métodos de teste estão começando a se agregar em várias abordagens e filosofias distintas.

PANORAMA

O que é? Software é testado para descobrir erros que foram feitos inadvertidamente no momento em que foi projetado e construído. Entretanto, como os testes são conduzidos? Você deve desenvolver um plano formal para os seus testes? Deve testar o programa inteiro como um todo ou testar apenas uma pequena parte dele? Deve reexecutar testes que já foram conduzidos quando adiciona novos componentes a um sistema grande? Quando o cliente deve ser envolvido? Essas e muitas outras questões são respondidas quando você desenvolve uma estratégia de teste de software.

Quem faz? Uma estratégia de teste de software é desenvolvida pelo gerente do projeto, engenheiros de software e especialistas em teste.

Por que é importante? O teste freqüentemente responde por mais esforço de projeto que qualquer outra atividade de engenharia de software. Se é conduzido ao acaso, tempo é desperdiçado, esforço desnecessário é despendido e, ainda pior, erros se infiltram sem serem descobertos. Assim, seria razoável estabelecer uma estratégia sistemática para o teste de software.

Quais são os passos? O teste começa "no varejo" e progride para o "atacado". Com isso queremos dizer que os primeiros testes focalizam um único componente ou um pequeno grupo de componentes relacionados e são aplicados para descobrir erros nos dados e na lógica de processamento que ficaram encapsulados no(s) componente(s). Depois que componentes são testados eles precisam ser integrados até que o sistema completo seja construído. Nesse ponto, uma série de testes de alto nível é executada para descobrir erros na satisfação dos requisitos do cliente. À medida que os erros são descobertos, eles precisam ser diagnosticados e corrigidos usando um processo que é chamado de depuração.

Qual é o produto do trabalho? Uma Especificação de Teste documenta a abordagem da equipe de software para o teste, definindo um plano que descreve uma estratégia global e um procedimento que define passos específicos de teste e os testes que serão conduzidos.

Como tenho certeza de que fiz corretamente? Revisando a especificação de teste antes do teste, você pode avaliar a completeza dos casos e das tarefas de teste. Um plano e procedimento de teste efetivos vão levar à construção coordenada do software e à descoberta de erros em cada estágio do processo de construção.

Essas "abordagens e filosofias" são o que chamamos de *estratégia*. No Capítulo 14, apresentaremos a tecnologia de teste de software.

13.1 UMA ABORDAGEM ESTRATÉGICA AO TESTE DE SOFTWARE

Teste é um conjunto de atividades que podem ser planejadas antecipadamente e conduzidas sistematicamente. Por essa razão, um gabarito para teste de software — um conjunto de passos no qual podemos incluir técnicas de projeto de casos de teste e métodos de teste específicos — deve ser definido para o processo de software. O teste de sistemas orientados a objetos será discutido no Capítulo 23. Neste capítulo, concentramos nossa atenção na estratégia de teste de software.

Veja na Web

Recursos úteis para teste de software podem ser encontrados em www.mtsu.edu/~storm/.

Algumas estratégias de teste de software têm sido propostas na literatura. Todas fornecem ao desenvolvedor de software um gabarito de teste e as seguintes características genéricas:

- Para realizar teste efetivo, uma equipe de software deve conduzir revisões técnicas formais (Capítulo 26). Ao fazer isso, muitos erros serão eliminados antes do início do teste.
- O teste começa no nível de componente e prossegue "para fora", em direção à integração de todo o sistema baseado em computador.
- Diferentes técnicas de testes são adequadas em diferentes momentos.
- O teste é conduzido pelo desenvolvedor do software e (para projetos grandes) um grupo de teste independente.

O teste e a depuração são atividades diferentes, mas a depuração deve ser acomodada em qualquer estratégia de teste.

Uma estratégia de teste de software deve acomodar testes de baixo nível, que são necessários para verificar se um pequeno segmento de código-fonte foi corretamente implementado, bem como testes de alto nível, que validam as principais funções do sistema com base nos requisitos do cliente. Uma estratégia deve fornecer diretrizes para o profissional e um conjunto de referenciais para o gerente. Como os passos da estratégia de teste ocorrem quando a pressão do prazo de entrega começa a crescer, o progresso deve ser mensurável e os problemas devem aparecer tão cedo quanto possível.

AVISO

Não seja relaxado a ponto de pensar que o teste é uma rede de segurança que pegará todos os erros que ocorrem por causa de práticas inconsistentes de engenharia de software. Não pegará. Enfatize qualidade e detecção de erros ao longo de todo o processo de software.

13.1.1 Verificação e Validação

O teste de software é um elemento de um aspecto mais amplo, que é freqüentemente referido como verificação e validação (V&V). *Verificação* se refere ao conjunto de atividades que garante que o software implementa corretamente uma função específica. *Validação* se refere a um conjunto de atividades diferente que garante que o software construído corresponde aos requisitos do cliente.[1] Boehm [BOE81] diz isso de outro modo:

Verificação: Estamos construindo o produto corretamente?
Validação: Estamos construindo o produto certo?

A definição de V&V engloba muitas das atividades que são abrangidas pela *garantia da qualidade de software* (SQA) e discutidas em detalhe no Capítulo 26.

Verificação e validação abrangem um amplo conjunto de atividades SQA, que inclui revisões técnicas formais, auditoria de qualidade e configuração, monitoramento de desempenho, simulação, estudo de viabilidade, revisão da documentação, revisão da base de dados, análise de algoritmos, teste de desenvolvimento, teste de usabilidade, teste de qualificação e teste de instalação [WAL89]. Apesar de o teste desempenhar um papel extremamente importante em V&V, muitas outras atividades também são necessárias.

1 Deve ser observado que há uma forte divergência de opinião sobre que tipos de teste constituem "validação". Algumas pessoas acreditam que *todo* teste é verificação e que validação é conduzida quando requisitos são revisados e aprovados, e posteriormente, pelo usuário, quando o sistema estiver operacional. Outras pessoas consideram teste unitário e de integração (Seções 13.3.1 e 13.3.2) como verificação e teste de alta ordem (discutidos posteriormente neste capítulo) como validação.

Document shared on www.docsity.com

> "Teste é uma parte inevitável de qualquer esforço responsável para desenvolver um sistema de software."
>
> **William Howden**

O teste oferece efetivamente o último reduto no qual a qualidade pode ser avaliada e, mais pragmaticamente, erros podem ser descobertos. Mas o teste não deve ser encarado como rede de proteção. Como sabiamente se diz, "Você não pode testar a qualidade. Se ela não estiver lá antes de você começar a testar, ela não estará lá quando terminar de testar". A qualidade é incorporada ao software durante o processo de engenharia de software. A aplicação adequada de métodos e ferramentas, revisões técnicas formais efetivas e gerência e medição sólidas, todas levam à qualidade que é confirmada durante o teste.

Miller [MIL77] relaciona o teste de software à garantia de qualidade, afirmando que "a motivação subjacente ao teste de programa é afirmar a qualidade do software com métodos que podem ser aplicados econômica e efetivamente tanto a sistemas de grande porte quanto de pequeno porte".

13.1.2 Organização do Teste de Software

Em cada projeto de software há um conflito de interesses inerente que ocorre quando o teste começa. O pessoal que construiu o software é agora solicitado a testá-lo. Isso parece inócuo em si mesmo; afinal de contas, quem conhece o programa melhor do que seus desenvolvedores? Infelizmente, esses mesmos desenvolvedores têm um interesse oculto em demonstrar que o programa está livre de erros, que funciona de acordo com os requisitos do cliente e que será completado de acordo com o cronograma e dentro do orçamento. Cada um desses interesses se contrapõe a um teste rigoroso.

> "Otimismo é o risco ocupacional da programação; teste é o tratamento."
>
> **Kent Beck**

Do ponto de vista psicológico, a análise e o projeto de software (juntamente com a codificação) são tarefas construtivas. O engenheiro de software analisa, modela e depois cria um programa de computador e sua documentação. Como qualquer construtor, o engenheiro de software está orgulhoso do edifício que foi construído e olha atravessado qualquer um que tenta demoli-lo. Quando o teste começa, há uma tentativa sutil, mas definida, de "quebrar" a coisa que o engenheiro de software construiu. Do ponto de vista do construtor, o teste pode ser considerado (psicologicamente) destrutivo. Assim, o construtor segue suavemente, projetando e executando testes que vão demonstrar que o programa funciona, em vez de descobrir erros. Infelizmente, erros estarão presentes. E, se o engenheiro de software não os achar, o cliente achará!

PONTO CHAVE

Um grupo de teste independente não tem o "conflito de interesses" que os construtores do software podem ter.

Há freqüentemente algumas concepções equivocadas que podem ser erroneamente inferidas dessa discussão: (1) que o desenvolvedor do software não deve fazer nenhum teste; (2) que o software deve ser "jogado por cima do muro", para estranhos que vão testá-lo impiedosamente; (3) que os testadores se envolvem com o projeto apenas quando os passos de teste estão para começar. Cada uma dessas afirmações é incorreta.

O desenvolvedor de software é sempre responsável por testar as unidades individuais (componentes) do programa garantindo que cada uma realiza a função ou exibe o comportamento para o qual foi projetada. Em muitos casos, o desenvolvedor também conduz testes de integração — um passo de teste que leva à construção (e teste) da arquitetura completa do software. Apenas depois de a arquitetura do software ser completada, um grupo independente de teste começa a ser envolvido.

AVISO

Se não existe um ITG em sua organização, você terá de assumir o ponto de vista dele. Quando você testar, tente quebrar o software.

O papel do grupo independente de teste (*independent test group*, ITG) é remover os problemas inerentes associados com deixar o construtor testar a coisa que ele construiu. O teste independente remove o conflito de interesses que pode, de outra forma, estar presente. Afinal de contas, o pessoal de ITG é pago para encontrar erros.

No entanto, o engenheiro de software não entrega o programa ao ITG e se retira. O desenvolvedor e o ITG trabalham juntamente durante um projeto de software para garantir que testes rigorosos serão conduzidos. Durante a condução do teste, o desenvolvedor deve estar disponível para corrigir os erros eventualmente descobertos.

> "O primeiro erro que o pessoal comete é pensar que a equipe de teste é responsável pela garantia da qualidade."
>
> **Bryan Murich**

O ITG é parte da equipe do projeto de desenvolvimento de software no sentido de que é envolvido durante a análise e o projeto e continua envolvido (planejando e especificando procedimentos de teste) ao longo de um projeto de grande porte. No entanto, em muitos casos, o ITG pertence à organização de garantia da qualidade de software, alcançando assim um grau de independência que poderia não ser possível se ele fizesse parte da organização de engenharia de software.

13.1.3 Uma Estratégia de Teste de Software para Arquiteturas de Software Convencionais

O processo de engenharia de software pode ser visto como a espiral ilustrada na Figura 13.1. Inicialmente, a engenharia de sistemas define o papel do software e leva à análise dos requisitos de software, em que são estabelecidos o domínio da informação, a função, o comportamento, o desempenho, as restrições e os critérios de validação para o software. Movendo-se para dentro, ao longo da espiral, chegamos ao projeto e finalmente à codificação. Para desenvolver software de computador, nos movemos em espiral para dentro, ao longo de voltas que diminuem o nível de abstração em cada volta.

? Qual é a estratégia global para teste de software?

Uma estratégia para teste de software pode também ser vista no contexto da espiral (Figura 13.1). O *teste de unidade* começa no centro da espiral e se concentra em cada unidade (por exemplo, componente) do software como implementado em código-fonte. O teste progride movendo-se para fora ao longo da espiral para o *teste de integração*, em que o foco fica no projeto e na construção da arquitetura do software. Dando outra volta para fora, pela espiral, encontramos o *teste de validação*, em que os requisitos estabelecidos como parte da análise dos requisitos do software são validados em contraste com o software que acabou de ser construído. Finalmente, chegamos ao *teste de sistema*, em que o software e os outros elementos do sistema são testados como um todo. Para testar o software de computador, nos movemos pela espiral para fora ao longo de voltas que ampliam o escopo do teste a cada volta.

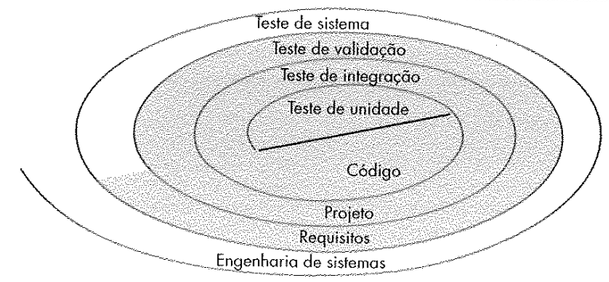
Veja na Web

Recursos úteis para testadores de software podem ser encontrados em www.SQAtester.com.

Considerando o processo de um ponto de vista procedimental, o teste no contexto da engenharia de software é na realidade uma série de quatro passos, que são implementados seqüencialmente. Os passos são mostrados na Figura 13.2. Inicialmente, o teste focaliza cada componente individualmente, garantindo que ele funciona adequadamente como uma unidade. Daí o nome de *teste de unidade*. Este teste faz uso intensivo das técnicas de teste que exercitam caminhos específicos na estrutura de controle de um componente, para garantir completa cobertura e máxima detecção de erros. Em seguida, os componentes devem ser montados ou integrados para formar o pacote de software completo. O teste de integração cuida dos tópicos associados com os problemas duais de verificação e construção de programas. As técnicas de projeto de casos de teste que enfocam as entradas e saídas são mais prevalentes durante a integração, apesar de as técnicas que exercitam caminhos específicos do programa poderem ser usadas para garantir a cobertura dos principais caminhos de controle. Depois que o software tiver sido integrado (construído), um conjunto de testes de alto nível é conduzido. Critérios de validação (estabelecidos durante a análise de

FIGURA 13.1

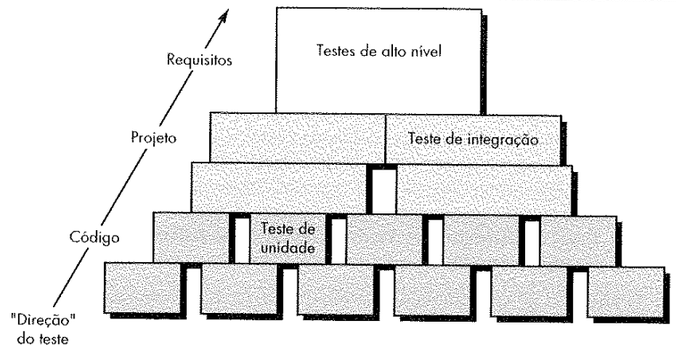
Estratégia de teste



Document shared on www.docsity.com

FIGURA 13.2

Passos de teste de software



requisitos) precisam ser avaliados. Os testes de validação fornecem garantia final de que o software satisfaz a todos os requisitos funcionais, comportamentais e de desempenho. O último passo de teste de alto nível avança fora dos limites da engenharia de software no contexto mais amplo da engenharia de sistemas de computação. O software, uma vez validado, deve ser combinado com os outros elementos do sistema (por exemplo, o hardware, o pessoal, os bancos de dados). O teste de sistema verifica se todos os elementos combinam adequadamente e se a função/desempenho global do sistema é alcançada.

13.1.4 Uma Estratégia de Teste de Software para Arquiteturas Orientadas a Objetos

O teste de sistemas orientados a objetos apresenta um conjunto diferente de desafios para o engenheiro de software. A definição de teste deve ser ampliada para incluir técnicas de descobrimento de erros (por exemplo, revisões técnicas formais) que são aplicadas a modelos de análise e projeto. A completeza e a consistência de representações orientadas a objetos precisam ser avaliadas à medida que elas são construídas. Teste de unidade perde um tanto do seu significado e estratégias de integração mudam significativamente. Em resumo, tanto as estratégias de teste quanto as táticas de teste (Capítulo 14) devem levar em conta as características singulares de software orientado a objetos.

PONTO CHAVE

Como o teste convencional, o teste OO começa "no varejo". No entanto, na maioria dos casos o menor elemento testado é uma classe ou pacote de classes que colaboram.

A estratégia global para software orientado a objetos é idêntica em filosofia à que é aplicada a arquiteturas convencionais, mas difere na abordagem. Começamos com o "teste no varejo" e trabalhamos para fora em direção ao "teste por atacado". No entanto, nosso foco quando "testando no varejo" desloca-se de um módulo individual (na visão convencional) para uma classe que abrange atributos e operações e implica comunicação e colaboração. À medida que as classes são integradas em uma arquitetura orientada a objetos, uma série de testes de regressão é feita para descobrir erros devidos a comunicação e colaboração entre classes (componentes) e efeitos colaterais causados pela adição de novas classes (componentes). Finalmente, o sistema como um todo é testado para garantir que erros nos requisitos sejam descobertos.

CASASEGURA

Preparação do Teste

A cena: Escritório de Doug Miller, enquanto o projeto no nível de componente continua e a construção de certos componentes é iniciada.

Os personagens: Doug Miller, gerente de engenharia de software; Vinod, Jamie, Ed e Shakira — membros da equipe de engenharia de software do *CasaSegura*.

A conversa:

Doug: Parece-me que não empregamos tempo suficiente falando sobre teste.

Vinod: Certo, mas todos nós temos estado um pouco ocupados. Além disso, nós estivemos pensando sobre isso... na verdade, mais do que pensando.

Doug (sorrindo): Eu sei... estamos todos sobrecarregados, mas ainda temos de pensar adiante.

Shakira: Eu gosto da idéia de projetar testes de unidade antes de começar a codificar qualquer dos meus componentes, assim, é isso que tenho tentado fazer. Tenho um arquivo bastante grande de testes para fazer quando o código dos meus componentes estiver pronto.

Doug: Esse é um conceito de Programação Extrema [um processo ágil de desenvolvimento de software, veja Capítulo 4]. Não?

Ed: É. Mesmo que não estejamos usando Programação Extrema em si, decidimos que seria uma boa idéia projetar testes de unidade antes de construir o componente — o projeto nos dá toda a informação de que precisamos.

Jamie: Tenho feito a mesma coisa.

Vinod: E eu assumi o papel de integrador, assim, toda vez que alguém me passa um componente, eu o integro e faço uma série de testes de regressão no programa parcialmente integrado. Venho trabalhando para projetar um conjunto de testes adequados para cada função do sistema.

Doug (para Vinod): Com que freqüência você faz os testes?

Vinod: Todo dia... até que o sistema esteja integrado... bem, eu quero dizer, até que o incremento de software que nós planejamos integrar esteja integrado.

Doug: Vocês estão muito adiante de mim!

Vinod (rindo): Antecipação é tudo no negócio de software, chefe.

13.1.5 Critérios para Completamento do Teste

Uma questão clássica é levantada toda vez que o teste de software é discutido: "Quando terminarmos o teste como saberemos que testamos suficientemente?". Infelizmente, não há resposta definitiva para essa questão, mas há algumas respostas pragmáticas e primeiras tentativas de diretrizes experimentais.

? Quando terminamos o teste?

Uma resposta à questão é: Você nunca acaba de testar, a tarefa simplesmente passa de você (o engenheiro de software) para o seu cliente. Cada vez que o cliente/usuário executa um programa de computador, o programa está sendo testado. Esse simples fato enfatiza a importância das outras atividades de garantia de qualidade do software. Outra resposta (um tanto cínica, mas não obstante precisa) é: Você acaba de testar quando o tempo acaba ou quando o dinheiro acaba.

Apesar de poucos profissionais discordarem dessas respostas, um engenheiro de software precisa de critérios mais rigorosos para determinar quando o teste foi suficientemente conduzido. Musa e Ackerman [MUS89] sugerem uma resposta que é baseada em critérios estatísticos: "Não, nós não podemos ficar absolutamente certos de que o software não vai falhar nunca, mas, relativamente a um modelo estatístico, teoricamente bem fundamentado e experimentalmente validado, fizemos testes suficientes quando podemos dizer, com 95% de confiança, que a probabilidade de mil horas de operação de CPU livre de falhas, em um ambiente probabilisticamente definido, é de pelo menos 0,995". Usando modelagem estatística e teoria de confiabilidade de software, podem ser desenvolvidos modelos de falhas de software (descobertos durante o teste) como função do tempo de execução (por exemplo, veja [MUS89], [SIN99] ou [IEE01]).

Coletando métricas durante o teste de software e fazendo uso dos modelos existentes de confiabilidade de software, é possível desenvolver diretrizes significativas para responder à questão: Quando acabamos de testar? Há pouca discussão em relação a se fazer mais trabalho antes que regras quantitativas para teste possam ser estabelecidas, mas as abordagens experimentais que atualmente existem são consideravelmente melhores do que a simples intuição.

13.2 TÓPICOS ESTRATÉGICOS

Posteriormente, neste capítulo, exploraremos uma estratégia sistemática para teste de software. Mas, mesmo a melhor estratégia falhará se uma série de tópicos importantes não for cuidada. Tom Gilb [GIL95] alega que os seguintes tópicos merecem atenção, se uma estratégia de teste de software bem-sucedida tiver de ser implementada:

Document shared on www.docsity.com

? Que diretrizes levam a uma estratégia de teste bem-sucedida?

Especifique os requisitos do produto de um modo quantificável muito antes de o teste começar. Apesar de o objetivo principal do teste ser encontrar erros, uma boa estratégia de teste também avalia outras características de qualidade tais como portabilidade, manutenibilidade e usabilidade (Capítulo 15). Essas devem ser especificadas de um modo que sejam mensuráveis, a fim de que os resultados de teste não sejam ambíguos.

Enuncie explicitamente os objetivos do teste. Os objetivos específicos do teste devem ser enunciados em termos mensuráveis. Por exemplo, a efetividade do teste, a cobertura do teste, o tempo médio entre falhas, o custo de encontrar e consertar defeitos, a densidade restante de defeitos ou a freqüência de ocorrência e o número de horas de trabalho de teste por teste de regressão, todos devem ser enunciados no plano de teste [GIL95].

Entenda os usuários do software e desenvolva um perfil para cada categoria de usuário. Casos de uso que descrevem o cenário de interação para cada classe de usuário podem reduzir o esforço global de teste focalizando o teste no uso real do produto.

Desenvolva um plano de teste que enfatize "teste de ciclo rápido". Gilb [GIL95] recomenda que uma equipe de engenharia de software "aprenda a testar em ciclos rápidos (2% do esforço do projeto) de incrementos de funcionalidade e/ou aperfeiçoamento da qualidade úteis ao cliente, ou pelo menos passíveis de experimentação no campo". A realimentação gerada por esses testes de ciclo rápido pode ser usada para controlar os níveis de qualidade e as correspondentes estratégias de teste.

Construir software "robusto" que é projetado para testar a si próprio. O software deve ser projetado de modo que use técnicas antidefeitos (Seção 13.3.1), isto é, o software deve ser capaz de diagnosticar certas classes de erros. Além disso, o projeto deve acomodar automação de teste e teste de regressão.

Veja na Web

Uma excelente lista de recursos de testes pode ser encontrada em www.io.com/~wazmo/qa/.

Use revisões técnicas formais efetivas como filtro antes do teste. Revisões técnicas formais (Capítulo 16) podem ser tão efetivas quanto o teste na descoberta de erros. Por isso, as revisões podem reduzir a quantidade de esforço de teste que é necessária para produzir software de alta qualidade.

Conduza revisões técnicas formais para avaliar a estratégia de teste e os casos de teste propriamente ditos. As revisões técnicas formais podem descobrir inconsistências, omissões e erros gritantes na abordagem de teste. Isso poupa tempo e também aperfeiçoa a qualidade do produto.

Desenvolva uma abordagem de aperfeiçoamento contínuo para o processo de teste. A estratégia de teste deve ser medida. As métricas coletadas durante o teste devem ser usadas como parte de uma abordagem estatística de controle do processo para o teste de software.

> "Testar apenas com base nos requisitos perceptíveis ao usuário final é como inspecionar um edifício com base no trabalho feito pelo decorador de interiores, em detrimento das fundações, da estrutura e dos encanamentos."
>
> Boris Beizer

13.3 Estratégias de Teste para Software Convencional

Há muitas estratégias que podem ser usadas para teste de software. Em um extremo, uma equipe de software pode esperar até que o sistema esteja totalmente construído e, então, conduzir testes no sistema inteiro na esperança de encontrar erros. Essa abordagem, apesar de atraente, simplesmente não funciona. Ela vai resultar em software defeituoso que desaponta o cliente e o usuário final. No outro extremo, um engenheiro de software pode conduzir testes diariamente, sempre que qualquer parte do sistema seja construída. Essa abordagem, apesar de menos atraente para muitos, pode ser muito efetiva. Infelizmente, a maioria dos desenvolvedores de software hesita em usá-la, o que fazer?

Uma estratégia de teste que é escolhida pela maioria das equipes de software fica entre os dois extremos. Ela adota uma visão incremental do teste, começando com o teste de unidades individuais de programa, avançando para testes projetados a fim de facilitar a integração das unidades e culmina com testes que exercitam o sistema construído. Cada uma dessas classes de teste é descrita nas seções seguintes.

13.3.1 Teste de Unidade

O *teste de unidade* focaliza o esforço de verificação na menor unidade de projeto do software — o componente ou módulo de software. Usando a descrição de projeto no nível de componente como guia, caminhos de controle importantes são testados para descobrir erros dentro dos limites do módulo. A complexidade relativa dos testes e dos erros descobertos é limitada pelo escopo restrito estabelecido para o teste de unidade. O teste de unidade enfoca a lógica interna de processamento e as estruturas de dados dentro dos limites de um componente. Esse tipo de teste pode ser conduzido em paralelo para diversos componentes.

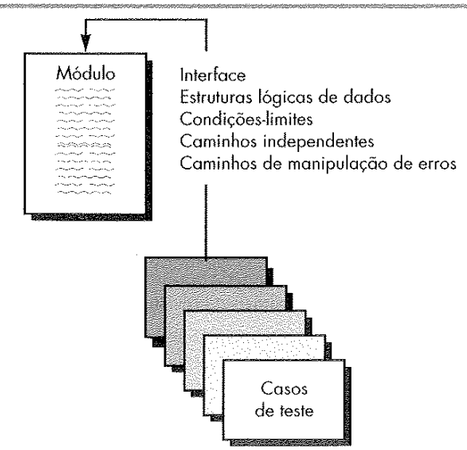
Considerações sobre o teste de unidade. Os testes que ocorrem como parte do teste de unidade são ilustrados esquematicamente na Figura 13.3. A interface do módulo é testada para garantir que a informação flui adequadamente para dentro e para fora da unidade de programa que está sendo testada. A estrutura de dados local é examinada para garantir que os dados armazenados temporariamente mantenham sua integridade durante todos os passos de execução de um algoritmo. Todos os caminhos independentes (caminhos básicos) ao longo da estrutura de controle são exercitados para garantir que todos os comandos de um módulo tenham sido executados pelo menos uma vez. As condições-limite são testadas para garantir que o módulo opere adequadamente nos limiares estabelecidos para limitar ou restringir o processamento. Finalmente, todos os caminhos de manipulação de erros são testados. Testes de fluxo de dados através da interface de um módulo são necessários, antes que qualquer outro teste seja iniciado. Se os dados não entram e saem adequadamente, todos os outros testes são discutíveis. Além disso, as estruturas de dados locais devem ser exercitadas e o impacto local nos dados globais deve ser verificado (se possível) durante o teste de unidade.

? Quais erros são comumente encontrados durante o teste de unidade?

O teste seletivo de caminhos de execução é uma tarefa essencial durante o teste de unidade. Casos de teste devem ser projetados para descobrir erros devidos a cálculos errados, comparações incorretas ou fluxo de controle inadequado. Entre os erros mais comuns no cálculo estão (1) precedência aritmética mal entendida ou incorreta, (2) operações em modo misto, (3) inicialização incorreta, (4) falta de precisão, (5) representação incorreta de uma expressão simbólica. Comparação e fluxo de controle são intimamente acoplados entre si (por exemplo, freqüentemente ocorre mudança de fluxo após uma comparação). Casos de teste devem descobrir erros tais como (1) comparação de tipos de dados diferentes, (2) operadores ou precedência lógica incorretos, (3) expectativa de igualdade quando o erro de precisão torna a igualdade improvável, (4) comparação incorreta de variáveis, (5) terminação de ciclo inadequada ou inexistente, (6) falha na saída quando iteração divergente é encontrada, e (7) variáveis de ciclo inadequadamente modificadas.

Figura 13.3

Teste de unidade



Document shared on www.docsity.com

Veja na Web

Informação útil sobre uma grande variedade de artigos e recursos para "teste ágil" pode ser encontrada em testing.com/agile/.

Teste nos limites é uma das mais importantes tarefas do teste de unidade. O software freqüentemente falha nos seus limites. Isto é, erros ocorrem freqüentemente quando o *n*-ésimo elemento de um vetor de dimensão *n* é processado, quando a *i*-ésima repetição de um ciclo com *i* passagens é chamado, quando o valor máximo ou mínimo permitido é encontrado. Casos de teste que exercitam estruturas de dados, fluxo de controle e valores de dados imediatamente abaixo, imediatamente acima e no máximo e mínimo muito provavelmente descobrirão erros.

O bom projeto determina que condições de erros sejam antecipadas e caminhos de manipulação de erros estabelecidos para redirecionar ou claramente terminar o processamento quando um erro efetivamente ocorre. Yourdon [YOU75] chama essa abordagem de *antidefeitos*. Infelizmente, há uma tendência de incorporar manipulação de erros no software e depois nunca testá-la. Uma história verdadeira pode servir para ilustrar:

> Um sistema de projeto apoiado por computador foi desenvolvido sob contrato. Em um módulo de processamento de transações, um profissional brincalhão colocou a seguinte mensagem de manipulação de erro, depois de uma série de testes condicionais que acionaram vários ramos do fluxo de controle: ERRO! NÃO HÁ JEITO DE VOCÊ CHEGAR AQUI. Essa "mensagem de erro" foi descoberta por um cliente durante o treinamento do usuário!

AVISO

Certifique-se de que você projetou testes para executar cada caminho de manipulação de erro. Se não fizer isso, o caminho pode falhar quando for acionado, exacerbando uma situação já delicada.

Entre os erros potenciais que devem ser testados quando a manipulação de erros é avaliada estão: (1) a descrição do erro é ininteligível; (2) o erro mencionado não corresponde ao erro encontrado; (3) a condição de erro provoca a intervenção do sistema antes da manipulação do erro; (4) o processamento da condição de exceção está incorreto; ou (5) a descrição de erro não fornece informação suficiente para ajudar na localização da causa do erro.

Procedimentos de testes de unidade. O teste de unidade é normalmente considerado um apêndice ao passo de codificação. O projeto de teste de unidade pode ser realizado antes que o código seja iniciado (uma abordagem ágil preferida) ou depois de o código-fonte ter sido gerado. Uma revisão da informação de projeto fornece diretrizes para o estabelecimento de casos de teste que têm probabilidades de descobrir erros de cada uma das categorias anteriormente discutidas. Cada caso de teste deve ser acoplado a um conjunto de resultados esperados.

Como um componente não é um programa isolado, o software para um *pseudocontrolador* (*driver*) e/ou para um *pseudocontrolado* (*stub*) deve ser desenvolvido para cada teste de unidade. O ambiente de teste de unidade é ilustrado na Figura 13.4. Na maioria das aplicações um *pseudocontrolador* nada mais é do que um "programa principal", que aceita dados do caso de teste, passa tais dados ao componente (a ser testado) e imprime os resultados relevantes. Os *pseudocontrolados* servem para substituir módulos que são subordinados (chamados pelo) ao componente a ser testado. Um pseudocontrolado, ou "pseudo subprograma", usa a interface dos módulos subordinados, pode fazer um mínimo de manipulação de dados, fornece verificação da entrada e devolve o controle ao módulo que está sendo testado.

AVISO

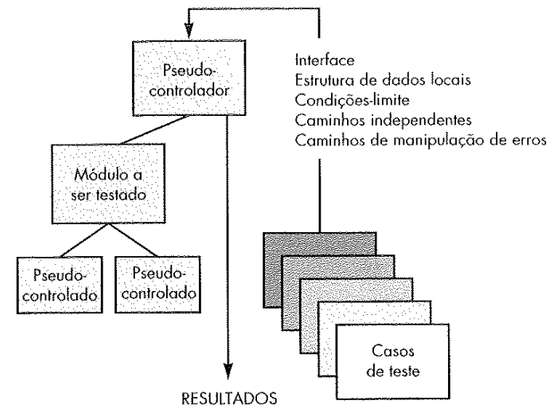
Há algumas situações em que você não terá recursos para fazer teste de unidade abrangente. Selecione módulos críticos e aqueles que têm complexidade ciclomática alta e faça o teste de unidade apenas neles.

Pseudocontroladores e pseudocontrolados representam despesas indiretas. Isto é, ambos são softwares que precisam ser escritos (projeto formal não é comumente aplicado), mas não são entregues com o produto final do software. Se pseudocontroladores e pseudocontrolados são mantidos bem simples, as despesas indiretas reais são relativamente baixas. Infelizmente, muitos componentes não podem ser testados no nível de unidade de modo adequado com software adicional "simples". Em tais casos, o teste completo pode ser adiado até o passo de teste de integração (em que pseudocontroladores ou pseudocontrolados são também usados).

O teste de unidade é simplificado quando um componente com alta coesão é projetado. Quando apenas uma função é implementada por um componente, o número de casos de teste é reduzido e os erros podem ser mais facilmente previstos e descobertos.

FIGURA 13.4

Ambiente de teste de unidade



13.3.2 Teste de Integração

Um novato no mundo do software poderia formular, após todos os módulos terem sido testados no nível de unidade, uma questão, aparentemente legítima: "Se todos eles funcionam individualmente, por que você duvida que vão funcionar quando colocados em conjunto?". O problema, sem dúvida, é "colocá-los juntos" — interfaces. Dados podem ser perdidos através de uma interface; um módulo pode ter um efeito imprevisto ou adverso sobre outro; subfunções, quando combinadas, podem não produzir a função principal desejada; imprecisão aceitável individualmente pode ser amplificada para níveis inaceitáveis; estruturas de dados globais podem apresentar problemas. Infelizmente, a lista prossegue.

Teste de integração é uma técnica sistemática para construir a arquitetura do software enquanto, ao mesmo tempo, conduz testes para descobrir erros associados às interfaces. O objetivo é, a partir de componentes testados no nível de unidade, construir uma estrutura de programa determinada pelo projeto.

AVISO

Adotar a abordagem big-bang para a integração é uma estratégia preguiçosa que está fadada ao fracasso. Integre incrementalmente, testando à medida que progride.

Há freqüentemente uma tendência de tentar integração não-incremental, isto é, construir o programa usando uma abordagem *big-bang*. Todos os componentes são combinados com antecedência. O programa inteiro é testado de uma só vez. E usualmente resulta em caos! Um conjunto de erros é encontrado. A correção é difícil porque o isolamento das causas é complicado pelo vasto espaço do programa inteiro. Uma vez corrigidos esses erros, novos erros aparecem e o processo continua em um ciclo aparentemente interminável.

A integração incremental é a antítese da abordagem *big-bang*. O programa é construído e testado em pequenos incrementos, em que erros são mais fáceis de isolar e corrigir; é mais provável que as interfaces sejam testadas completamente e pode ser aplicada uma abordagem sistemática de teste. Nos parágrafos seguintes, algumas estratégias incrementais de integração diferentes são discutidas.

AVISO

Quando você desenvolve um cronograma detalhado de projeto tem que considerar a maneira pela qual a integração será feita, de modo que os componentes estejam disponíveis quando necessário.

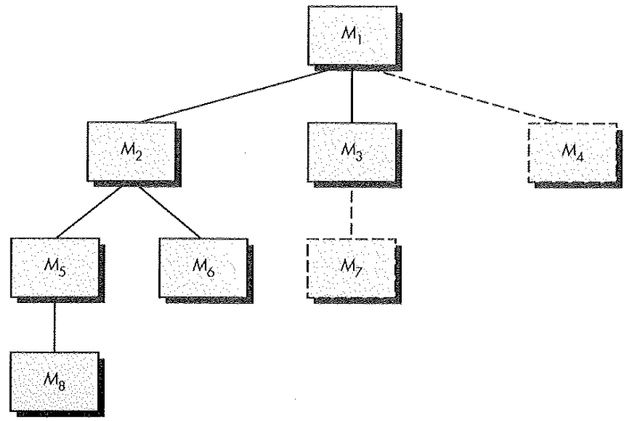
Integração descendente. *O teste de integração descendente* (*top-down*) é uma abordagem incremental para a construção da arquitetura do software. Os módulos são integrados movendo-se descendentemente pela hierarquia de controle, começando com o módulo de controle principal (programa principal). Os módulos subordinados (e os que são subordinados em última instância) ao módulo de controle principal são incorporados à estrutura de maneira primeiro-em-profundidade ou primeiro-em-largura.

Observando a Figura 13.5, a *integração primeiro-em-profundidade* integra todos os componentes, em um caminho de controle principal da estrutura. A seleção de um caminho principal é um tanto arbitrária e depende de características específicas da aplicação. Por exemplo, selecionando o caminho à esquerda, os componentes M_1, M_2 e M_5 seriam integrados primeiro. A seguir, M_8 ou (se necessário para o funcionamento adequado de M_2) M_6 seria integrado. Então os caminhos de

Document shared on www.docsity.com

FIGURA 13.5

Integração descendente



controle central e à direita são construídos. *A integração primeiro-em-largura* incorpora todos os componentes diretamente subordinados em cada nível, movendo-se ao longo da estrutura horizontalmente. Pela figura, os componentes M_2, M_3 e M_4 seriam integrados primeiro. Segue-se o próximo nível de controle, M_5, M_6 etc. O processo de integração é realizado em uma série de cinco passos:

1. O módulo de controle principal é usado como pseudocontrolador do teste, e pseudocontrolados são substituídos por todos os componentes diretamente subordinados ao módulo de controle principal.
2. Dependendo da abordagem de integração selecionada (por exemplo, primeiro-em-profundidade ou largura), os pseudocontrolados subordinados são substituídos, um de cada vez, pelos componentes reais.
3. Testes são conduzidos à medida que cada componente é integrado.
4. Ao término de cada conjunto de testes, outro pseudocontrolado é substituído pelo componente real.
5. O teste de regressão (discutido posteriormente nesta seção) pode ser conduzido para garantir que novos erros não tenham sido introduzidos.

? Quais são os passos para a integração descendente?

O processo continua a partir do passo 2 até que toda a estrutura do programa seja construída.

A estratégia de integração descendente verifica os principais pontos de controle ou decisão logo no início do processo de teste. Em uma estrutura de programa bem fatorada, a tomada de decisão ocorre nos altos níveis da hierarquia e assim é encontrada primeiro. Se existem efetivamente problemas de controle importantes, o pronto reconhecimento é essencial. Se a integração primeiro-em-profundidade é selecionada, uma função completa do software pode ser implementada e demonstrada. Por exemplo, considere uma estrutura clássica de transação (Capítulo 10) na qual uma série complexa de entradas interativas é solicitada, adquirida e validada ao longo de um caminho aferente. O caminho aferente pode ser integrado de modo descendente. Todo o processamento de entrada (para o subseqüente despacho das transações) pode ser demonstrado antes que outros elementos da estrutura tenham sido integrados. A demonstração da capacidade funcional, logo no início, é um fator de confiança, tanto para o desenvolvedor quanto para o cliente.

? Que problemas podem ser encontrados quando a integração descendente é escolhida?

A estratégia descendente parece relativamente não-complicada, mas, na prática, problemas logísticos podem surgir. O mais comum desses problemas ocorre quando o processamento nos níveis baixos da hierarquia é necessário para testar adequadamente os níveis superiores. Pseudocontrolados ficam no lugar dos módulos de baixo nível, no início do teste descendente; assim, nenhum dado significativo pode fluir para cima na estrutura do programa. O testador fica com três escolhas: (1) adiar muitos testes, até que pseudocontrolados sejam substituídos pelos módulos reais; (2) desenvolver pseudocontrolados que realizam funções limitadas, que simulam o módulo real; ou (3) integrar o software de baixo para cima, na hierarquia.

A primeira abordagem (adiar os testes até que pseudocontrolados sejam substituídos pelos módulos reais) faz que percamos algum controle sobre a correspondência entre testes específicos e a incorporação de módulos específicos. Isso pode levar a dificuldades em determinar a causa de erros e tende a violar a natureza altamente restrita da abordagem descendente. A segunda abordagem é trabalhável, mas pode levar a despesas indiretas significativas, na medida em que os pseudocontrolados tornam-se mais e mais complexos. A terceira abordagem, chamada de *teste ascendente* (*bottom-up*), é discutida na próxima seção.

Integração ascendente. *O teste de integração ascendente (bottom-up)*, como o seu nome sugere, inicia a construção e teste de *módulos atômicos* (por exemplo, componentes nos níveis mais baixos da estrutura do programa). Como os componentes são integrados de baixo para cima, o processamento necessário para os componentes subordinados em um determinado nível está sempre disponível e as necessidades de pseudocontrolados é eliminada. Uma estratégia de integração ascendente pode ser implementada com os seguintes passos:

? Quais são os passos para a integração ascendente?

1. Componentes de baixo nível são combinados em agregados (*clusters*, algumas vezes chamados de *construções*), que realizam uma subfunção específica do software.
2. Um pseudocontrolador (*driver*, um programa de controle para o teste) é escrito para coordenar a entrada e a saída do caso de teste.
3. O agregado é testado.
4. Pseudocontroladores (*drivers*) são removidos e agregados são combinados movendo-se para cima na estrutura do programa.

PONTO CHAVE

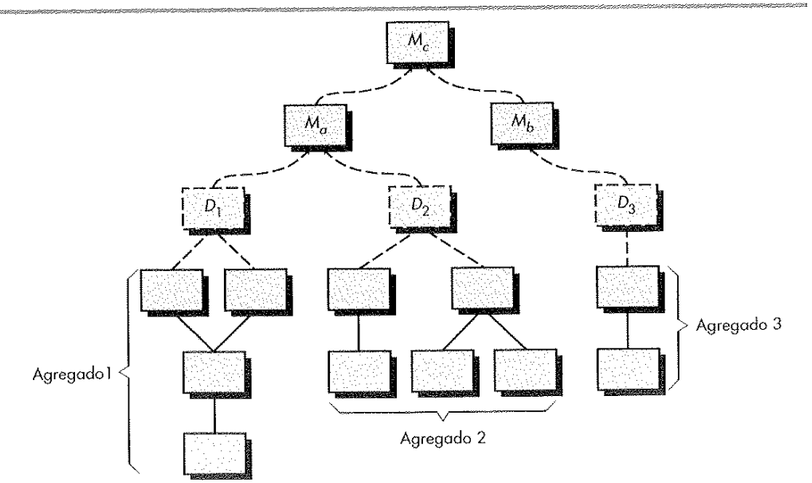
A integração ascendente elimina a necessidade de pseudocontrolados complexos.

A integração segue um padrão ilustrado na Figura 13.6. Os componentes são combinados para formar os agregados (*clusters*) 1, 2 e 3. Cada um dos agregados é testado usando um pseudocontrolador (mostrado como um bloco tracejado). Os componentes nos agregados 1 e 2 são subordinados a M_a. Os pseudocontroladores D_1 e D_2 são removidos e os agregados são interfaceados diretamente com M_a. Analogamente, o pseudocontrolador D_3 para o agregado 3 é removido antes da integração com o módulo M_b. Tanto o M_a quanto o M_b finalmente serão integrados pelo componente M_c e assim por diante.

À medida que a integração se move para cima, a necessidade de pseudocontroladores de teste separados diminui. De fato, se os dois níveis superiores da estrutura do programa são integrados descendentemente, o número de pseudocontroladores pode ser reduzido substancialmente e a integração de agregados é grandemente simplificada.

FIGURA 13.6

Integração ascendente



Document shared on www.docsity.com

AVISO

O teste de regressão é uma estratégia importante para reduzir "efeitos colaterais". Execute testes de regressão toda vez que uma mudança importante é feita no software (inclusive a integração de novos componentes).

Teste de regressão. Cada vez que um novo módulo é adicionado como parte do teste de integração, o software se modifica. Novos caminhos de fluxo de dados são estabelecidos, nova E/S pode ocorrer e nova lógica de controle é acionada. Essas modificações podem causar problemas com funções que previamente funcionavam impecavelmente. No contexto de uma estratégia de teste de integração, o *teste de regressão* é a reexecução de algum subconjunto de testes que já foi conduzido para garantir que as modificações não propagassem efeitos colaterais indesejáveis.

Em um contexto mais amplo, testes bem-sucedidos (de qualquer espécie) resultam na descoberta de erros, e erros precisam ser corrigidos. Sempre que o software é corrigido, algum tópico da configuração do software (o programa, sua documentação ou os dados que a apóiam) é modificado. O teste de regressão é a atividade que ajuda a garantir que modificações (devidas ao teste ou a outras razões) não introduzam comportamento indesejável ou erros adicionais.

O teste de regressão pode ser conduzido manualmente, reexecutando um subconjunto de todos os casos de teste ou usando ferramentas automatizadas de captação/reexecução. *Ferramentas de captação/reexecução* permitem ao engenheiro de software captar casos de teste e resultados para subseqüente reexecução e comparação.

A suíte de teste de regressão (o subconjunto de testes a ser executado) contém três diferentes classes de casos de teste:

- Uma amostra representativa de testes que vão exercitar todas as funções do software.
- Testes adicionais que focalizam funções do software, que serão provavelmente afetadas pela modificação.
- Testes que focalizam os componentes de software que foram modificados.

À medida que o teste de integração prossegue, o número de testes de regressão pode crescer significativamente. Assim sendo, a suíte de testes de regressão deve ser projetada para incluir apenas aqueles testes que cuidam de uma ou mais classes de erros em cada uma das principais funções do programa. Não é prático e eficiente reexecutar cada teste para cada função do programa, toda vez que ocorre uma modificação.

Teste fumaça. *Teste fumaça* é uma abordagem de teste de integração comumente usada quando produtos de software estão sendo desenvolvidos. É projetado como mecanismo de marca-passo para projetos de prazo crítico, permitindo à equipe de software avaliar seu projeto em bases freqüentes. Em essência, a abordagem de teste fumaça abrange as seguintes atividades:

PONTO CHAVE

O teste fumaça pode ser caracterizado como uma estratégia de integração constante. O software é reconstruído (com novos componentes adicionados) e submetido a teste fumaça todos os dias.

1. Componentes de software que foram traduzidos para código são integrados em uma "construção". Uma construção inclui todos os arquivos de dados, bibliotecas, módulos reusáveis e componentes submetidos a engenharia, que são necessários para implementar uma ou mais funções do produto.
2. Uma série de testes é projetada para expor erros que impeçam a construção de desempenhar adequadamente a sua função. O propósito deve ser descobrir erros "bloqueadores" (*show stopper*) que têm a maior probabilidade de colocar o projeto de software fora do cronograma.
3. A construção é integrada com outras construções e o produto inteiro (na sua forma atual) passa pelo teste fumaça *diariamente*. A abordagem de integração pode ser descendente ou ascendente.

A freqüência diária de testar o produto inteiro pode surpreender alguns leitores. No entanto, testes freqüentes dão, tanto aos gerentes quanto aos profissionais, uma avaliação realística do progresso do teste de integração. McConnell [MCO96] descreve o teste fumaça do seguinte modo:

> O teste fumaça deve exercitar o sistema inteiro de ponta a ponta. Ele não precisa ser exaustivo, mas deve ser capaz de expor os problemas principais. O teste fumaça deve ser suficientemente rigoroso para que, se a construção passar, você possa assumir que ela é suficientemente estável para ser testada mais rigorosamente.

O teste fumaça fornece vários benefícios quando é aplicado em projetos de engenharia de software complexos, de prazo crítico:

- *O risco de integração é minimizado.* Como os testes fumaça são conduzidos diariamente, incompatibilidades e outros erros de bloqueio são descobertos logo no início, reduzindo conseqüentemente a probabilidade de impacto sério no cronograma quando erros são descobertos.
- *A qualidade do produto final é aperfeiçoada.* Como a abordagem é orientada para a construção (integração), é provável que o teste fumaça descubra tanto erros funcionais quanto defeitos de projeto arquitetural e no nível de componente. Se esses defeitos são corrigidos no início, o resultado é um produto de melhor qualidade.
- *Diagnóstico e correção de erros são simplificados.* Como em todas as abordagens de teste de integração, os erros descobertos durante o teste fumaça são provavelmente associados com "novos incrementos de software" — isto é, o software que acabou de ser adicionado à(s) construção(ões) é uma causa provável do erro recém-descoberto.
- *Progresso é fácil de avaliar.* A cada dia que passa, mais tem sido integrado ao software e mais tem sido demonstrado que funciona. Isso melhora o moral da equipe e dá aos gerentes uma boa indicação de que está ocorrendo progresso.

> "Trate a construção diária como a batida do coração do projeto. Se não há batida do coração, o projeto está morto."
>
> **Jim McCarthy**

Veja na Web

Links para comentários sobre estratégias de teste podem ser encontrados em www.qalinks.com.

Opções estratégicas. Tem havido muita discussão (por exemplo, [BEI84]) acerca das vantagens e desvantagens relativas do teste de integração descendente *versus* ascendente. Em geral, as vantagens de uma estratégia tendem a resultar em desvantagens para a outra estratégia. A principal desvantagem da abordagem descendente é a necessidade de pseudocontrolados e as dificuldades de teste conseqüentes que podem ser a eles associadas. Os problemas associados com pseudocontrolados podem ser compensados pela vantagem de testar logo as principais funções de controle. A principal desvantagem da integração ascendente é que "o programa, como uma entidade, não existe até que o último módulo seja adicionado" [MYE79]. Essa desvantagem é compensada pelo projeto de casos de teste facilitado e pela ausência de pseudocontrolados.

A seleção de uma estratégia de integração depende das características do software e, algumas vezes, do cronograma do projeto. Em geral, uma abordagem combinada (algumas vezes chamada de *teste sanduíche*), que usa testes descendentes para os níveis mais altos da estrutura do programa, acoplada com testes ascendentes para os níveis subordinados, pode ser o melhor compromisso.

? O que é "módulo crítico" e por que devemos identificá-lo?

À medida que o teste de integração é conduzido, o testador deve identificar os módulos críticos. Um *módulo crítico* tem uma ou mais das seguintes características: (1) aborda vários requisitos do software; (2) tem um alto nível de controle (situa-se em um ponto relativamente alto na estrutura do programa); (3) é complexo ou propenso a erro; ou (4) tem requisitos de desempenho bem definidos. Módulos críticos devem ser testados tão cedo quanto possível. Além disso, testes de regressão devem focalizar função de módulo crítico.

Documentação do teste de integração. Um plano global para integração do software e uma descrição dos testes específicos são documentados em uma *Especificação de Teste*. Esse documento contém um plano de teste e um procedimento de teste, é um produto de trabalho do processo de software e torna-se parte da configuração de software.

O plano de teste descreve a estratégia global de integração. O teste é dividido em fases e construções que tratam de características funcionais e comportamentais específicas do software. Por exemplo, o teste de integração para um sistema CAD pode ser dividido nas seguintes fases de teste:

- Interação com o usuário (seleção de comandos, criação de desenhos, representação de tela, processamento e representação de erros).
- Manipulação e análise de dados (criação de símbolos, dimensionamento, rotação, cálculo de propriedades físicas).
- Processamento e geração de tela (telas bidimensionais e tridimensionais, grafos e mapas).
- Gestão de banco de dados (acesso, atualização, integridade, desempenho).

Document shared on www.docsity.com

Cada uma dessas fases e subfases (denotadas entre parênteses) delineia uma categoria funcional ampla dentro do software e pode geralmente ser relacionada a um domínio específico da estrutura do programa. Conseqüentemente, construções de programa (grupos de módulos) são criadas para corresponder a cada fase. Os seguintes critérios e testes correspondentes são aplicados a todas as fases de teste:

Integridade da interface. As interfaces interna e externa são testadas à medida que cada módulo (ou agrupamento) é incorporado à estrutura.

Validade funcional. Testes projetados para descobrir erros funcionais são conduzidos.

Conteúdo informacional. Testes projetados para descobrir erros associados com estruturas de dados locais ou globais são conduzidos.

Desempenho. Testes projetados para verificar os limites de desempenho estabelecidos durante o projeto do software são conduzidos.

Um cronograma para integração, o desenvolvimento de software de uso geral e tópicos relacionados são também discutidos como parte do plano de teste. Datas iniciais e finais de cada fase são estabelecidas e "janelas de disponibilidade" para módulos submetidos a teste de unidade são definidas. Uma breve descrição do software de uso geral (pseudocontroladores e pseudocontrolados) concentra-se nas características que poderiam requerer esforço especial. Finalmente, o ambiente e recursos de testes são descritos. Configurações de hardware não usuais, simuladores exóticos e ferramentas ou técnicas de teste especiais são alguns de muitos tópicos que também podem ser discutidos.

O procedimento detalhado de teste, que é necessário para realizar o plano de teste, é descrito em seguida. A ordem de integração e os testes correspondentes em cada passo de integração são descritos. Uma lista de todos os casos de teste (anotados para referência subseqüente) e resultados esperados é também incluída.

Um histórico dos resultados reais do teste, problemas ou peculiaridades é registrado no *Relatório de Teste* que pode ser anexado à *Especificação de Teste* se desejado. A informação contida nesta seção pode ser vital durante a manutenção do software. Referências e apêndices adequados são também apresentados.

Como quaisquer outros elementos de uma configuração de software, o formato da especificação de teste pode ser feito sob medida para as necessidades locais de uma organização de engenharia de software. É importante notar, no entanto, que uma estratégia de integração (contida no plano de teste) e detalhes de teste (descritos em um procedimento de teste) são ingredientes essenciais e devem aparecer.

> "O melhor testador não é aquele que encontra mais erros . . . o melhor testador é aquele que corrige a maior parte dos erros."
>
> Cem Kaner *et al.*

13.4 Estratégias de Teste para Software Orientado a Objetos

O objetivo do teste é simplesmente encontrar o maior número possível de erros com uma quantidade de esforço gerenciável aplicada durante um intervalo de tempo realístico. Apesar de esse objetivo fundamental permanecer inalterado para software orientado a objetos, a natureza do software orientado a objetos muda tanto a estratégia de teste quanto a tática de teste (Capítulo 14).

13.4.1 Teste de Unidade no Contexto OO

Quando é considerado o software orientado a objetos, o conceito de unidades se modifica. O encapsulamento guia a definição de classes e objetos. Isso significa que cada classe e cada instância de uma classe (objeto) empacotam os atributos (dados) e as operações (funções) que manipulam esses dados. Uma classe encapsulada é usualmente o foco do teste de unidade. No entanto, operações dentro da classe são a menor unidade testável. Como uma classe pode conter um certo

PONTO CHAVE

O teste de classe para software OO é análogo ao teste de unidade de módulo para o software convencional. Não é aconselhável testar operações isoladamente.

número de operações diferentes, e uma particular operação pode existir como parte de um certo número de classes diferentes, a tática aplicada ao teste de unidade precisa modificar-se. Não podemos mais testar uma única operação isoladamente (visão convencional do teste de unidade), e sim como parte de uma classe. Para ilustrar, considere uma hierarquia de classes na qual uma operação X é definida para a superclasse e é herdada por várias subclasses. Cada subclasse usa a operação X, mas ela é aplicada dentro do contexto dos atributos e operações privadas que foram definidas para a subclasse. Como o contexto no qual a operação X é usada varia de modo sutil, torna-se necessário testar a operação X no contexto de cada uma das subclasses. Isso significa que testar a operação X isoladamente (a abordagem de teste de unidade tradicional) não é eficaz no contexto da orientação a objetos.

O teste de classe para software OO é equivalente ao teste de unidade para software convencional. Diferentemente do teste de unidade do software convencional, que tende a focalizar o detalhe algorítmico de um módulo e os dados que fluem através da interface do módulo, o teste de classe para software OO é guiado pelas operações encapsuladas na classe e pelo estado de comportamento da classe.

13.4.2 Teste de Integração no Contexto OO

Como o software orientado a objetos não tem uma estrutura óbvia de controle hierárquico, as estratégias de integração descendente e ascendente (Seção 13.3.2) têm pouco significado. Além disso, a integração de operações, uma de cada vez em uma classe (abordagem convencional de integração incremental), é freqüentemente impossível por causa das "interações diretas e indiretas dos componentes que constituem a classe" [BER93].

PONTO CHAVE

Uma importante estratégia de teste de integração de software OO é teste baseado no caminho de execução. Caminhos de execução são conjuntos de classes que respondem a uma entrada ou evento. Testes baseados no uso enfocam as classes que não colaboram intensamente com outras classes.

Há duas estratégias diferentes para teste de integração de sistemas OO [BIN94]. A primeira, *teste baseado no caminho de execução (thread-based testing)*, integra o conjunto de classes necessárias para responder a uma entrada ou um evento do sistema. Cada caminho de execução é integrado e testado individualmente. O teste de regressão é aplicado para garantir que nenhum efeito colateral ocorra. A segunda abordagem de integração, *teste baseado no uso (use-based testing)*, começa a construção do sistema testando aquelas classes (chamadas de *classes independentes*) que usam muito poucas (ou nenhuma) classes servidoras. Depois que as classes independentes são testadas, a camada seguinte de classes, chamadas de *classes dependentes*, que usam as classes independentes, são testadas. Essa seqüência de teste de camadas de classes dependentes continua até que todo o sistema seja construído.

O uso de pseudocontroladores e pseudocontrolados também muda quando o teste de integração de sistemas OO é conduzido. Pseudocontroladores podem ser usados para testar operações no mais baixo nível e para testar grupos inteiros de classes. Um pseudocontrolador pode também ser usado para substituir a interface com o usuário de modo que testes da funcionalidade do sistema possam ser conduzidos antes da implementação da interface. Pseudocontrolados podem ser usados em situações nas quais a colaboração entre classes é necessária, mas uma ou mais das classes colaboradoras ainda não foi totalmente implementada.

O *teste de agregado* é uma etapa no teste de integração de software OO. Aqui, um agregado de classes colaboradoras (determinado pelo exame dos modelos CRC e objeto-relacionamento) é exercitado projetando-se casos de teste que tentam descobrir erros nas colaborações.

13.5 Teste de Validação

O *teste de validação* começa no fim do teste de integração, quando componentes individuais já foram exercitados, o software está completamente montado como um pacote, e os erros de interface foram descobertos e corrigidos. Na validação ou no nível de sistema, a distinção entre o software convencional e orientado a objetos desaparece. O teste focaliza ações visíveis ao usuário e saídas do sistema reconhecidas pelo usuário.

Validação pode ser definida de vários modos, mas uma definição simples (no entanto rigorosa) é que ela se torna bem-sucedida quando o software funciona de um modo que pode ser razoavelmente esperado pelo cliente. Nesse ponto, um desenvolvedor de software endurecido pelas batalhas poderia protestar: "Quem ou o quê é o árbitro da razoabilidade das expectativas?".

Document shared on www.docsity.com

PONTO CHAVE

Como todos os outros passos de teste, a validação tenta descobrir erros, mas o foco é no nível de requisitos — em coisas que ficam imediatamente aparentes ao usuário final.

Expectativas razoáveis são definidas nas *Especificações dos Requisitos de Software* — um documento que descreve todos os atributos do software visíveis ao usuário. A especificação contém uma seção chamada de *Critérios de Validação*. A informação contida nessa seção forma a base para a abordagem do teste de validação.

13.5.1 Critérios do Teste de Validação

A validação do software é conseguida por intermédio de uma série de testes que demonstram conformidade com os requisitos. Um plano de teste delineia as classes de teste a ser conduzidas e um procedimento de teste define os casos de teste específicos. Tanto o plano quanto o procedimento são projetados para garantir que todos os requisitos funcionais sejam satisfeitos, todas as características comportamentais sejam conseguidas, todos os requisitos de desempenho sejam alcançados, a documentação esteja correta, usabilidade e outros requisitos sejam satisfeitos (por exemplo, transportabilidade, compatibilidade, recuperação de erro e manutenibilidade).

Depois que cada caso de teste de validação tenha sido conduzido, uma de duas possíveis condições se realiza: (1) a característica de função ou desempenho satisfaz à especificação e é aceita ou (2) é descoberto um desvio da especificação e uma *lista de deficiências* é criada. Desvios ou erros descobertos neste estágio, em um projeto, raramente podem ser corrigidos antes da entrega programada. É freqüentemente necessário negociar com o cliente para estabelecer um método de resolução de deficiências.

13.5.2 Revisão da Configuração

Um importante elemento do processo de validação é a *revisão da configuração*. O objetivo da revisão é garantir que todos os elementos da configuração de software tenham sido adequadamente desenvolvidos, estejam catalogados e tenham os detalhes necessários para apoiar a fase de suporte do ciclo de vida do software. A revisão de configuração, algumas vezes chamada de *auditoria*, é discutida em mais detalhes no Capítulo 27.

13.5.3 Testes Alfa e Beta

É virtualmente impossível para um desenvolvedor de software prever como o cliente usará realmente um programa. As instruções de uso podem ser mal interpretadas; combinações estranhas de dados podem ser usadas regularmente; saída que parecia clara para o testador pode ser ininteligível para um usuário no campo.

Quando um software sob encomenda é construído para um cliente, uma série de testes de aceitação é conduzida para permitir ao cliente validar todos os requisitos. Conduzido pelo usuário final em vez de pelos engenheiros de software, um teste de aceitação pode variar de uma "volta de teste" informal para uma série de testes planejada e executada sistematicamente. De fato, o teste de aceitação pode ser conduzido ao longo de um período de semanas ou meses, descobrindo conseqüentemente erros cumulativos que poderiam degradar o sistema ao longo do tempo.

> "Com olhos suficientes, todos os erros são superficiais (por exemplo, dada uma base de testadores beta e co-desenvolvedores suficientemente grande, quase todo o problema será caracterizado rapidamente e o seu conserto ficará óbvio para alguém)."
>
> E. Raymond

Se o software é desenvolvido como um produto a ser usado por vários clientes, não é prático realizar testes formais de aceitação com cada um. A maioria dos construtores de produtos de software usa os processos chamados de *testes alfa* e *beta* para descobrir erros que apenas o usuário final parece ser capaz de descobrir.

O *teste alfa* é conduzido na instalação do desenvolvedor com os usuários finais. O software é usado em um ambiente natural com o desenvolvedor "olhando sobre o ombro" dos usuários típicos e registrando erros e problemas de uso. Testes alfa são conduzidos em um ambiente controlado.

O *teste beta* é conduzido nas instalações dos usuários finais. Diferente do teste alfa, o desenvolvedor geralmente não está presente. Conseqüentemente, o teste beta é uma aplicação "ao vivo" do software em um ambiente que não pode ser controlado pelo usuário final. O cliente registra todos os problemas (reais ou imaginários) que são encontrados durante o teste beta e os relata ao desenvolvedor em intervalos regulares. Como resultado dos problemas relatados durante os testes beta, os engenheiros de software fazem modificações e depois preparam-se para liberar o produto de software para toda a base de clientes.

CASASEGURA

Preparação para Validação

A cena: Escritório de Doug Miller, à medida que o projeto no nível de componente continua e a construção de certos componentes começa.

Os personagens: Doug Miller, gerente de engenharia de software, Vinod, Jamie, Ed e Shakira — membros da equipe de engenharia de software do *CasaSegura*.

A conversa:

Doug: O primeiro incremento vai ficar pronto para a validação dentro de... por volta de três semanas?

Vinod: É por aí. A integração vai indo bem. Temos feito teste fumaça diariamente, encontramos alguns erros, mas nada que não podemos dar conta. Até agora tudo bem.

Doug: E sobre validação?

Shakira: Bem, nós vamos usar todos os casos de uso como base para o nosso projeto de teste. Ainda não comecei, mas eu permanecerei desenvolvendo teste para todos os casos de uso pelos quais fiquei responsável.

Ed.: O mesmo aqui.

Jamie: Eu também, mas nós temos de juntar nossas ações para o teste de aceitação e também para os testes alfa e beta, não?

Doug: Sim, na verdade eu estive pensando que nós poderíamos trazer alguém de fora para nos ajudar na validação. Eu tenho dinheiro no orçamento... e isso nos daria um novo ponto de vista.

Vinod: Eu acho que nós estamos mantendo o controle.

Doug: Lógico que você está, mas um ITG nos dá uma visão independente do software.

Jamie: Nós estamos apertados de tempo, Doug. Eu, por exemplo, não tenho tempo para servir de babá para quem quer que seja que você traga para fazer o serviço.

Doug: Eu sei, eu sei. Mas um ITG trabalha a partir de requisitos e casos de uso, não vai precisar muito de babá.

Vinod: Eu ainda acho que nós estamos com tudo sob controle.

Doug: Eu estou ouvindo, Vinod, mas vou insistir nisso. Vamos planejar um encontro com o representante do ITG no fim desta semana. Vamos iniciá-los e ver aonde eles chegam.

Vinod: Certo, talvez eles possam aliviar um pouco a carga.

13.6 TESTE DE SISTEMA

No começo deste livro reforçamos o fato de que o software é apenas um elemento de um sistema maior baseado em computador. Em última análise, o software é incorporado aos outros elementos do sistema (por exemplo, hardware, pessoal e informação) e uma série de testes de integração e validação do sistema é conduzida. Esses testes saem do escopo do processo de software e não são conduzidos somente por engenheiros de software. No entanto, passos tomados durante o projeto e o teste de software podem melhorar muito a probabilidade de integração de software bem-sucedida no sistema maior.

> "Como a morte e os impostos, o teste é tanto desagradável quanto inevitável."
>
> Ed Yourdon

Um problema clássico de teste de sistema é ser "dedo-duro". Isso ocorre quando um erro é descoberto, e cada desenvolvedor de um elemento do sistema culpa o outro pelo problema. Em vez de se satisfazer com esse contra-senso, o engenheiro de software deve antecipar problemas potenciais de interface e (1) projetar caminhos de manipulação de erros que testem toda a informação que chega de outros elementos do sistema; (2) conduzir uma série de testes que simule maus dados ou outros erros em potencial na interface do software; (3) registrar os resultados dos testes para usá-

Document shared on www.docsity.com

los como "evidência", se houver dedo-duro; e (4) participar do planejamento e projeto de testes de sistema para garantir que o software seja adequadamente testado.

Teste de sistema é na verdade uma série de diferentes testes cuja finalidade principal é exercitar por completo o sistema baseado em computador. Apesar de cada teste ter uma finalidade distinta, todos trabalham para verificar se os elementos do sistema foram adequadamente integrados e executam as funções a eles alocadas. Nas seções seguintes, discutiremos os tipos de teste de sistema [BEI84] que valem a pena para sistemas baseados em software.

13.6.1 Teste de Recuperação

Muitos sistemas baseados em computador devem se recuperar de falhas e retomar o processamento dentro de um tempo pré-especificado. Em alguns casos, um sistema deve ser *tolerante a falhas*; isto é, falhas de processamento não devem causar a interrupção da função global do sistema. Em outros casos, uma falha do sistema precisa ser corrigida dentro de um período de tempo especificado ou ocorrerá um sério prejuízo econômico.

O *teste de recuperação* é um teste de sistema que força o software a falhar de diversos modos e verifica se a recuperação é adequadamente realizada. Se a recuperação é automática (realizada pelo próprio sistema), a reinicialização, os mecanismos de verificação, a recuperação dos dados e o reinício são avaliados quanto à correção. Se a recuperação requer intervenção humana, o *tempo médio para reparo* (*mean-time-to-repair*, MTTR) é avaliado para determinar se está dentro de limites aceitáveis.

13.6.2 Teste de Segurança

Qualquer sistema baseado em computador que administra informação sensível ou causa ações que podem inadequadamente prejudicar (ou beneficiar) indivíduos é um alvo para invasão imprópria ou ilegal. A invasão abrange um amplo espectro de atividades: *hackers* que tentam invadir os sistemas por esporte; empregados descontentes que tentam invadir por vingança; indivíduos desonestos que tentam invadir em busca de ganhos pessoais ilícitos.

O *teste de segurança* verifica se os mecanismos de proteção incorporados a um sistema vão de fato protegê-lo de invasão imprópria. Para citar Beizer [BEI84]: "A segurança do sistema deve, certamente, ser testada quanto à invulnerabilidade a um ataque frontal — mas também deve ser testada em relação à invulnerabilidade quanto a ataques pelos flancos ou pela retaguarda".

Durante o teste de segurança, o testador desempenha o(s) papel(éis) do indivíduo que deseja invadir o sistema. Vale tudo! O testador pode tentar obter senhas de funcionários externos; pode atacar o sistema com software projetado sob medida para destruir quaisquer defesas que tenham sido construídas; pode tomar o sistema negando conseqüentemente serviço a outros; pode causar erros no sistema de propósito, esperando invadi-lo durante a recuperação; pode percorrer dados inseguros, esperando descobrir a chave para entrada no sistema.

Com tempo e recursos suficientes, o bom teste de segurança vai acabar invadindo o sistema. O papel do projetista do sistema é tornar o custo da invasão maior do que o valor da informação que será obtida.

13.6.3 Teste de Estresse

Os passos de teste de software discutidos anteriormente neste capítulo dão como resultado a rigorosa avaliação das funções e dos desempenhos normais do programa. Os testes de estresse são projetados para submeter programas a situações anormais. Em essência, o testador que realiza o teste de estresse pergunta: "Quanto a gente pode judiar disto antes que falhe?".

O teste de estresse executa um sistema de tal forma que demanda recursos em quantidade, freqüência ou volume anormais. Por exemplo, (1) testes especiais podem ser projetados para gerar dez interrupções por segundo, quando a média é de uma ou duas; (2) a velocidade de entrada de dados pode ser aumentada de uma ordem de grandeza para determinar como as funções de entrada vão reagir; (3) casos de teste que exigem um máximo de memória ou de outros recursos são executados; (4) casos de teste que podem causar problemas de gestão de memória são projetados; (5) casos de teste que podem causar busca excessiva de dados residentes em disco são criados. Em suma, o testador tenta arrebentar o programa.

> "Se você está tentando encontrar defeitos reais no sistema e nunca sujeitou seu software a um teste real de estresse, então é hora de começar."
>
> **Boris Beizer**

Uma variante do teste de estresse é uma técnica chamada de *teste de sensibilidade*. Em algumas situações (a mais comum ocorre em algoritmos matemáticos), um intervalo de dados muito pequeno, contido dentro dos limites de validade dos dados para um programa, pode causar processamento extremo e até errôneo, ou profunda degradação de desempenho. O teste de sensibilidade tenta descobrir combinações de dados, dentro das classes de entrada válidas, que podem causar instabilidade ou processamento inadequado.

13.6.4 Teste de Desempenho

Para sistemas de tempo real e embutidos, o software que fornece a função requisitada, mas não satisfaz aos requisitos de desempenho, é inaceitável. O *teste de desempenho* é projetado para testar o desempenho do software durante a execução, no contexto de um sistema integrado. O teste de desempenho ocorre ao longo de todos os passos do processo de teste. Mesmo no nível de unidade, o desempenho de um módulo individual pode ser avaliado à medida que testes são conduzidos. No entanto, o verdadeiro desempenho de um sistema não pode ser avaliado antes que todos os elementos do sistema estejam plenamente integrados.

Testes de desempenho são freqüentemente acoplados a testes de estresse e usualmente requerem instrumentação, tanto de hardware quanto de software. Ou seja, é freqüentemente necessário medir a utilização de recursos (por exemplo, ciclos de processador) de modo preciso. Instrumentação externa pode monitorar intervalos de execução, registrar eventos (por exemplo, interrupções) à medida que eles ocorrem e retirar amostras do estado da máquina em base regular. Instrumentando um sistema, o testador pode descobrir situações que levam à degradação e possível falha do sistema.

FERRAMENTAS DE SOFTWARE

Planejamento e Gestão de Teste

Objetivo: Essas ferramentas assistem à equipe de software no planejamento da estratégia de teste que é escolhida e na gestão do processo de teste enquanto ele é conduzido.

Mecânica: Ferramentas nesta categoria tratam do planejamento de teste, armazenamento de teste, gestão e controle, monitoramento de requisitos, integração, monitoramento de erros e geração de relatórios. Gerentes de projeto usam-nas para suplementar as ferramentas de cronogramação de projetos. Os testadores usam essas ferramentas para planejar as atividades de teste e para controlar o fluxo de informação enquanto o processo de teste prossegue.

Ferramentas Representativas[2]

OTF (*Object Testing Framework*), desenvolvida por MCG Software, Inc. (www.mcgsoft.com), fornece um arcabouço para gerir seqüências de testes para objetos Smalltalk.

QADirector, desenvolvida por Compuware Corp. (www.compuware.com/qacenter), fornece um único ponto de controle para gerir todas as fases de projeto de testes.

TestWorks, desenvolvida por Software Research, Inc. (www.soft.com/Products/index.html), contém uma seqüência totalmente integrada de ferramentas de teste, inclusive ferramentas para gestão e relatórios de teste.

2 As ferramentas mencionadas aqui não representam uma indicação, mas, em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

Document shared on www.docsity.com

13.7 A Arte da Depuração

O teste de software é uma ação que pode ser sistematicamente planejada e especificada. Projeto de casos de teste pode ser conduzido, uma estratégia pode ser definida e resultados podem ser avaliados com base nas expectativas prescritas.

A depuração ocorre como conseqüência de teste bem-sucedido. Isto é, quando um caso de teste descobre um erro, a depuração é a ação que resulta na reparação do erro. Apesar de a depuração poder e dever ser um processo ordenado, é ainda excessivamente uma arte. Um engênheiro de software, avaliando os resultados de um teste, é freqüentemente confrontado com uma indicação "sintomática" de um problema do software. Isto é, a manifestação externa do erro e a causa interna do erro podem não ter relação óbvia uma com a outra. O processo mental mal compreendido que conecta um sintoma com uma causa é a depuração.

> "Logo que começamos a programar descobrimos, para nossa surpresa, que não era tão fácil obter programas corretos como tínhamos pensado. A depuração tinha de ser descoberta. Posso me lembrar do momento exato em que percebi que uma grande parte da minha vida daí para a frente seria gasta encontrando erros nos meus programas."
>
> Maurice Wilkes descobre a depuração, 1949

13.7.1 O Processo de Depuração

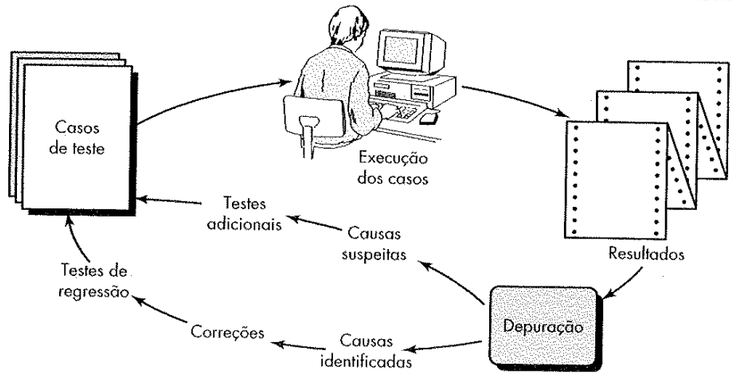
A depuração não é teste, mas sempre ocorre como conseqüência do teste.[3] Com referência à Figura 13.7, o processo de depuração começa com a execução de um caso de teste. Os resultados são avaliados e uma falta de correspondência entre a execução esperada e a obtida é encontrada. Em muitos casos, os dados não correspondentes são um sintoma de uma causa subjacente ainda oculta. A depuração tenta relacionar sintoma com causa, levando assim à correção do erro.

A depuração terá sempre um de dois possíveis resultados: (1) a causa será encontrada e corrigida, ou (2) a causa não será encontrada. No último caso, a pessoa que está realizando a depuração pode suspeitar de uma causa, projetar um caso de teste para ajudá-la a validar aquela suspeita e trabalhar para a correção do erro de um modo interativo.

Por que a depuração é tão difícil? Ao que parece, a psicologia humana (veja a próxima seção) tem mais a ver com uma resposta que a tecnologia de software. No entanto, algumas características de defeitos fornecem algumas pistas:

FIGURA 13.7

O processo de depuração



3 Ao fazer essa afirmação, nós adotamos a mais ampla visão de teste. Não apenas o desenvolvedor testa o software antes da entrega, mas o cliente/usuário testa o software cada vez que ele é usado!

? Por que a depuração é tão difícil?

1. O sintoma e a causa podem ser geograficamente remotos. Isto é, o sintoma pode aparecer em uma parte do programa, enquanto a causa pode ser localizada efetivamente em um lugar que está bastante longe. Estruturas de programa altamente acopladas (Capítulo 11) agravam essa situação.
2. O sintoma pode desaparecer (temporariamente) quando outro erro é corrigido.
3. O sintoma pode, na verdade, ser causado por não-erros (por exemplo, imprecisões de arredondamento).
4. O sintoma pode ser causado por erro humano, que não é facilmente rastreado.
5. O sintoma pode ser resultado de problemas de tempo, em vez de problemas de processamento.
6. Pode ser difícil reproduzir precisamente condições de entrada (por exemplo, uma aplicação em tempo real na qual a ordem das entradas é indeterminada).
7. O sintoma pode ser intermitente. Isso é particularmente comum em sistemas embutidos, que acoplam hardware e software inextricavelmente.
8. O sintoma pode ser devido a causas que estão distribuídas entre várias tarefas sendo executadas em diferentes processadores [CHE90].

Durante a depuração, encontramos erros que vão de levemente incômodos (por exemplo, um formato de saída incorreto) a catastróficos (por exemplo, o sistema falha causando sérios prejuízos econômicos ou físicos). À medida que as conseqüências de um erro aumentam, o volume de pressão para encontrar a causa também aumenta. Freqüentemente, a pressão às vezes força um desenvolvedor de software a consertar um erro e ao mesmo tempo introduzir dois outros.

> "Todos sabem que depurar é duas vezes mais difícil do que escrever um programa inicialmente. Assim, se você for tão esperto quanto puder quando o estiver escrevendo, como vai poder depurá-lo?"
>
> Brian Kernighan

13.7.2 Considerações Psicológicas

Infelizmente, parece haver alguma evidência de que a aptidão para depuração é uma característica humana inata. Algumas pessoas são boas nisso, e outras não são. Apesar de a evidência experimental sobre depuração estar aberta para várias interpretações, grandes variações na habilidade de depuração têm sido relatadas sobre programadores com a mesma educação e experiência.

Comentando sobre os aspectos humanos da depuração, Shneiderman [SHN80] declara:

> A depuração é uma das partes mais frustrantes da programação. Tem elementos de solução de problemas ou de quebra-cabeças, combinados com o reconhecimento desagradável de que você cometeu um erro. A elevada ansiedade e a má vontade em aceitar a possibilidade de erros aumentam a dificuldade da tarefa. Felizmente, há um suspiro de alívio e uma diminuição de tensão quando o defeito é finalmente... corrigido.

Apesar de poder ser difícil "aprender" a depurar, algumas abordagens ao problema podem ser propostas. Nós as examinaremos na seção seguinte.

CASASEGURA

Depuração

A cena: Cubículo de Ed, à medida que a codificação e o teste de unidade é conduzido.

Os personagens: Ed e Shakira, membros da equipe de software do *CasaSegura*.

A conversa:

Shakira (olhando para dentro na entrada do cubículo): Oi! onde você foi na hora do almoço?

Ed: Aqui mesmo... trabalhando.

Shakira: Você parece um trapo... o que aconteceu?

Ed (suspirando alto): Eu venho trabalhando nesse erro de *bleep* desde que eu o descobri às 9h30. E agora são 14h45 e não tenho nenhuma pista.

Shakira: Eu achei que todos nós concordamos em não gastar mais de uma hora depurando coisas que fizemos, depois iríamos pedir ajuda, certo?

Document shared on www.docsity.com

Ed: É, mas...

Shakira (entrando no cubículo): Então, qual é o problema?

Ed: É complicado. E, além disso, eu estive olhando aí por mais de cinco horas. Você não vai achá-lo.

Shakira: Me dá uma chance... qual é o problema?

(Ed explica o problema a Shakira que olha para o programa durante cerca de 30 segundos sem falar.)

Shakira (com um sorriso se formando no rosto): Ah, bem aqui, a variável chamada *armeCondição de Alarme*. Ela não devia ser ajustada para "falso" antes que o ciclo comece?

(Ed olha para a tela sem acreditar, curva a cabeça para a frente e começa a batê-la gentilmente contra o monitor. Shakira, sorrindo amplamente agora, levanta-se e vai embora.)

13.7.3 Abordagens de Depuração

Independentemente da abordagem que é adotada, a depuração tem um objetivo primordial: encontrar e corrigir a causa de um erro de software. O objetivo é realizado por uma combinação de avaliação sistemática, intuição e sorte. Bradley [BRA85] descreve a abordagem de depuração do seguinte modo:

> A depuração é uma aplicação direta do método científico que foi desenvolvido durante 2.500 anos. A base da depuração é localizar a fonte do problema [a causa] por particionamento binário, por meio de hipóteses de trabalho que prevêem novos valores a ser examinados.
>
> Considere um exemplo simples fora da área de software: uma lâmpada da minha casa não funciona. Se nada funciona na casa, a causa pode estar na chave geral ou fora; olho em volta para ver se a vizinhança está às escuras. Ponho a lâmpada suspeita em uma tomada que funcione e um aparelho que funcione no circuito suspeito. E assim prossegue a alternância de hipóteses e testes.

Em geral, três estratégias de depuração foram propostas [MYE79]: (1) força bruta, (2) rastreamento e (3) eliminação de causa. Cada uma dessas estratégias pode ser conduzida manualmente, mas ferramentas modernas de depuração podem tornar o processo muito mais efetivo.

> "O primeiro passo no conserto de um programa com defeito é fazê-lo repetir o erro (no exemplo mais simples possível)."
>
> **T. Duff**

AVISO

Estabeleça um limite de tempo, por exemplo, uma hora, para o prazo que você emprega tentando depurar sozinho um problema. Depois disso, busque ajuda!

Táticas de depuração. A categoria de depuração *força bruta* é provavelmente o método mais comum e menos eficiente para isolar a causa de um erro de software. Aplicamos métodos de depuração de força bruta quando tudo o mais falha. Usando uma filosofia "deixe o computador encontrar o erro", são feitas listagens da memória, são invocados rastreadores da execução e o programa é carregado com comandos de saída. Temos esperança de que em algum lugar no emaranhado de informação que é produzida vamos encontrar uma pista que pode nos levar à causa do erro. Apesar de a massa de informação produzida poder levar finalmente ao sucesso, mais freqüentemente leva à perda de esforço e tempo. Primeiro deve ser empregado o raciocínio!

Rastreamento é uma abordagem de depuração bastante comum que pode ser usada com sucesso em programas pequenos. Começando no lugar em que um sintoma foi descoberto, o código-fonte é rastreado (manualmente) até que o lugar da causa seja encontrado. Infelizmente, à medida que o número de linhas-fonte aumenta, o número de caminhos potenciais para trás pode se tornar inadmissivelmente grande.

A terceira abordagem de depuração — *eliminação de causa* — é manifestada por indução ou dedução e introduz o conceito de *particionamento binário*. Os dados relacionados à ocorrência do erro são organizados para isolar causas em potencial. Uma "hipótese de causa" é concebida e os dados mencionados são usados para provar ou rejeitar a hipótese. Alternativamente, uma lista de todas as causas possíveis é desenvolvida e são conduzidos testes para eliminar cada uma. Se os testes iniciais indicam que uma hipótese particular de causa é promissora, os dados são refinados em uma tentativa de isolar o defeito.

Depuração automatizada. Cada uma dessas abordagens de depuração pode ser complementada com ferramentas de depuração que fornecem apoio semi-automático para o engenheiro de software, à medida que estratégias de depuração são tentadas. Hailpern e Santhanam [HAI02] resumem o estado da arte dessas ferramentas quando afirmam, "... muitas novas abordagens têm sido propostas e muitos ambientes comerciais de depuração estão disponíveis. Ambientes integrados de desenvolvimento (Integrated Development Environments, IDEs) fornecem um modo de captar alguns dos erros predeterminados específicos de linguagens (por exemplo, caracteres em falta no fim de declarações, variáveis indefinidas etc.) sem exigir compilação". Uma área que tem se apossado da imaginação da indústria é a visualização das construções de programa subjacentemente necessárias como meio de analisar um programa (BAE97). Uma grande variedade de compiladores de depuração, ajudas dinâmicas de depuração ("rastreadores"), geradores automáticos de casos de teste e ferramentas de mapeamento de referência cruzada estão disponíveis. No entanto, ferramentas não são substituto para avaliação cuidadosa baseada em um modelo completo de projeto de software e em um código-fonte claro.

FERRAMENTAS DE SOFTWARE

Depuração

Objetivo: Essas ferramentas fornecem apoio automatizado para aqueles que precisam depurar problemas de software. A intenção é fornecer conhecimento que pode ser difícil de obter se o processo de depuração for abordado manualmente.

Mecânica: A maioria das ferramentas de depuração são específicas de linguagem e ambiente de programação.

Ferramentas Representativas[4]:

Jprobe ThreadAnalyzer, desenvolvida por Sitraka (www.sitraka.com), auxilia na avaliação de problemas de trilha — condições de bloqueios, de parada e de corrida que podem colocar várias armadilhas para o desempenho em aplicações Java.

C_Test, desenvolvida por Parasoft (www.parasoft.com), é uma ferramenta de teste de unidade que apóia um conjunto completo de testes em código C e C++. As características de depuração apóiam o diagnóstico de erros que são encontrados.

CodeMedic, desenvolvida por NewPlanet Software (www.newplanetsoftware.com/medic/), fornece uma interface gráfica para o depurador padrão UNIX, gdb, e implementa suas características mais importantes. Gdb atualmente apóia C/++, Java, PalmOS, vários sistemas embutidos, linguagens de montagem, FORTRAN e Modula-2.

BugCollector Pro, desenvolvida por Nesbitt Software Corp. (www.nesbitt.com/), implementa um banco de dados multiusuário que apóia uma equipe de software na monitoração de erros relatados e outras solicitações de manutenção e na gestão de fluxo de trabalho de depuração.

GNATS, uma aplicação freeware (www.gnu.org/software/gnats/), é um conjunto de ferramentas para monitorar relatórios de erros.

O fator pessoal. Qualquer discussão sobre abordagens e ferramentas de depuração fica incompleta sem a menção de um poderoso aliado — outras pessoas! Um novo ponto de vista,[5] não anuviado por horas de frustração, pode fazer maravilhas. Uma máxima final para depuração poderia ser: "Quando tudo mais falha, busque ajuda!".

13.7.4 Correção do Erro

Uma vez encontrado um defeito, ele precisa ser corrigido. No entanto, como já notamos, a correção de um defeito pode introduzir outros erros e, conseqüentemente, fazer mais mal do que bem. Van Vleck [VAN89] sugere três perguntas simples que todo engenheiro de software deve formular antes de fazer a "correção" que remove a causa de um defeito:

4 As ferramentas mencionadas aqui não representam uma indicação, mas, em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

5 O conceito de programação aos pares (recomendada como parte do modelo de processo de Programação Extrema discutido no Capítulo 4) fornece um mecanismo para "depuração" à medida que o software é projetado e codificado.

Document shared on www.docsity.com

? Quando corrijo um defeito, que perguntas devo fazer a mim mesmo?

1. *A causa do defeito está reproduzida em outra parte do programa?* Em muitas situações, um defeito de programa é causado por um padrão de lógica errado, que pode estar reproduzido em outro lugar. Uma consideração explícita do padrão de lógica pode resultar na descoberta de outros erros.
2. *Qual o "próximo defeito" que pode ser introduzido pelo conserto que estou prestes a fazer?* Antes que a correção seja feita, o código-fonte (ou, melhor, o projeto) deve ser avaliado para observar o acoplamento da lógica e das estruturas de dados. Se a correção for feita em uma parte do programa altamente acoplada, cuidado especial deve ser tomado quando qualquer modificação é feita.
3. *O que poderíamos ter feito para prevenir a ocorrência desse defeito?* Essa questão é o primeiro passo em direção ao estabelecimento de uma abordagem estatística de garantia de qualidade de software (Capítulo 26). Se corrigimos o processo, bem como o produto, o defeito será removido do programa em questão e pode ser eliminado de todos os futuros programas.

13.8 Resumo

O teste de software corresponde à mais alta porcentagem de esforço técnico do processo de software. No entanto, estamos apenas começando a entender as sutilezas do planejamento, da execução e do controle sistemático dos testes.

O objetivo do teste de software é descobrir erros. Para alcançar esse objetivo, uma série de passos de testes — testes de unidade, integração, validação e de sistema — é planejada e executada. Os testes de unidade e de integração concentram-se na verificação funcional de um componente e na incorporação de componentes em uma estrutura de programa. Os testes de validação demonstram a rastreabilidade aos requisitos do software e os testes de sistema validam o software depois de ter sido incorporado a um sistema maior.

Cada passo de teste é realizado por uma série sistemática de técnicas de teste que ajudam no projeto de casos de teste. Com cada passo de teste, o nível de abstração no qual o software é considerado é ampliado.

Diferentemente do teste (uma atividade sistemática, planejada), a depuração deve ser vista como uma arte. Começando com uma indicação sintomática do problema, a atividade de depuração deve rastrear a causa de um erro. Dos vários recursos disponíveis durante a depuração, o mais valioso é o conselho de outros membros da equipe de engenharia de software.

A necessidade de software de alta qualidade exige uma abordagem de teste mais sistemática. Para citar Dunn e Ullman [DUN82]:

> O que é necessário é uma estratégia global, cobrindo o espaço estratégico de teste, tão deliberadamente em sua metodologia quanto foi o desenvolvimento sistemático, no qual a análise, o projeto e o código foram baseados.

Neste capítulo examinamos o espaço estratégico de teste, considerando as etapas que têm a maior probabilidade de alcançar o objetivo prevalente do teste: encontrar e remover erros de um modo ordenado e efetivo.

Referências Bibliográficas

[BAE97] Baecker, R., DiGiano, C., e Marcus, A., "Software Visualization for Debugging", *Communications of the ACM*, v. 40, n. 4, abril 1997, p. 44-54, e outros artigos no mesmo número do periódico.

[BEI84] Beizer, B., *Software System Testing and Quality Assurance*, Van Nostrand-Reinhold, 1984.

[BER93] Berard, E., *Essays on Object-Oriented Software Engineering*, v. 1, Addison-Wesley, 1993.

[BIN94] Binder, R., "Testing Object-Oriented Systems: A Status Report", American *Programmer*, v. 7, n. 4, abril 1994, p. 23-28.

[BOE81] Boehm, B., *Software Engineering Economics*, Prentice-Hall, 1981, p. 37.

[BRA85] Bradley, J. H., "The Science and Art of Debugging", *Computerworld*, agos. 19, 1985, p. 35-38.

[CHE90] Cheung, W. H., Black, J. P., e Manning, E. "A Framework for Distributed Debugging", *IEEE Software*, jan. 1990, p. 106-115.

[DUN82] Dunn, R., e Ullman, R., *Quality Assurance for Computer Software*, McGraw-Hill, 1982, p. 158.

[GIL95] Gilb, T., "What We Fail to Do in Our Current Testing Culture", *Testing Techniques Newsletter* (on-line edition, ttn@soft.com), Software Research, jan. 1995.

[HAI02] Hailpern, B. e Santhanam, P., "Software Debugging, Testing and Verification", *IBM Systems Journal*, vol. 41, n. 1, 2002, disponível em http://www.research.ibm.com/journal/sj/411/ hailpern.html.

[IEE01] *Software Reliability Engineering, 12th International Symposium*, IEEE, 2001.

[MCO96] McConnell, S., "Best Practices: Daily Build and Smoke Test", *IEEE Software*, v. 13, n. 4, jul. 1996, p. 143-144.

[MIL77] Miller, E., "The Philosophy of Testing", em *Program Testing Techniques*, IEEE Computer Society Press, 1977, p. 1-3.

[MUS89] Musa, J. D., e Ackerman, A. E., "Quantifying Software Validation: When to Stop Testing?", *IEEE Software*, maio 1989, p. 19-27.

[MYE79] Myers, G., *The Art of Software Testing*, Wiley, 1979.

[SHO83] Shooman, M. L., *Software Engineering*, McGraw-Hill, 1983.

[SHN80] Shneiderman, B., *Software Psychology*, Winthrop Publishers, 1980, p. 28.

[SIN99] Singpurwalla, N., e Wilson, S., *Statistical Methods in Software Engineering: Reliability and Risk*, Springer-Verlag, 1999.

[VAN89] Van Vleck, T., "Three Questions About Each Bug You Find", ACM Software Engineering Notes, v. 14, n. 5, jul. 1989, pp. 62-63.

[WAL89] Wallace, D. R., e Fujii, R. U., "Software Verification and Validation: An Overview", *IEEE Software*, maio 1989, p. 10-17.

[YOU75] Yourdon, E., *Techniques of Program Structure and Design*, Prentice-Hall, 1975.

Problemas e Pontos a Considerar

13.1. Usando suas próprias palavras, descreva a diferença entre verificação e validação. Ambas fazem uso dos métodos de projeto de casos de teste e das estratégias de teste?

13.2. Liste alguns problemas que poderiam ser associados com a criação de um grupo independente de teste. Um grupo ITG e um grupo SQA são constituídos das mesmas pessoas?

13.3. É sempre possível desenvolver uma estratégia para testar software que usa a seqüência de passos de teste descrita na Seção 13.1.3? Que possíveis complicações poderiam surgir em sistemas embutidos?

13.4. Por que um módulo altamente acoplado é difícil de se submeter a teste de unidade?

13.5. O conceito de "antidefeitos" (Seção 13.3.1) é um modo extremamente efetivo de fornecer ajuda para depuração pré-instalada quando um erro é descoberto:

a. Desenvolva um conjunto de diretrizes antidefeitos.

b. Discuta as vantagens do uso dessa técnica.

c. Discuta as desvantagens.

13.6. Como o cronograma do projeto pode afetar o teste de integração?

13.7. O teste de unidade é possível ou mesmo desejável em todas as circunstâncias? Dê exemplos para justificar sua resposta.

13.8. Quem deve realizar o teste de validação — o desenvolvedor de software ou o usuário de software? Justifique sua resposta.

13.9. Desenvolva uma estratégia completa de teste para o sistema *CasaSegura* discutido anteriormente neste livro. Documente-a em uma *Especificação de Teste*.

13.10. Como projeto de classe, desenvolva um *Guia de Depuração* para sua instalação. O guia deve fornecer dicas orientadas a linguagem e sistema que foram aprendidos na "escola dos contratempos!". Comece por um esboço dos tópicos que serão revistos pela classe e seu instrutor. Divulgue o guia para os outros no seu ambiente de trabalho.

Document shared on www.docsity.com

Leituras e Fontes de Informação Adicionais

Virtualmente todo livro sobre teste de software discute estratégias juntamente com métodos para projeto de caso de teste. Craig e Kaskiel (*Systematic Software Testing*, Artech House, 2002), Tamres (*Introducing Software Testing*, Addison-Wesley, 2002), Whittaker (*How to Break Software*, Addison-Wesley, 2002), Jorgensen (*Software Testing: A Craftman's Approach*, CRC Press, 2002), Splaine e seus colegas (*The Web Testing Handbook*, Software Quality Engineering Publishing, 2001), Patton (*Software Testing*, Sams Publishing, 2000), Kaner e seus colegas (*Testing Computer Software*, segunda edição, Wiley, 1999) discutem os princípios, conceitos, estratégias e métodos de teste. Os livros de Black (*Managing the Testing Process*, Microsoft Press, 1999) e Perry (*Surviving the Top Ten Challenges of Software Testing: A People-Oriented Approach*, Dorset House, 1997) também tratam de estratégias de teste de software.

Para aqueles leitores com interesse em métodos de desenvolvimento ágil de software, Crispin e House (*Testing Extreme Programmming*, Addison-Wesley, 2002) e Beck (*Test Driven Development: By Example*, Addison-Wesley, 2002) apresentam estratégias e táticas de teste para Programação Extrema. Kamer e seus colegas (*Lessons Learned in Software Testing*, Wiley, 2001) apresentam uma coleção de mais de 300 "lições" pragmáticas (diretrizes) que todo testador de software deveria aprender. Watkins (*Testing IT: An Off-the Shelf Testing Process*, Cambridge University Press, 2001) estabelece um arcabouço de teste efetivo para todos os tipos de software desenvolvidos e adquiridos.

Lewis (*Software Testing and Continuous Quality Improvement*, CRC Press, 2000) e Koomen e seus colegas (*Test Process Improvement*, Addison-Wesley, 1999) discutem estratégias para melhorar continuamente o processo de teste.

Sykes e McGregor (*Practical Guide to Testing Object-Oriented Software*, Addison-Wesley, 2001), Bashir e Goel (*Testing Object-Oriented Software*, Springer-Verlag, 2000), Binder (*Testing Object-Oriented Systems*, Addison-Wesley, 1999), Kung e seus colegas (*Testing Object-Oriented Software*, IEEE Computer Society Press, 1998), e Marick (*The Craft of Software Testing*, Prentice-Hall, 1997) apresentam estratégias e métodos para teste de sistemas OO.

Diretrizes para depuração estão contidas nos livros de Agans (*Debugging: The Nine Indispensable Rules for Finding Even the Most Elusive Hardware and Software Problems*, AMACON, 2002), Tells e Hsieh (*The Science of Debugging*, The Coreolis Group, 2001), Robbins (*Debugging Applications*, Microsoft Press, 2000), e Dunn (*Software Defect Removal*, McGraw-Hill, 1984). Rosenberg (*How Debuggers Work*, Wiley, 1996) trata de tecnologia para ferramentas de depuração. Younessi (*Object-Oriented Defect Management of Software*, Prentice-Hall, 2002) apresenta técnicas para gerir defeitos que são encontrados em sistemas orientados a objetos. Beizer [BEI84] apresenta uma interessante "taxonomia de erros" que pode levar a métodos efetivos para planejamento de teste. Ball (*Debugging Embedded Microprocessor Systems*, Newnes Publishing, 1998) trata a natureza especial da depuração de software embutido em microprocessadores.

Uma ampla variedade de fontes de informação sobre estratégias de teste de software está disponível na Internet. Uma lista atualizada de referências deste livro que são relevantes para as estratégias de teste de software pode ser encontrada no site: **http://www.mhhe.com/pressman**.

Técnicas de Teste de Software

CAPÍTULO 14

Conceitos-Chave

BVA 329
complexidade ciclomática .. 321
grafos de fluxo 319
padrões 344
particionamento de equivalência 329
testabilidade 316
teste
  com base em cenário .. 334
  baseado em erro 333
  caminho básico 319
  caixa-branca 318
  caixa-preta 327
  de ciclos 326
  de estrutura de controle 324
  ao nível de classe 336
  orientado a objetos 332

O teste expõe uma anomalia interessante para os engenheiros de software, que são por natureza pessoas construtivas. O teste exige que o desenvolvedor descarte noções preconcebidas da "corretividade" do software recém-desenvolvido e depois trabalhe duro para projetar casos de teste que "quebrem" o software. Beizer [BEI90] descreve efetivamente essa situação quando declara:

> Há um mito de que, se fôssemos realmente bons em programação, não haveria erros a revelar. Se pudéssemos realmente nos concentrar, se todos usassem apenas programação estruturada, projeto *por detalhamento progressivo*, tabelas de decisão, se programas fossem escritos em SQUISH, se tivéssemos as balas de prata adequadas, então não haveria erros. E assim vai o mito. Há erros, diz o mito, porque nós somos ruins no que fazemos, e se somos ruins nisso, deveríamos sentir-nos culpados por isso. Assim sendo, testes e projeto de casos de teste é uma admissão de falha, que instila uma boa dose de culpa. E o tédio de testar é apenas punição pelos nossos erros. Punição por quê? Por sermos humanos? Culpa por quê? Por falharmos em alcançar uma perfeição humana? Por não distinguir entre o que outro programador pensa e o que diz? Por falha em ser telepático? Por não resolvermos os problemas de comunicação humana que têm sido passados adiante... durante quarenta séculos?

O teste deveria inculcar culpa? O teste é realmente destrutivo? A resposta a essas questões é Não!

Neste capítulo, discutimos técnicas para o projeto de casos de teste de software. O projeto de casos de testes enfoca um conjunto de técnicas para criação de casos de teste, que satisfazem aos objetivos globais e às estratégias de teste discutidas no Capítulo 13.

Panorama

O que é? Uma vez gerado o código-fonte, o software deve ser testado para descobrir (e corrigir) tantos erros quanto possível antes da entrega ao seu cliente. Sua meta é projetar uma série de casos de teste que têm uma grande probabilidade de encontrar erros; mas como? É aí que as técnicas de teste de software entram em cena. Essas técnicas fornecem diretrizes sistemáticas para projetar testes que (1) exercitam a lógica interna e as interfaces de cada componente de software, e (2) exercitam os domínios de entrada e saída do programa para descobrir erros na função, no comportamento e no desempenho do programa.

Quem faz? Durante os primeiros estágios de teste um engenheiro de software realiza todos os testes. No entanto, à medida que o processo de teste progride, especialistas podem ser envolvidos.

Por que é importante? Revisões e outras atividades de SQA podem e efetivamente descobrem erros, mas elas não são suficientes. Toda vez que o programa é executado, o cliente o testa! Assim, você tem que executar um programa antes que ele chegue ao cliente, com o objetivo específico de encontrar e remover todos os erros. Para encontrar o maior número possível de erros, testes devem ser conduzidos sistematicamente e casos de teste devem ser projetados usando técnicas disciplinadas.

Quais são os passos? Para aplicações convencionais, o software é testado sob duas perspectivas diferentes: (1) a lógica interna do programa é exercitada usando técnicas de projeto de casos de teste "caixa-branca". Requisitos de software são exercitados por meio de técnicas de projeto de casos de teste "caixa-preta". Para aplicações orientadas a objetos, o "teste" começa antes da existência do código-fonte, mas, uma vez gerado o código, uma série de testes é projetada para exercitar operações em uma classe e examinar se existem erros à medida que uma classe colabora com outras. Conforme as classes são integradas para formar um subsistema, testes baseados no uso, com abordagens baseadas em falhas, são aplicados para exercitar plenamente as classes que colaboram. Finalmente, casos de uso ajudam no projeto de testes para descobrir erros

Document shared on www.docsity.com

no âmbito da validação do software. Em cada caso, o objetivo é encontrar o maior número de erros com a menor quantidade de esforço e tempo.

Qual é o produto do trabalho? Um conjunto de casos de teste projetado para exercitar a lógica interna, interfaces, colaborações de componentes e os requisitos externos é projetado e documentado, os resultados esperados são definidos e os resultados reais são registrados.

Como tenho certeza de que fiz corretamente? Quando você começar a testar, modifique o seu ponto de vista. Tente arduamente "quebrar" o software! Projete casos de teste de um modo disciplinado e revise os casos de teste que você criou, quanto ao rigor. Além disso, você pode avaliar a cobertura do teste e monitorar as atividades de detecção de erros.

14.1 Fundamentos do Teste de Software

Os objetivos e princípios fundamentais de teste foram discutidos no Capítulo 5. Reafirma-se que o objetivo de teste é encontrar erros e que um bom teste é aquele que tem alta probabilidade de encontrar um erro. Assim, um engenheiro de software deve projetar e implementar um sistema ou um produto baseado em computador com "testabilidade" em mente. Ao mesmo tempo, os testes devem exibir um conjunto de características que atinge o objetivo de encontrar a maioria dos erros com um mínimo de esforço.

> "Todo programa faz algo certo; isso pode não ser a coisa que desejamos que ele faça."
>
> Autor desconhecido

Testabilidade. James Bach[1] fornece a seguinte definição para testabilidade: "A *testabilidade de software* é simplesmente quão fácil [um programa de computador] pode ser testado". As seguintes características levam a um software testável:

? Quais são as características da testabilidade?

Operabilidade. "Quanto melhor funciona, mais eficientemente pode ser testado." Se um sistema é projetado e implementado com qualidade em mente, poucos defeitos vão bloquear a execução dos testes, permitindo que o teste progrida sem arrancos.

Observabilidade. "O que você vê é o que você testa." Entradas fornecidas como parte do teste produzem saídas distintas. Estados e variáveis do sistema são visíveis ou consultáveis durante a execução. Saída incorreta é facilmente identificada. Erros internos são automaticamente detectados. O código-fonte é acessível.

Controlabilidade. "Quanto melhor você pode controlar o software, mais o teste pode ser automatizado e otimizado." Estados e variáveis do software e do hardware podem ser controlados diretamente pelo engenheiro de teste. Testes podem ser especificados, automatizados e reproduzidos convenientemente.

Decomponibilidade. "Controlando o escopo do teste, podemos isolar problemas mais rapidamente e realizar retestagem mais racionalmente." O sistema de software é construído por meio de módulos independentes, que podem ser testados independentemente.

Simplicidade. "Quanto menos houver a testar, mais rapidamente podemos testá-lo." O programa deve exibir *simplicidade funcional* (por exemplo, o conjunto de características é o mínimo necessário para satisfazer aos requisitos), *simplicidade estrutural* (por exemplo, a arquitetura é modularizada para limitar a propagação de defeitos), *simplicidade do código* (por exemplo, uma norma de codificação é adotada para facilitar a inspeção e a manutenção).

Estabilidade. "Quanto menos modificações, menos interrupções no teste." Modificações no software não são freqüentes, controladas quando ocorrem e não invalidam os testes existentes. O software recupera-se bem das falhas.

1 Os parágrafos que se seguem são usados com permissão de James Bach, © de 1994, e foram adaptados de material que originalmente apareceu em um pôster no grupo de notícias comp.software-eng.

Compreensibilidade. "Quanto mais informações temos, mais racionalmente vamos testar." O projeto arquitetural e as dependências entre componentes internos, externos e compartilhados são bem compreendidos. Documentação técnica é acessível instantaneamente, bem organizada, específica, detalhada e precisa. Modificações ao projeto são comunicadas aos testadores.

Os atributos sugeridos por Bach podem ser usados por um engenheiro de software para desenvolver uma configuração de software (isto é, programas, dados e documentos) que é fácil de testar.

> "Erros são mais comuns, mais disseminados e mais problemáticos no software do que em outras tecnologias."
>
> David Parnas

Características do teste. E sobre os testes propriamente ditos? Kaner, Falk e Nguyen [KAN93] sugerem os seguintes atributos para um "bom" teste:

? O que é um "bom" teste?

1. *Um bom teste tem alta probabilidade de encontrar um erro.* Para alcançar essa meta, o testador deve entender o software e tentar desenvolver uma imagem mental de como o software pode falhar. O ideal é que as classes de falhas sejam investigadas. Por exemplo, uma classe de falhas em potencial em uma interface gráfica com o usuário (*graphical user interface*, GUI) é uma falha em reconhecer a posição correta do mouse. Um conjunto de testes seria projetado para exercitar o mouse em uma tentativa de demonstrar erro no reconhecimento da sua posição.
2. *Um bom teste não é redundante.* O tempo e os recursos para testes são limitados. Não há sentido conduzir um teste que tenha a mesma finalidade de outro. Cada teste deve ter uma finalidade diferente (mesmo que sutilmente diferente).
3. *Um bom teste deve ser "de boa cepa"* [KAN93]. Em um grupo de testes que têm um objetivo semelhante, as limitações de tempo e recursos podem ser abrandadas no sentido da execução de apenas um subconjunto desses testes. Em tais casos, o teste que tenha maior probabilidade de revelar toda uma classe de erros deve ser usado.
4. *Um bom teste não deve ser muito simples nem muito complexo.* Apesar de ser possível combinar algumas vezes uma série de testes em um caso de teste, os efeitos colaterais possíveis, associados com essa abordagem, podem mascarar erros. Em geral, cada teste deve ser executado separadamente.

CasaSegura

Projeto de Testes Singulares

A cena: Cubículo de Vinod.

Os personagens: Vinod e Ed — membros da equipe de engenharia de software do *CasaSegura*.

A conversa:

Vinod: Então esses são os casos de teste que você pretende executar para a operação *validaçãoDeSenha*.

Ed: É, eles devem abranger muito bem todas as possibilidade de tipos de senhas que um usuário possa introduzir.

Vinod: Então vejamos... você percebe que a senha correta vai ser 8080, certo?

Ed: É.

Vinod: E você especifica as senhas 1234 e 6789 para testar contra erros no reconhecimento de senhas inválidas?

Ed: Certo, e eu também testo senhas que estão próximas da senha correta, veja... 8081 e 8180.

Vinod: Essas estão bem, mas eu não vejo muito sentido em você executar as entradas 1234 e 6789. Elas são redundantes... testam a mesma coisa, não é verdade?

Ed: Bem, elas são valores diferentes.

Vinod: Isso é verdade, mas se 1234 não descobre um erro... em outras palavras, *validaçãoDeSenha* percebe que é uma senha inválida, é provável que 6789 não vá nos mostrar nada de novo.

Ed: Entendo o que você quer dizer.

Vinod: Eu não estou tentando ser chato aqui... é que nós temos um tempo limitado para fazer o teste, assim, é uma boa idéia executar testes que tenham uma alta probabilidade de encontrar novos erros.

Ed: Nenhum problema... vou dedicar a isso um pouco mais de raciocínio.

Document shared on www.docsity.com

14.2 Testes Caixa-Preta e Caixa-Branca

Qualquer produto que passe por engenharia (e muitas outras coisas) pode ser testado por uma das duas maneiras: (1) conhecendo-se a função especificada que o produto foi projetado para realizar, podem ser realizados testes que demonstrem que cada função está plenamente operacional e, ao mesmo tempo, procurem erros em cada função; (2) sabendo-se como é o trabalho interno de um produto, podem ser realizados testes para garantir que "todas as engrenagens combinem", isto é, que as operações internas sejam realizadas de acordo com as especificações e que todos os componentes internos foram adequadamente exercitados. A primeira abordagem de teste é chamada de teste caixa-preta e a segunda, de teste caixa-branca.[2]

Teste caixa-preta refere-se a testes que são conduzidos na interface do software. Um teste caixa-preta examina algum aspecto fundamental do sistema, pouco se preocupando com a estrutura lógica interna do software. *Teste caixa-branca* de software é baseado em um exame rigoroso do detalhe procedimental. Caminhos lógicos internos ao software e colaborações entre componentes são testados, definindo-se casos de testes que exercitam conjuntos específicos de condições e/ou ciclos.

> "Há apenas uma regra para o projeto de casos de teste: abranja todas as características, mas não faça muitos casos de teste."
>
> Tsuneo Yamaura

PONTO CHAVE

Testes caixa-branca só podem ser projetados depois que o projeto ao nível de componente (o código-fonte) existir. Os detalhes lógicos do programa devem estar disponíveis.

À primeira vista poderia parecer que um teste caixa-branca bastante rigoroso levaria a programas 100% corretos. Tudo que precisaríamos seria definir todos os caminhos lógicos, desenvolver casos de teste para exercitá-los e avaliar os resultados, isto é, gerar casos de teste para exercitar a lógica do programa de forma exaustiva. Infelizmente, um teste completo apresenta certos problemas logísticos (veja a discussão da moldura). Um teste caixa-branca não deve, no entanto, ser descartado como não-prático. Um número limitado de caminhos lógicos importantes pode ser selecionado e exercitado. Estruturas de dados importantes podem ser submetidas a prova quanto à validade.

INFO

Teste Exaustivo

Considere um programa de 100 linhas em linguagem C. Depois de algumas declarações básicas de dados, o programa contém dois ciclos aninhados, que executam de 1 a 20 vezes cada um, dependendo das condições especificadas na entrada. Dentro do ciclo interior, quatro construções se-então-senão são necessárias. Há aproximadamente 10^{14} caminhos possíveis que podem ser executados nesse programa!

Para colocar esse número em perspectiva, consideramos que um processador de teste mágico ("mágico" porque não existe tal processador) tenha sido desenvolvido para teste completo. O processador pode desenvolver um caso de teste, executá-lo e avaliar os resultados em um milissegundo. Trabalhando 24 horas por dia, 365 dias por ano, o processador trabalharia durante 3.170 anos para testar o programa. Isso iria, inegavelmente, tumultuar a maioria dos cronogramas de desenvolvimento.

Assim, é razoável afirmar que teste exaustivo é impossível para grandes sistemas de software.

14.3 Teste Caixa-Branca

Teste caixa-branca, algumas vezes chamado de *teste caixa de vidro*, é uma filosofia de projeto de casos de teste que usa a estrutura de controle descrita como parte do projeto ao nível de componentes para derivar casos de teste. Usando métodos de teste caixa-branca, o engenheiro de software pode derivar casos de teste que (1) garantam que todos os caminhos independentes de um módulo tenham sido exercitados pelo menos uma vez, (2) exercitem todas as decisões lógicas em seus lados verdadeiro e falso, (3) executem todos os ciclos nos seus limites e dentro de seus intervalos operacionais, e (4) exercitem as estruturas de dados internas para garantir sua validade.

2 Os termos *teste funcional* e *teste estrutural* são algumas vezes usados no lugar de testes caixa-preta e caixa-branca, respectivamente.

> "Os defeitos juntam-se nos cantos e se congregam nas fronteiras."
>
> Boris Beizer

14.4 Teste de Caminho Básico

Teste de caminho básico é uma técnica de teste caixa-branca proposta inicialmente por Tom McCabe [MCC76]. O método de caminho básico permite ao projetista de casos de teste originar uma medida da complexidade lógica de um projeto procedimental e usar essa medida como guia para definir um conjunto básico de caminhos de execução. Casos de testes derivados para exercitar o conjunto básico executam com garantia cada comando do programa pelo menos uma vez durante o teste.

14.4.1 Notação de Grafo de Fluxo

AVISO

Um grafo de fluxo deve ser desenhado somente quando a estrutura lógica de controle de um componente é complexa. O grafo de fluxo permite seguir mais facilmente os caminhos do programa.

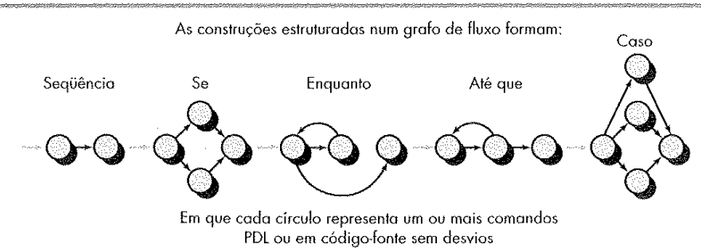
Antes que o método de caminho básico possa ser introduzido, uma notação simples para a representação do fluxo de controle, chamada de *grafo de fluxo* (ou *grafo de programa*) deve ser introduzida.[3] O grafo de fluxo mostra o fluxo de controle lógico usando a notação ilustrada da Figura 14.1. Cada construção estruturada (Capítulo 11) tem um símbolo correspondente de grafo de fluxo.

Para ilustrarmos o uso de um grafo de fluxo, consideramos a representação de projeto procedimental na Figura 14.2a. Lá, um fluxograma é usado para mostrar a estrutura de controle do programa. A Figura 14.2b mapeia o fluxograma em um grafo de fluxo correspondente (considerando que nenhuma condição composta está contida nos losangos de decisão do fluxograma). Com referência à Figura 14.2b, cada círculo, chamado de *nó do grafo de fluxo*, representa um ou mais comandos procedimentais. Uma seqüência de caixas de processamento e um losango de decisão podem ser mapeados em um único nó. As setas do grafo de fluxo, chamadas de *arestas* ou *ligações*, representam o fluxo de controle e são análogas às setas de fluxograma. Uma aresta deve terminar em um nó, mesmo que este não represente nenhum comando procedimental (por exemplo, veja o símbolo de grafo de fluxo para a construção se-então-senão da Figura 14.1). As áreas limitadas por arestas e nós são chamadas de *regiões*. Ao contarmos regiões, incluímos a área fora do grafo como uma região.[4]

Quando condições compostas são encontradas em um projeto procedimental, a geração de um grafo de fluxo torna-se ligeiramente mais complicada. Uma condição composta ocorre quando um ou mais operadores booleanos (OU, E, NÃO-E, NÃO-OU lógicos [em inglês: OR, AND, NAND, NOR])

FIGURA 14.1

Notação de grafo de fluxo

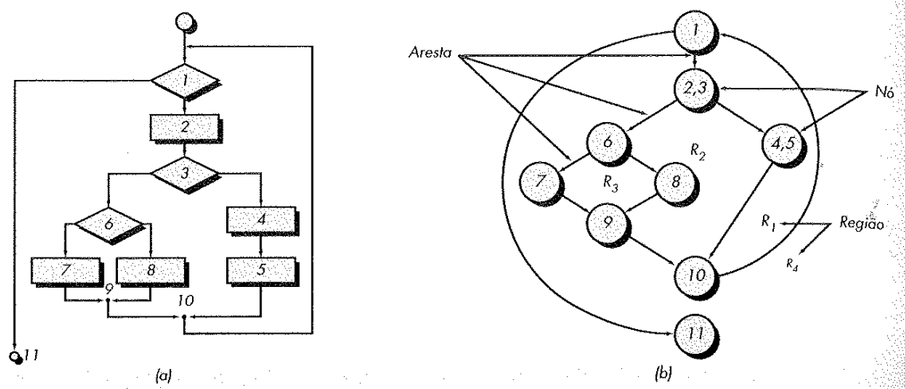


3 Na realidade, o método de caminho básico pode ser conduzido sem uso de grafos de fluxo. No entanto, eles servem como notação útil para entender o fluxo de controle e ilustrar a abordagem.

4 Uma discussão mais detalhada de grafos e seus usos é apresentada na Seção 14.6.1.

Document shared on www.docsity.com

FIGURA 14.2 Fluxograma (a) e grafo de fluxo (b)



está presente em um comando condicional. Com referência à Figura 14.3, o trecho de PDL é traduzido no grafo de fluxo mostrado. Note que um nó separado é criado para cada uma das condições *a* e *b* no comando SE *a* OU *b*. Cada nó que contém uma condição é chamado de *nó predicado* e é caracterizado por duas ou mais arestas saindo dele.

14.4.2 Caminhos Independentes de Programa

Um *caminho independente* é qualquer caminho ao longo do programa que introduz pelo menos um novo conjunto de comandos de processamento ou uma nova condição. Quando enunciado em termos de grafo de fluxo, um caminho independente deve incluir pelo menos uma aresta que não tenha sido atravessada antes de o caminho ser definido. Por exemplo, um conjunto de caminhos independentes para o grafo de fluxo mostrado na Figura 14.2b é

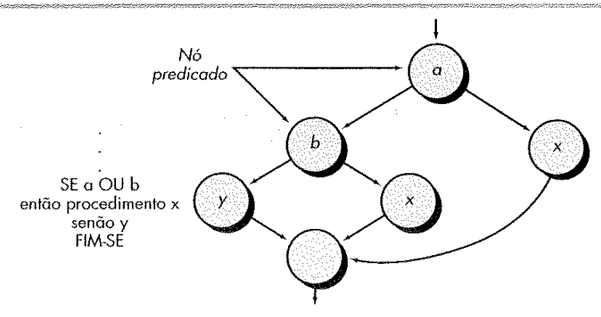
caminho 1: 1-11
caminho 2: 1-2-3-4-5-10-1-11
caminho 3: 1-2-3-6-8-9-10-1-11
caminho 4: 1-2-3-6-7-9-10-1-11

Note que cada novo caminho introduziu uma nova aresta. O caminho

1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

não é considerado caminho independente, porque é apenas uma combinação de caminhos já especificados e não atravessa nenhuma nova aresta.

FIGURA 14.3 Lógica composta



AVISO

A complexidade ciclomática é uma métrica útil para previsão dos módulos que provavelmente sejam propensos a erro. Ela pode ser usada tanto para planejamento de teste quanto para projeto de casos de teste.

? Como a complexidade ciclomática é calculada?

PONTO CHAVE

A complexidade ciclomática fornece o limite superior do número de casos de teste que precisam ser executados, para garantir que cada comando de um componente tenha sido executado pelo menos uma vez.

Os caminhos 1, 2, 3 e 4 constituem um *conjunto-base* para o grafo de fluxo da Figura 14.2b. Isto é, se testes podem ser projetados para forçar a execução desses caminhos (conjunto-base), todo comando do programa terá sido garantidamente executado pelo menos uma vez e cada condição terá sido executada no seu lado verdadeiro e no seu lado falso. Deve-se notar que o conjunto-base não é único. De fato, diversos conjuntos-base diferentes podem ser derivados para um dado projeto procedimental.

Como sabemos quantos caminhos procurar? O cálculo da complexidade ciclomática fornece a resposta. *Complexidade ciclomática* é uma métrica de software que fornece uma medida quantitativa da complexidade lógica de um programa. Quando usada no contexto do método de teste de caminho básico, o valor calculado para a complexidade ciclomática define o número de caminhos independentes no conjunto-base de um programa e nos fornece um limite superior para a quantidade de testes que deve ser conduzida para garantir que todos os comandos sejam executados pelo menos uma vez.

A complexidade ciclomática tem fundamentação na teoria dos grafos e é calculada por uma de três maneiras:

1. O número de regiões corresponde à complexidade ciclomática.
2. A complexidade ciclomática, $V(G)$, para um grafo de fluxo, G, é definida como

   $V(G) = E - N + 2$

   em que E é o número de arestas (*edges*-E) do grafo de fluxo e N é o número de nós (*nodes*-N) do grafo de fluxo.
3. A complexidade ciclomática, $V(G)$, para um grafo de fluxo, G, é também definida como

   $V(G) = P + 1$

   em que P é o número de nós predicados (*predicate nodes*-P) contidos no grafo de fluxo G.

Com referência mais uma vez ao grafo de fluxo da Figura 14.2b, a complexidade ciclomática pode ser calculada usando-se cada um dos algoritmos anteriormente mencionados:

1. O grafo de fluxo tem quatro regiões.
2. $V(G)$ = 11 arestas – 9 nós + 2 = 4.
3. $V(G)$ = 3 nós predicados + 1 = 4.

Mais importante, o valor de $V(G)$ nos dá um limite superior para o número de caminhos independentes que formam o conjunto-base e, por implicação, um limite superior para o número de testes que precisam ser projetados e executados para garantir a cobertura de todos os comandos do programa.

CASASEGURA

Uso da Complexidade Ciclomática

A cena: Cubículo de Shakira.

Os personagens: Vinod e Shakira – membros da equipe de engenharia de software do *CasaSegura* que estão trabalhando no planejamento de teste para a função de segurança.

A conversa:

Shakira: Veja... eu sei que nós deveríamos fazer teste de unidade em todos os componentes da função de segurança, mas eles são muitos e se você considerar o número de operações que têm de ser exercitadas, eu não sei... talvez devêssemos esquecer o teste caixa-branca, integrar tudo e começar a executar testes caixa-preta.

Vinod: Você acha que nós não temos tempo suficiente para fazer testes de componentes, exercitar as operações e depois integrar?

Shakira: O prazo do primeiro incremento está se esgotando mais do que eu gostaria... é, estou preocupada.

Vinod: Por que você pelo menos não executa testes caixa-branca nas operações que são mais prováveis de serem propensas a erros?

Document shared on www.docsity.com

Shakira (desesperada): E como exatamente eu vou saber quais as que são mais propensas a erros?

Vinod: *V* de *G*.

Shakira: O quê?

Vinod: Complexidade ciclomática – *V de G*. Apenas calcule *V*(*G*) para cada uma das operações dentro de cada um dos componentes e veja quais têm o mais alto valor de *V*(*G*). Eles são os mais propensos a erro.

Shakira: E como eu calculo *V* de *G*?

Vinod: É muito fácil. Aqui está um livro que descreve como fazê-lo.

Shakira (folheando as páginas): Certo, não parece difícil, vou fazer uma tentiva. As operações com o *V*(*G*) mais alto serão os candidatos ao teste caixa-branca.

Vinod: Lembre-se apenas de que não há garantia. Um componente com um *V*(*G*) baixo pode ainda ser propenso a erro.

Shakira: Tudo bem. Mas, pelo menos isso vai me ajudar a diminuir o número de componentes que precisam passar por teste caixa-branca.

14.4.3 Derivação de Casos de Teste

O método de teste do caminho básico pode ser aplicado a um projeto procedimental ou ao código-fonte. Nesta seção, apresentamos o teste do caminho básico como uma série de passos. O procedimento *média*, representado em PDL na Figura 14.4, será usado como exemplo para ilustrar cada passo do método de projeto de casos de teste. Note que *média*, apesar de ser um algoritmo extremamente simples, contém condições compostas e ciclos. Os seguintes passos podem ser aplicados para originar o conjunto-base:

1. **Usando o projeto ou código como base, desenhe o grafo de fluxo correspondente.** Um grafo de fluxo é criado usando-se símbolos e regras de construção apresentados na Seção 14.4.1. Referindo-se ao PDL para a média na Figura 14.4, um grafo de fluxo é criado enumerando-se os comandos PDL que vão ser mapeados nos nós correspondentes do grafo de fluxo. O grafo de fluxo correspondente está na Figura 14.5.
2. **Determine a complexidade ciclomática do grafo de fluxo resultante.** A complexidade ciclomática, *V*(*G*) é determinada pela aplicação dos algoritmos descritos na Seção 14.4.2. Deve-se notar que *V*(*G*) pode ser determinado sem desenvolver-se um grafo de fluxo, contando todos os comandos condicionais na PDL (para o procedimento *média*, as condições compostas contam como duas) e somando 1.

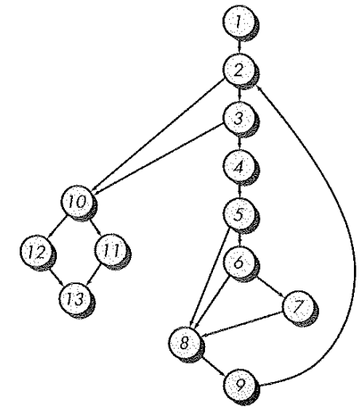
FIGURA 14.4

PDL com nós identificados

```
PRODIMENTO médio;

  * Este procedimento calcula a média de 100
    ou menos números situados entre valores limites;
    calcula também a soma e o total de números válidos.

  INTERFACE RETORNA média, total.entrada, total.válidas;
  INTERFACE ACEITA valor, mínimo, máximo;

  TIPO valor[1:100] É VETOR DE ESCALA;
  TIPO média, total.entrada, total.válidas;
     mínimo, máximo, soma É ESCALA;
  TIPO i É INTEIRO;

1 { i = 1;
    total.entradas = total.válidas = 0;      2
    sum = 0;
    FAÇA ENQUANTO   valor[i] <> -999  E   total.input < 100      3

  4 incremente total.entrada de 1;
      SE  valor[i] > = mínimo   E   valor[i] < = máximo          6
5 {     ENTÃO incremente total.válidas de 1;
  7 {         soma = s soma + valor[i]
        SENÃO pule
8 { FIM-SE
    incremente i de 1;
9 FIM-ENQUANTO
  SE  total.validas > 0          10
  11  ENTÃO média = soma / total.válidas;
12 ---> ELSE average = -999;
13 FIM-SE
FIM média
```


FIGURA 14.5

Grafo de fluxo para o procedimento *média*



Em relação à Figura 14.5,

$V(G)$ = 6 regiões
$V(G)$ = 17 arestas – 13 nós + 2 = 6
$V(G)$ = 5 nós predicados + 1 = 6

> "Errar é humano, encontrar um erro é divino."
>
> Robert Dunn

3. **Determine um conjunto-base de caminhos linearmente independentes.** O valor de *V*(*G*) fornece um número de caminhos linearmente independentes na estrutura de controle do programa. No caso do procedimento *média*, esperamos especificar seis caminhos:

   caminho 1: 1-2-10-11-13
   caminho 2: 1-2-10-12-13
   caminho 3: 1-2-3-10-11-13
   caminho 4: 1-2-3-4-5-8-9-2-...
   caminho 5: 1-2-3-4-5-6-8-9-2-...
   caminho 6: 1-2-3-4-5-6-7-8-9-2-...

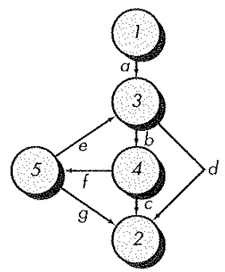
   As reticências (...) após os caminhos 4, 5 e 6 indicam que qualquer caminho pelo resto da estrutura de controle é aceitável. Freqüentemente vale a pena identificar os nós predicados como ajuda para a derivação dos casos de teste. Nesse caso, os nós 2, 3, 5, 6 e 10 são nós predicados.
4. **Prepare casos de teste que vão forçar a execução de cada caminho do conjunto-base.** Os dados devem ser escolhidos de modo que as condições nos nós predicados sejam adequadamente ajustadas à medida que cada caminho é testado. Cada caso de teste é executado e comparado aos resultados esperados. Uma vez completados todos os casos de teste, o testador pode estar certo de que todos os comandos do programa foram executados pelo menos uma vez.

É importante notar que alguns caminhos independentes (por exemplo, caminho 1, no nosso exemplo) não podem ser testados de um modo individual. Isto é, a combinação de dados necessária para percorrer o caminho não pode ser conseguida no fluxo normal do programa. Em tais casos, esses caminhos são testados como parte de outro teste de caminho.

Document shared on www.docsity.com

FIGURA 14.6

Matriz de grafo



Grafo de fluxo

Nó \ Conectado ao nó	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Matriz de grafo

14.4.4 Matrizes de Grafos

O procedimento para originar o grafo de fluxo e mesmo determinar um conjunto de caminhos básicos é passível de mecanização. Para desenvolver uma ferramenta de software que assista ao teste de caminho básico, uma estrutura de dados, chamada de *matriz de grafo*, pode ser bastante útil.

Uma matriz de grafo é uma matriz quadrada, cujo tamanho (isto é, o número de linhas e de colunas) é igual ao número de nós do grafo de fluxo. Cada linha e coluna corresponde a um nó identificado, e as entradas na matriz correspondem às conexões (aresta) entre nós. Um exemplo simples de um grafo de fluxo e sua correspondente matriz de grafo [BEI90] é mostrado na Figura 14.6.

Referindo-se à figura, cada nó do grafo de fluxo é identificado por números, enquanto cada aresta é identificada por letras. A entrada de uma letra é feita na matriz para corresponder à conexão entre dois nós. Por exemplo, o nó 3 está conectado ao nó 4 pela aresta *b*.

? O que é uma matriz de grafo e como a estendemos para uso no teste?

Até este ponto, a matriz de grafo não é nada mais do que uma representação tabular de um grafo de fluxo. No entanto, adicionando um *peso de ligação* a cada entrada da matriz, a matriz de grafo pode tornar-se uma possante ferramenta para avaliar a estrutura de controle do programa durante o teste. O peso da ligação fornece informação adicional sobre o fluxo de controle. Em sua forma mais simples, o peso da ligação é 1 (existe uma conexão) ou 0 (não existe uma conexão). Mas pesos de ligação podem ser atribuídos de acordo com outras propriedades mais interessantes:

- A probabilidade de que uma ligação (aresta) será executada.
- O tempo de processamento gasto durante o percurso de uma ligação.
- A memória necessária durante o percurso de uma ligação.
- Os recursos necessários durante o percurso de uma ligação.

Beizer [BEI90] fornece um rigoroso tratamento de outros algoritmos matemáticos que podem ser aplicados a matrizes de grafo. Usando essas técnicas, a análise necessária para projetar casos de teste pode ser parcial ou totalmente automatizada.

> "Dar mais atenção à execução dos testes do que ao projeto deles é um erro clássico."
>
> Brian Marick

14.5 TESTE DE ESTRUTURA DE CONTROLE

A técnica de teste de caminho básico descrita na Seção 14.4 é uma de várias técnicas de teste da estrutura de controle. Apesar de o teste de caminho básico ser simples e altamente eficaz, não é suficiente por si só. Nesta seção, outras variações do teste da estrutura de controle são discutidas brevemente. Elas ampliam a cobertura de teste e melhoram a qualidade do teste caixa-branca.

PONTO CHAVE

Erros são muito mais comuns na vizinhança de condições lógicas do que próximos aos comandos de processamento seqüencial.

14.5.1 Teste de Condição

Teste de condição[TAI89] é um método de projeto de caso de teste que exercita as condições lógicas contidas em um módulo de programa. Uma *condição simples* é uma variável booleana ou uma expressão relacional, possivelmente precedida por um operador NÃO (¬). Uma expressão relacional toma a forma

E_1 <operador-relacional> E_2

em que E_1 e E_2 são expressões aritméticas e <operador-relacional> é um dos seguintes: <, ≤, =, ≠ (desigualdade), > ou ≥. Uma *condição composta* é formada por duas ou mais condições simples, operadores booleanos e parênteses. Consideramos que operadores booleanos permitidos em uma condição composta incluem OU (|), E (&) e NÃO (¬). Uma condição sem expressões relacionais é referida como expressão booleana. Assim, os tipos de elementos possíveis em uma condição incluem um operador booleano, uma variável booleana, um par de parênteses (envolvendo uma condição simples ou composta), um operador relacional ou uma expressão aritmética.

Se uma condição está incorreta, então pelo menos um componente da condição está incorreto. Assim, os tipos de erros em uma condição incluem erros de operador booleano (operadores booleanos incorretos/faltando/extra), erros de variável booleana, erros de parênteses booleano, erros de operador relacional e erro de expressão aritmética. O método de teste de condição focaliza o teste de cada condição do programa para garantir que não contém erros.

14.5.2 Teste de Fluxo de Dados

O método *de teste de fluxo de dados* seleciona caminhos de teste de um programa de acordo com a localização das definições e dos usos das variáveis no programa. Para ilustrar a abordagem de teste de fluxo de dados, considere que a cada comando de um programa é atribuído um número de comando único e que cada função não modifica seus parâmetros ou variáveis globais. Para um comando com *S* como seu número de comando,

DEF(*S*) = {*X* | comando *S* contém uma definição de *X*}
USO(*S*) = {*X* | comando *S* contém um uso de *X*}

AVISO

É irreal considerar que o teste de fluxo de dados será usado extensamente quando se testa um sistema grande. No entanto, ele pode ser direcionado a áreas do software duvidosas.

Se o comando *S* é um comando *se* ou de *ciclo*, seu conjunto DEF é vazio e seu conjunto USO é baseado na condição do comando *S*. A definição da variável *X* no comando *S* é considerada *viva* no comando *S´*, se existir um caminho do comando *S* para o comando *S´* que não contenha nenhuma outra definição de *X*.

Uma *cadeia definição-uso* (DU) da variável *X* é da forma [*X*, *S*, *S´*] em que *S* e *S´* são números de comandos, *X* pertence a DEF (*S*) e USO (*S´*), e a definição de *X* no comando *S* está viva no comando *S´*.

Uma estratégia simples de teste de fluxo de dados é exigir que cada cadeia DU seja coberta pelo menos uma vez. Nos referimos a essa estratégia como a *estratégia de teste DU*. Foi mostrado que o teste DU não garante a cobertura de todos os ramos de um programa. No entanto, não é garantido que um ramo seja coberto por teste DU apenas em situações raras tais como construções se-então-senão nas quais a *parte então* não tem definição de nenhuma variável e a *parte senão* não existe. Nessa situação, o ramo *senão* de comando *se* não é necessariamente coberto pelo teste DU. Um certo número de estratégias de teste de fluxo de dados foi estudado e comparado (por exemplo, [FRA88], [NTA88], [FRA93]). O leitor interessado é estimulado a considerar essas outras referências.

> "Bons testadores são mestres em notar 'algo esquisito' e agir com base nisso."
>
> Brian Marick

Document shared on www.docsity.com

14.5.3 Teste de Ciclo

Os ciclos (*loops*) são a pedra fundamental da grande maioria de todos algoritmos implementados em software. No entanto, freqüentemente nós lhes damos pouca atenção ao conduzirmos testes de software.

Teste de ciclo é uma técnica de teste caixa-branca que focaliza exclusivamente a validade de construções de ciclo. Quatro diferentes classes de ciclos [BEI90] podem ser definidas: ciclos simples, concatenados, aninhados e desestruturados (Figura 14.7).

Ciclos simples. O seguinte conjunto de testes pode ser aplicado a ciclos simples em que n é o número máximo de passagens permitidas pelo ciclo.

1. Pule o ciclo completamente.
2. Apenas uma passagem pelo ciclo.
3. Duas passagens pelo ciclo.
4. m passagens pelo ciclo em que $m < n$.
5. $n - 1$, n, $n + 1$ passagens pelo ciclo.

Ciclos aninhados. Se tivéssemos que estender a abordagem de teste de ciclos simples para ciclos aninhados, o número de testes possíveis cresceria geometricamente, à medida que o nível de aninhamento crescesse. Isso resultaria em um número de testes impraticável. Beizer [BEI90] sugere uma abordagem que vai ajudar a reduzir o número de testes:

1. Comece no ciclo mais interno. Ajuste todos os outros ciclos para os valores mínimos.
2. Conduza testes de ciclo simples para o ciclo mais interno enquanto mantém os ciclos externos nos seus valores mínimos do parâmetro de iteração (por exemplo, contador de ciclo). Adicione outros testes para valores fora do intervalo ou excluídos.
3. Trabalhe em direção ao exterior, conduzindo testes para o ciclo seguinte, mas mantendo todos os outros ciclos externos nos valores mínimos e os outros ciclos aninhados em valores "típicos".
4. Continue até que todos os ciclos tenham sido testados.

AVISO

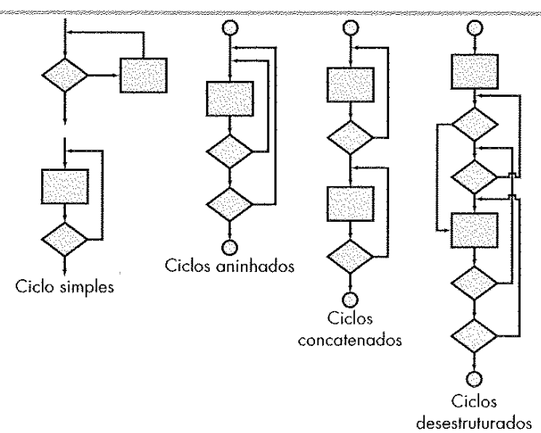
Você não pode testar efetivamente ciclos desestruturados. Reprojete-os.

Ciclos concatenados. Esses ciclos podem ser testados usando a abordagem definida para ciclos simples, se cada um dos ciclos é independente do outro. No entanto, se dois ciclos são concatenados e o contador de ciclo, para o ciclo 1, é usado como valor inicial para o ciclo 2, então os ciclos não são independentes. Quando os ciclos não são independentes, a abordagem aplicada a ciclos aninhados é recomendada.

Ciclos desestruturados. Sempre que possível essa classe de ciclos deve ser reprojetada para refletir o uso de construções de programação estruturada (Capítulo 11).

FIGURA 14.7

Classes de ciclos



14.6 Teste Caixa-Preta

Um *teste caixa-preta*, também chamado de *teste comportamental*, focaliza os requisitos funcionais do software. Isto é, o teste caixa-preta permite ao engenheiro de software derivar conjuntos de condições de entrada que vão exercitar plenamente todos os requisitos funcionais de um programa. Ele não é uma alternativa às técnicas caixa-branca, ao contrário, é uma abordagem complementar, que provavelmente descobrirá uma classe diferente de erros do que os métodos caixa-branca.

O teste caixa-preta tenta encontrar erros das seguintes categorias: (1) funções incorretas ou omitidas, (2) erros de interface, (3) erros de estrutura de dados ou de acesso à base de dados externa, (4) erros de comportamento ou desempenho e (5) erros de iniciação e término.

Diferentemente do teste caixa-branca, que é realizado no início do processo de teste, o teste caixa-preta tende a ser aplicado durante os últimos estágios do teste (veja o Capítulo 13). Como o teste caixa-preta despreza, de propósito, a estrutura de controle, a atenção é focalizada no domínio da informação. Os testes são projetados para responder às seguintes questões:

? *À que questões os testes caixa-preta respondem?*

- Como a validade funcional é testada?
- Como o comportamento e o desempenho do sistema são testados?
- Que classes de entrada vão constituir bons casos de teste?
- O sistema é particularmente sensível a certos valores de entrada?
- Como são isolados os limites de uma classe de dados?
- Que taxas e volumes de dados o sistema pode tolerar?
- Que efeito as combinações específicas de dados vão ter na operação do sistema?

Aplicando técnicas caixa-preta, derivamos um conjunto de casos de teste que satisfaz aos seguintes critérios [MYE79]: (1) casos de teste que reduzem, de um valor que é maior do que 1, o número adicional de casos de teste que precisam ser projetados para atingir um teste razoável e (2) casos de teste que nos dizem algo sobre a presença ou ausência de classes de erros, em vez de um erro associado somente com o teste específico em mãos.

14.6.1 Métodos de Teste Baseados em Grafo

O primeiro passo no teste caixa-preta é entender os objetos[5] que estão modelados no software e as relações que conectam esses objetos. Uma vez que isso tenha sido conseguido, o passo seguinte é definir uma série de testes que verifica se "todos os objetos têm a relação esperada uns com os outros [BEI95]". Dito de outro modo, o teste de software começa criando um grafo dos objetos importantes e de suas relações e depois estabelecendo uma série de testes que vão cobrir o grafo de modo que cada objeto e relação sejam exercitados e que os erros sejam descobertos.

PONTO CHAVE

Um grafo representa a relação entre objetos de dados e objetos de programa, permitindo-nos derivar casos de teste que procuram erros associados a essas relações.

Para executar esses passos, o engenheiro de software começa criando um *grafo* — uma coleção de nós que representam objetos; *ligações* que representam as relações entre objetos; *pesos de nó* que descrevem as propriedades de um nó (por exemplo, um valor de dados específico ou comportamento de um estado); e *pesos de ligação* que descrevem algumas características de uma ligação.

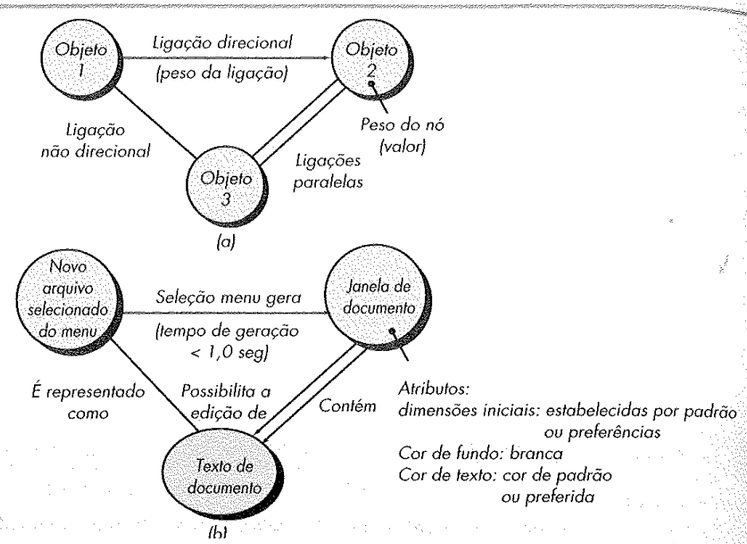
A representação simbólica de um grafo é mostrada na Figura 14.8a. Os nós são representados com círculos conectados por ligações que assumem algumas formas diferentes. Uma *ligação direcionada* (representada por uma seta) indica que a relação se move em apenas uma direção. Uma *ligação bidirecional*, também chamada de *ligação simétrica*, implica que a relação é aplicada em ambas as direções. *Ligações paralelas* são usadas quando diversas relações diferentes são estabelecidas entre nós de grafo.

5 Aqui, consideramos o termo "objeto" no contexto mais amplo possível. Ele abrange os objetos de dados, componentes tradicionais (módulos) e elementos orientados a objetos de software de computador.

Document shared on www.docsity.com

FIGURA 14.8

(a) Notação de grafo
(b) Exemplo simples



Como exemplo simples, considere uma parte de um grafo para uma aplicação de processamento de texto (Figura 14.8b) em que

Objeto 1 = **arquivoNovo** (selecionado no menu)
Objeto 2 = **janelaDeDocumento**
Objeto 3 = **textoDeDocumento**

Com referência à figura, uma seleção de menu de **arquivoNovo** gera uma **janelaDeDocumento**. O peso do nó de = **janelaDeDocumento** fornece uma lista de atributos da janela, que devem ser esperados quando ela é gerada. O peso da ligação indica que a janela deve ser gerada em menos que 1,0 segundo. Uma ligação não direcional estabelece uma relação simétrica entre **arquivoNovo** selecionado no menu e **textoDeDocumento**, e ligações paralelas indicam relações entre **janelaDeDocumento** e **textoDeDocumento**. Na verdade, um grafo muito mais detalhado deveria ser gerado como precursor do projeto de casos de teste. O engenheiro de software origina depois casos de teste percorrendo o grafo e cobrindo cada uma das relações mostradas. Esses casos de teste são projetados em uma tentativa de encontrar erros em quaisquer das relações.

Beizer [BEI95] descreve alguns métodos de teste de comportamento que podem fazer uso de grafos:

Modelagem de fluxo de transação. Os nós representam passos em alguma transação (por exemplo, os passos necessários para fazer uma reserva em uma linha aérea usando um serviço on-line), e as ligações representam as conexões lógicas entre os passos. O diagrama de fluxo de dados (Capítulo 8) pode ser usado para ajudar a criar grafos desse tipo.

Modelagem de estado finito. Os nós representam diferentes estados do software observáveis pelo usuário (por exemplo, cada uma das "telas" que aparece, enquanto um funcionário, que dá entrada em pedidos, recebe um pedido por telefone) e as ligações representam as transições que ocorrem para ir de um estado para outro estado. O diagrama de transição de estados (Capítulo 8) pode ser usado para ajudar na criação de grafos desse tipo.

Modelagem do fluxo de dados. Os nós são objetos de dados e as ligações são as transformações que ocorrem para traduzir um objeto de dados em outro. Por exemplo, o nó **imposto.retido (*FICA tax withheld*, FTW)** é calculado com base no **salário bruto (*gross wages*, GW)** usando a relação **FTW = 0,62 × GW**.

Modelagem de tempo. Os nós são objetos de programa e as ligações, as conexões seqüenciais entre esses objetos. Os pesos das ligações são usados para especificar os tempos de execução necessários enquanto o programa é executado.

Uma discussão detalhada de cada um desses métodos de teste baseados em grafos foge do escopo deste livro. O leitor interessado deve ver [BEI95] para uma análise abrangente.

14.6.2 Particionamento de Equivalência

O *particionamento de equivalência* é um método de teste caixa-preta que divide o domínio de entrada de um programa em classes de dados, das quais os casos de teste podem ser derivados. Um caso de teste ideal descobre sozinho uma classe de erros (por exemplo, processamento incorreto de todos os dados de caracteres), que poderia de outra forma exigir que muitos casos fossem executados antes que um erro geral fosse observado. O particionamento de equivalência busca definir um caso de teste que descobre classes de erros, reduzindo assim o número total de casos de testes que precisam ser desenvolvidos.

AVISO

Classes de entrada são conhecidas relativamente no início do processo de software. Por essa razão, comece a pensar no particionamento de equivalência logo que o projeto é criado.

Projeto de casos de teste para particionamento de equivalência é baseado em uma avaliação das classes de equivalência para uma condição de entrada. Usando-se conceitos introduzidos na seção anterior, se o conjunto de objetos puder ser ligado por relações simétricas, transitivas e reflexivas, uma classe de equivalência estará presente [BEI95]. Uma *classe de equivalência* representa um conjunto de estados válidos ou inválidos para condições de entrada. Tipicamente, uma condição de entrada é um valor numérico específico, um intervalo de valores, um conjunto de valores relacionados ou uma condição booleana. Classes de equivalência podem ser definidas de acordo com as seguintes diretrizes:

? Como defino classes de equivalência para teste?

1. Se uma condição de entrada especifica um intervalo, uma classe de equivalência válida e duas inválidas são definidas.
2. Se uma condição de entrada exige um valor específico, uma classe de equivalência válida e duas inválidas são definidas.
3. Se uma condição de entrada especifica o membro de um conjunto, uma classe de equivalência válida e uma inválida são definidas.
4. Se uma condição de entrada é booleana, uma classe de equivalência válida e uma inválida são definidas.

Pela aplicação dessas diretrizes para derivação das classes de equivalência, casos de teste para cada objeto de dados do domínio de entrada podem ser desenvolvidos e executados. Casos de testes são selecionados de modo que o maior número de atributos de uma classe de equivalência seja exercitado ao mesmo tempo.

14.6.3 Análise de Valor-limite

PONTO CHAVE

BVA estende o particionamento de equivalência focalizando os dados nas "fronteiras" de uma classe de equivalência.

Um grande número de erros ocorre nas fronteiras do domínio de entrada em vez de no "centro". É por essa razão que a *análise de valor-limite* (*boundary value analysis*, BVA) foi desenvolvida como técnica de teste. BVA leva à seleção de casos de teste que exercitam os valores limítrofes.

A análise de valor-limite é uma técnica de projeto de casos de teste que completa o particionamento de equivalência. Em vez de selecionar qualquer elemento de uma classe de equivalência, a BVA leva à seleção de casos de teste nas "bordas" da classe. Em vez de focalizar somente as condições de entrada, ela deriva casos de teste também para o domínio de saída [MYE79].

As diretrizes para a BVA são semelhantes em muitos aspectos às fornecidas para o particionamento de equivalência:

? Como crio casos de testes BVA?

1. Se uma condição de entrada especifica um intervalo limitado pelos valores *a* e *b*, casos de teste devem ser projetados com os valores *a* e *b*, e imediatamente acima e imediatamente abaixo de *a* e *b*.
2. Se uma condição de entrada especifica vários valores, casos de teste devem ser desenvolvidos para exercitar os números mínimo e máximo. Valores imediatamente acima e imediatamente abaixo do mínimo e do máximo também são testados.

Document shared on www.docsity.com

3. Aplique as diretrizes 1 e 2 às condições de saída. Por exemplo, considere que uma tabela de temperatura *versus* pressão é esperada como saída de um programa de análise de engenharia. Casos de teste devem ser projetados para criar um relatório de saída que produza o número máximo (e mínimo) admissível de entradas na tabela.
4. Se as estruturas de dados internas do programa têm limites prescritos (por exemplo, um vetor tem um limite definido de 100 entradas), certifique-se de projetar um caso de teste para exercitar a estrutura de dados no seu limite.

A maioria dos engenheiros de software realiza a BVA intuitivamente em um certo grau. Aplicando essas diretrizes, o teste de limite será mais completo, tendo assim uma maior probabilidade de detecção de erro.

> "O foguete Ariane 5 explodiu na subida devido tão-somente a um defeito de software (um erro) envolvendo a conversão de um valor de ponto flutuante com 64 bits em um inteiro de 16 bits. O foguete e seus quatro satélites estavam sem seguro e valiam 500 milhões de dólares. Um teste de sistema abrangente teria encontrado o erro, mas foi vetado por razões orçamentárias."
>
> Uma notícia

14.6.4 Teste de Matriz Ortogonal

Há muitas aplicações nas quais o domínio de entrada é relativamente limitado, isto é, o número de parâmetros de entrada é pequeno e os valores que cada um dos parâmetros pode assumir são claramente limitados. Quando esses números são muito pequenos (por exemplo, três parâmetros de entrada assumindo três valores discretos cada um), é possível considerar cada permutação de entrada e testar exaustivamente o processamento do domínio de entrada. No entanto, à medida que o número de valores de entrada cresce e o número de valores discretos para cada item de dados aumenta, o teste exaustivo torna-se impraticável ou impossível.

Teste de matriz ortogonal pode ser aplicado a problemas nos quais o domínio de entrada é relativamente pequeno, mas grande demais para acomodar o teste exaustivo. O método de teste de matriz ortogonal é particularmente útil para encontrar erros associados com *falhas de regiões* — uma categoria de erros associada com lógica defeituosa em um componente de software.

PONTO CHAVE

O teste de matriz ortogonal permite que você projete casos de teste que fornecem máxima cobertura com um número razoável de casos de teste.

Para ilustrar a diferença entre teste de matriz ortogonal e abordagens mais convencionais "um item de entrada de cada vez", considere um sistema que tem três itens de entrada, *X, Y* e *Z*. Cada um desses itens de entrada tem três valores discretos associados a ele. Há então $3^3 = 27$ possíveis casos de teste. Phadke [PHA97] sugere uma visão geométrica dos possíveis casos de teste associados a *X, Y* e *Z* ilustrada na Figura 14.9. Com referência à figura, um item de entrada de cada vez pode ser variado em seqüência ao longo de cada eixo de entrada. Isso resulta em uma cobertura relativamente limitada do domínio de entrada (representada pelo cubo à esquerda na figura).

Quando o teste de matriz ortogonal ocorre, é criada uma *matriz ortogonal L9* de casos de teste. A matriz ortogonal L9 tem uma "propriedade de balanceamento" [PHA97]. Isto é, casos de teste (representados pelas bolas pretas da figura) estão "espalhados uniformemente pelo domínio de teste", como é ilustrado no cubo à direita na Figura 14.9. A cobertura de teste ao longo do domínio de entrada é mais completa.

Para ilustrar o uso da matriz ortogonal L9, considere a função *envie* de uma aplicação de fax. Quatro parâmetros, P1, P2, P3 e P4, são passados para a função *envie*. Cada um assume três valores específicos.

Por exemplo, P1 assume os valores:

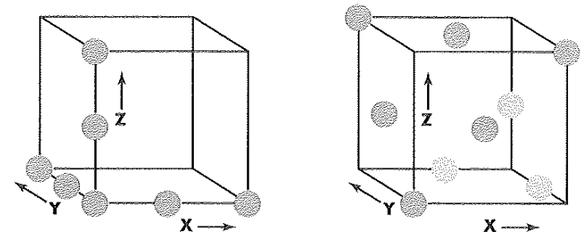
P1 = 1, envie já
P1 = 2, envie uma hora mais tarde
P1 = 3, envie depois da meia-noite

P2, P3 e P4 também assumiriam valores de 1, 2 e 3, significando outras funções *envie*.

Se uma estratégia de teste "um item de entrada de cada vez" fosse escolhida, a seguinte seqüência de testes (P1, P2, P3, P4) seria especificada: (1, 1, 1, 1), (2, 1, 1, 1), (3, 1, 1, 1), (1, 2, 1, 1), (1, 3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1), (1, 1, 1, 2) e (1, 1, 1, 3). Phadke [PHA97] avalia esses casos de teste do seguinte modo:

FIGURA 14.9

Uma visão geométrica dos casos de teste (PHA97)



Um item de entrada de cada vez **Matriz ortogonal L9**

Tais casos de teste são úteis apenas quando há certeza de que esses parâmetros de testes não interagem. Eles podem detectar falhas de lógica em que um único valor de parâmetro provoca o mau funcionamento do software. Essas falhas são chamadas, *falhas de modo singular*. Esse método não pode detectar falhas de lógica que causam mau funcionamento quando dois ou mais parâmetros assumem simultaneamente certos valores; isto é, não pode detectar nenhuma interação. Assim sua habilidade de detectar falhas é limitada.

Como o número de parâmetros de entrada e de valores discretos é relativamente pequeno, o teste exaustivo é possível. O número de testes necessário é $3^4 = 81$, grande, mas exeqüível. Todas as falhas associadas com permutação dos itens de dados seriam encontradas, mas o esforço necessário é relativamente alto.

A abordagem de teste de matriz ortogonal nos possibilita obter boa cobertura de teste com muito menos casos de teste que a estratégia exaustiva. Uma matriz ortogonal L9 para a função *envie* de um fax é ilustrada na Figura 14.10.

Phadke [PHA97] avalia o resultado dos testes usando a matriz ortogonal L9 do seguinte modo:

Detecta e isola todas as falhas de modo singular. Uma falha de modo singular é um problema consistente com qualquer nível de qualquer parâmetro singular. Por exemplo, se todos os casos de teste do fator P1 = 1 causam um erro de condição, trata-se de uma falha de modo singular. Nesse exemplo, os testes 1, 2 e 3 [Figura 14.10] vão revelar erros. Pela análise da informação sobre quais testes revelam erros, pode-se identificar quais valores de parâmetro causam a falha. Nesse exemplo, notando que os testes 1, 2 e 3 causam erro, pode-se isolar [o processamento lógico associado com "envie já" (P1 = 1)] como causa do erro. Esse isolamento da falha é importante para corrigir a falha.

Detecta todas as falhas de modo duplo. Se existe um problema consistente quando níveis específicos de dois parâmetros ocorrem simultaneamente, ele é chamado de *falha de modo duplo*.

FIGURA 14.10

Uma matriz ortogonal L9

Caso de teste	Parâmetros de teste			
	P_1	P_2	P_3	P_4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Document shared on www.docsity.com

De fato, uma falha de modo duplo é uma indicação de incompatibilidade do par ou de interações danosas entre dois parâmetros de teste.

Falhas de multimodo. Matrizes ortogonais [do tipo mostrado] podem garantir a detecção apenas de falhas de modo singular e duplo. No entanto, muitas falhas de multimodo são também detectadas por esses testes.

Uma discussão detalhada do teste de matriz ortogonal pode ser encontrada em [PHA89].

FERRAMENTAS DE SOFTWARE

Projeto de Caso de Teste

Objetivo: Apoiar a equipe de software no desenvolvimento de um conjunto completo de casos de teste tanto para teste caixa-preta quanto para caixa-branca.

Mecânica: Essas ferramentas estão em duas amplas categorias: teste estático e teste dinâmico. Três diferentes tipos de ferramentas de teste estático são usados na indústria: ferramentas de teste baseadas em código, linguagens especializadas de teste e ferramentas de teste baseadas em requisitos. Ferramentas de teste baseadas em código aceitam código-fonte como entrada e executam um certo número de análises que resultam na geração de casos de teste. Linguagens especializadas de teste (por exemplo, ATLAS) possibilitam ao engenheiro de software escrever especificações de teste detalhadas que descrevem cada caso de teste e a logística para sua execução. Ferramentas de teste baseadas nos requisitos isolam os requisitos específicos do usuário e sugerem casos de teste (ou classes de teste) que exercitam os requisitos. Ferramentas dinâmicas de teste interagem com um programa de execução, checam a cobertura do caminho, testam as assertivas sobre valores de variáveis específicas e ainda instrumentam o fluxo de execução do programa.

Ferramentas Representativas[6]

McCabe Test, desenvolvida por McCabe & Associates (*www.mccabe.com*), implementa uma variedade de técnicas de teste de caminho derivada com base em uma avaliação de complexidade ciclomática e outras métricas de software.

Panorama, desenvolvida por International Software Automation, Inc. (*www.softwareautomation.com*), engloba um conjunto completo de ferramentas para desenvolvimento de software orientado a objetos incluindo ferramentas que apóiam o projeto de caso de teste e planejamento de teste.

TestWorks, desenvolvida por Software Research, Inc. (*www.soft.com/Products*), é um conjunto completo de ferramentas automatizadas de teste que apóia o projeto de casos de teste para software desenvolvido em C/C++ e Java e fornece suporte para teste de regressão.

T-Vec Test Generation System, desenvolvida por T-VEC Technologies (*www.t-vec.com*), é um conjunto de ferramentas que suporta teste de unidade, integração e validação apoiando o projeto de casos de teste por meio de informação contida em uma especificação de requisitos OO.

14.7 MÉTODOS DE TESTE ORIENTADOS A OBJETOS

A arquitetura de software orientado a objetos resulta em uma série de subsistemas em camadas que encapsulam classes de colaboração. Cada um desses elementos do sistema (subsistemas e classes) executa funções que ajudam a satisfazer aos requisitos do sistema. É necessário testar um sistema OO em uma variedade de níveis diferentes para descobrir erros que podem ocorrer à medida que classes colaboram umas com as outras e subsistemas se comunicam entre as camadas arquiteturais.

Teste orientado a objetos é estrategicamente similar ao teste de sistemas convencionais, mas é taticamente diferente. Como modelos de análise e projeto OO são similares na estrutura e no conteúdo para o programa OO resultante, o "teste" pode começar com a revisão desses modelos. Uma vez gerado o código, o teste OO real começa "no varejo" com uma série de testes projetados para exercitar operações de classes e examinar se existem erros quando uma classe colabora com

6 As ferramentas mencionadas aqui não representam uma recomendação, mas, em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

outras classes. À medida que as classes são integradas para formar um subsistema, o teste baseado no uso, em conjunto com abordagens tolerantes a falhas, é aplicado para exercitar completamente as classes que colaboram entre si. Finalmente, casos de uso são usados para descobrir erros no nível de validação do software.

Projeto de caso de teste convencional é dirigido por uma visão de software de entrada-processo-saída ou detalhes algorítmicos de módulos individuais. Teste orientado a objetos enfoca o projeto de seqüências apropriadas de operações para exercitar os estados de uma classe.

Veja na Web

Uma excelente coleção de artigos e recursos sobre o teste OO pode ser encontrada em www.rbsc.com.

14.7.1 Implicações no Projeto de Casos de Teste dos Conceitos OO

À medida que uma classe evolui ao longo dos modelos de análise e projeto, ela se torna alvo do projeto de casos de teste. Como os atributos e operações são encapsulados, o teste de operações fora da classe é geralmente improdutivo. Apesar de o encapsulamento ser um conceito de projeto essencial para OO, ele pode criar um pequeno obstáculo quando o teste é conduzido. De acordo com Binder [BIN94], "o teste exige registro do estado concreto e abstrato de um objeto". No entanto, o encapsulamento torna essa informação um tanto difícil de obter. A menos que operações embutidas sejam providenciadas para relatar os valores dos atributos da classe, um instantâneo do estado de um objeto pode ser difícil de obter.

A herança também leva a desafios adicionais para o projetista de casos de teste. Já mencionamos que cada novo contexto de uso exige retestagem, mesmo que o reuso tenha sido concebido. Além disso, herança múltipla[7] complica ainda mais o teste, aumentando o número de contextos para os quais o teste é necessário [BIN94]. Se subclasses instanciadas de uma superclasse são usadas no mesmo domínio do problema, é provável que o conjunto de casos de teste derivado para a superclasse possa ser usado para testar a subclasse. No entanto, se a subclasse é usada em um contexto inteiramente diferente, os casos de teste da superclasse terão pouca aplicabilidade e um novo conjunto de testes precisa ser projetado.

14.7.2 Aplicabilidade dos Métodos Convencionais de Projeto de Casos de Teste

Os métodos de teste caixa-branca, descritos nas seções anteriores, podem ser aplicados a operações definidas para uma classe. Caminho-base, teste de ciclos ou técnicas de fluxo de dados podem ajudar a garantir que cada declaração em uma operação tenha sido testada. No entanto, a estrutura concisa de muitas operações de classe faz alguns argumentarem que o esforço aplicado ao teste caixa-branca poderia ser mais bem redirecionado para testes no nível de classe.

Os métodos de teste caixa-preta são tão adequados para sistemas OO quanto são para sistemas desenvolvidos usando os métodos convencionais da engenharia de software. Como mencionamos antes neste capítulo, casos de uso podem fornecer entrada útil no projeto de testes caixa-preta e baseados em estado [AMB95].

PONTO CHAVE

A estratégia para teste baseado em erros é admitir, por hipótese, um conjunto de erros possíveis e depois derivar testes para provar cada hipótese.

14.7.3 Teste Baseado em Erro[8]

O objetivo do *teste baseado em erros* em um sistema OO é projetar testes que tenham uma grande probabilidade de descobrir erros plausíveis. Como o produto ou sistema deve satisfazer aos requisitos do cliente, o planejamento preliminar necessário para realizar o teste baseado em erros começa com o modelo de análise. O testador procura erros plausíveis (i. e., aspectos da implementação do sistema que podem resultar em defeitos). Para determinar se esses erros existem, casos de teste são projetados para exercitar o projeto ou o código.

Sem dúvida, a efetividade dessas técnicas depende de como os testadores consideram um erro plausível. Se erros reais em um sistema OO são considerados não plausíveis, então essa abordagem não é realmente melhor do que qualquer técnica de teste aleatório. No entanto, se os modelos de

7 Um conceito OO que deve ser usado com extremo cuidado.

8 As Seções 14.7.3 a 14.7.6 foram adaptadas de um artigo de Brian Marick inserido no grupo de notícias da Internet **comp.testing**. Essa adaptação está incluída com permissão do autor. Para mais informação sobre esses tópicos, veja [MAR94]. Deve-se notar que as técnicas discutidas nas Seções 14.7.3 a 14.7.6 também são aplicáveis para software convencional.

Document shared on www.docsity.com

análise e projeto puderem fornecer conhecimento aprofundado sobre o que é provável dar errado, então, o teste baseado em erros pode encontrar um número significativo de erros com dispêndio de esforço relativamente baixo.

? Que tipos de erros são encontrados na chamada de operações e nas conexões de mensagens?

O teste de integração procura erros plausíveis na chamada de operações ou nas conexões de mensagens. Três tipos de erros são encontrados nesse contexto: resultado inesperado, uso da operação/mensagem errada e invocação incorreta. Para determinar os erros plausíveis, quando as funções (operações) são invocadas, o comportamento da operação deve ser examinado.

O teste de integração se aplica tanto a atributos quanto a operações. Os "comportamentos" de um objeto são definidos pelos valores atribuídos a seus atributos. O teste deve exercitar os atributos para determinar se ocorrem valores adequados para os tipos distintos de comportamento do objeto.

É importante notar que o teste de integração tenta encontrar erros no objeto cliente, não no servidor. Dito em termos convencionais, o foco do teste de integração está em determinar se existem erros no código que chama, não no código chamado. A chamada da operação é usada como um indício, um modo de encontrar requisitos de teste que exercitem o código que chama.

> "Se você quer e espera que um programa funcione, é mais provável que veja um programa funcionando — você não vai encontrar as falhas."
>
> Cem Kaner *et al.*

14.7.4 Casos de Teste e Hierarquia de Classes

PONTO CHAVE

Mesmo que uma classe-base tenha sido exaustivamente testada, você ainda vai ter que testar todas as classes derivadas dela.

A herança não torna óbvia a necessidade de teste aprofundado de todas as classes derivadas. Na realidade, ela pode complicar o processo de teste. Considere a seguinte situação. Uma classe **Base** contém operações *inherited()* e *redefined()*. Uma classe **Derived** redefine *redefined()* para servir em um contexto local. É quase certo que **Derived::redefined()** tem que ser testado, porque representa um projeto novo e um código novo. Mas **Derived::inherited()** tem que ser retestado?

Se **Derived::inherited()** chama *redefined()* e o comportamento de *redefined()* sofreu modificação, **Derived::inherited()** pode processar errado o novo comportamento. Assim sendo, precisa de novos testes mesmo que o projeto e o código não tenham sido modificados. É importante notar, no entanto, que apenas um subconjunto de todos os testes para **Derived::inherited()** talvez tenha que ser conduzido. Se parte do projeto e do código para *inherited()* não depende de *redefined()* (isto é, não o chama nem chama nenhum código que o chame indiretamente), o código não precisa ser retestado na classe derivada.

Base::redefined() e **Derived::redefined()** são duas operações diferentes, com diferentes especificações e implementações. Cada uma terá um conjunto de requisitos de teste derivado da especificação e implementação. Esses requisitos de teste procuram erros plausíveis: erros de integração, erros de condição, erros de fronteira e assim por diante. Mas as operações são provavelmente similares. Seus conjuntos de requisitos de teste vão se superpor. Quanto melhor o projeto OO, maior é a superposição. Novos testes precisam ser originados apenas para aqueles requisitos **Derived::redefined()** que não são satisfeitos pelos testes **Base::redefined()**.

Para resumir, os testes **Base::redefined()** são aplicados aos objetos da classe **Derived**. Entradas de teste podem ser apropriadas tanto para a classe-base quanto à derivada, mas os resultados esperados podem diferir na classe derivada.

14.7.5 Teste com Base em Cenário

O teste baseado em erro deixa de encontrar dois tipos principais de erro: (1) especificações incorretas e (2) interações entre subsistemas. Quando ocorrem erros associados com especificações incorretas, o produto não faz o que o cliente deseja. Pode fazer a coisa errada ou pode omitir funcionalidade importante. Mas, em qualquer circunstância, a qualidade (conformidade com os requisitos) sofre. Ocorrem erros associados com a interação de subsistemas quando o comportamento de um subsistema cria circunstâncias (por exemplo, eventos, fluxo de dados) que provoquem a falha do outro subsistema.

PONTO CHAVE

O teste baseado em cenário vai descobrir erros que ocorrem quando qualquer ator interage com o software.

O *teste baseado em cenário* concentra-se no que o usuário faz, não no que o produto faz. Isso significa detectar as tarefas (por meio de casos de uso) que o usuário precisa realizar, depois aplicá-las, bem como suas variantes, aos testes.

Cenários descobrem erros de interação. Mas, para conseguir isso, os casos de teste precisam ser mais complexos e mais realísticos do que os testes baseados em erro. O teste baseado em cenário tende a exercitar múltiplos subsistemas em um único teste (os usuários não se limitam ao uso de um subsistema de cada vez).

Como exemplo, considere o projeto de testes baseados em cenário para um editor de texto. Os casos de uso são apresentados em seguida:

Caso de uso: *Consertar o Esboço Final*

Background: é comum imprimir o esboço "final", lê-lo e descobrir alguns erros evidentes, que não estavam óbvios na imagem de tela. Este caso de uso descreve a seqüência de eventos que ocorre quando isso acontece.

1. Imprime todo o documento.
2. Percorre o documento mudando algumas páginas.
3. Cada página é impressa à medida que é modificada.
4. Algumas vezes uma série de páginas é impressa.

Esse cenário descreve duas coisas: o teste e necessidades específicas do usuário. As necessidades do usuário são óbvias: (1) um método para imprimir páginas separadas e (2) um método para imprimir uma série de páginas. Do ponto de vista do teste, há necessidade de testar a edição depois da impressão (e vice-versa). O testador espera descobrir que a função impressão causa erros na função edição; isto é, que as duas funções do software não são adequadamente independentes.

AVISO

Apesar de o teste baseado em cenário ter mérito, você terá um retorno mais alto em relação ao tempo investido revisando os casos de uso quando são desenvolvidos como parte do modelo de análise.

Caso de uso: *Imprimir uma Nova Cópia*

Background: alguém pede ao usuário uma nova cópia do documento. Ela precisa ser impressa.

1. Abre o documento.
2. Imprime-o.
3. Fecha o documento.

Novamente, a abordagem de teste é relativamente óbvia, exceto que esse documento não apareceu do nada. Foi criado em uma tarefa anterior. Essa tarefa afeta a presente?

Em muitos editores modernos, os documentos lembram-se de como foram impressos da última vez. Por *padrão*, são impressos do mesmo modo na vez seguinte. Depois do cenário *Consertar o Esboço Final*, selecionar Imprimir no menu e clicar no botão "Imprimir", na caixa de diálogo, fará que a última página corrigida seja impressa novamente. Assim, de acordo com o editor, o cenário correto deveria ser o seguinte:

Caso de uso: *Imprimir uma Nova Cópia*

1. Abre o documento.
2. Seleciona "Imprimir" no menu.
3. Verifica se está imprimindo uma série de páginas; em caso positivo, clica para imprimir todo o documento.
4. Clica no botão Imprimir.
5. Fecha o documento.

Mas esse cenário indica um erro de especificação potencial. O editor não faz aquilo que o usuário de certa forma espera que ele faça. Os clientes freqüentemente vão esquecer a verificação mencionada no passo 3. Vão ficar chateados quando se dirigirem à impressora e encontrarem uma página, quando esperavam encontrar 100. Clientes chateados é sinal de erros de especificação.

Document shared on www.docsity.com

Um projetista de casos de teste poderia não perceber essa dependência no projeto de teste, mas é provável que o problema surja durante o teste. O testador teria então que se defrontar com a resposta provável "esse é o jeito que deve funcionar!".

14.7.6 Teste da Estrutura Superficial e da Estrutura Profunda

A estrutura superficial se refere à estrutura de um programa OO externamente observável, isto é, a estrutura que é imediatamente óbvia a um usuário final. Em vez de realizar funções, os usuários de muitos sistemas OO podem receber objetos para manipular de algum modo. Mas qualquer que seja a interface, os testes ainda são baseados nas tarefas do usuário. Captar essas tarefas envolve entender, observar e falar com usuários representativos (e tantos usuários não-representativos quanto valer a pena considerar).

PONTO CHAVE

O teste da estrutura superficial é análogo ao teste caixa-preta. O teste da estrutura profunda é semelhante ao teste caixa-branca.

Certamente haverá alguma diferença no detalhe. Por exemplo, em um sistema convencional com uma interface orientada a comandos, o usuário pode usar a lista de todos os comandos como uma verificação de teste. Se não existirem cenários de teste para exercitar um comando, o teste provavelmente não percebeu algumas tarefas do usuário (ou a interface tem comandos inúteis). Em uma interface baseada em objetos, o testador pode usar a lista de todos os objetos como uma verificação de teste.

Os melhores testes são originados quando o projetista olha para o sistema de um modo novo ou não convencional. Por exemplo, se o sistema ou o produto tem uma interface baseada em comando, serão originados testes mais vigorosos se o projetista de casos de teste fizer de conta que as operações são independentes dos objetos. Faça perguntas como "o usuário pode querer usar essa operação — quando aplica apenas o objeto **Scanner** — enquanto está trabalhando com a impressora?". Qualquer que seja o estilo da interface, o projeto de casos de teste que exercitam a estrutura superficial deve usar tanto os objetos quanto as operações como indícios que levem a tarefas despercebidas.

A estrutura profunda refere-se aos detalhes técnicos internos de um programa OO. Isto é, a estrutura que é entendida pelo exame do projeto e/ou código. O teste da estrutura profunda é projetado para exercitar dependências, comportamentos e mecanismos de comunicação que tiverem sido estabelecidos como parte do projeto do sistema e de objetos (Capítulos 9 ao 12) do software OO.

Os modelos de análise e projeto são usados como base para o teste de estrutura profunda. Por exemplo, o diagrama de colaboração UML ou o modelo de implantação mostram as colaborações entre objetos e subsistemas que podem não ser externamente visíveis. Então, o projetista de casos de teste pergunta: "Detectamos (como teste) alguma tarefa que exercita a colaboração mencionada no diagrama de colaboração? Em caso negativo, por que não?".

> "Não se envergonhe de erros, e, assim, os transforme em crimes."
>
> Confúcio

14.8 Métodos de Teste Aplicáveis ao Nível de Classe

AVISO

O número de permutações possíveis para o teste aleatório pode ficar muito grande. Uma estratégia semelhante ao teste de matriz ortogonal pode ser usada para melhorar a eficiência de teste.

No Capítulo 13, mencionamos que o teste de software começa "no varejo" e lentamente progride em direção ao teste "por atacado". O teste no varejo focaliza uma única classe e os métodos que são encapsulados pela classe. O teste aleatório e de partição são métodos que podem ser usados para exercitar uma classe durante o teste OO [KIR94].

14.8.1 Teste Aleatório para Classes OO

Para dar ilustrações resumidas desses métodos, considere uma aplicação bancária na qual uma classe **Conta** tem as seguintes operações: *abri()r, estabelece()r, depositar(), retirar(), obter saldo(), resumir(), limite de crédito()* e *fechar()* [KIR94]. Cada uma dessas operações pode ser aplicada à **Conta**, mas certas restrições (por exemplo, a conta precisa ter sido aberta antes que outras operações possam ser aplicadas e fechada depois de todas as operações terem sido completadas) são

implícitas pela natureza do problema. Mesmo com essas restrições, há muitas permutações das operações. O histórico de vida comportamental mínimo de um exemplo de **Conta** inclui as seguintes operações:

abrir • estabelecer • depositar • retirar • fechar

Isso representa a seqüência mínima de teste para **Conta**. No entanto, uma grande variedade de outros comportamentos pode ocorrer dentro dessa seqüência:

abrir • estabelecer • depositar • [depositar | retirar | obter | saldo | resumir | limite de crédito]^n • retirar • fechar

Uma variedade de diferentes seqüências de operação pode ser gerada aleatoriamente. Por exemplo:

Caso de teste r_1: **abrir • estabelecer • depositar • depositar • obter saldo • resumir • retirar • fechar**
Caso de teste r_2: **abrir • estabelecer • depositar • retirar • depositar • obter saldo • limite de crédito • retirar • fechar**

Esses e outros testes, em ordem aleatória, são conduzidos para exercitar diferentes históricos de vida de exemplos da classe.

CASASEGURA

Teste de Classe

A cena: cubículo de Shakira.

Os personagens: Jamie e Shakira — membros da equipe de engenharia de software do *CasaSegura* que estão trabalhando no projeto de casos de teste para a função de segurança.

A conversa:

Shakira: Eu desenvolvi alguns testes para a classe **Detector** [Figura 11.4] — você sabe, aquela que permite acesso a todos os objetos **Sensor** para a função de segurança. Você está familiarizado com ela?

Jamie (rindo): Claro, é aquela que lhe permitiu adicionar o sensor "raiva de cachorro".

Shakira: O próprio. De qualquer maneira, ele tem uma interface com quatro operações *ler(), habilitar(), desabilitar()* e *testar()*. Antes que um sensor possa ser lido, precisa ser habilitado. Uma vez habilitado, pode ser lido e testado. Pode ser desabilitado a qualquer momento, exceto se uma condição de alarme estiver sendo processada. Então, eu defini um único teste de seqüência que vai exercitar sua história de vida comportamental.

(Mostra a Jamie a seguinte seqüência:)

1: habilitar • testar • ler • desabilitar

Jamie: Vai funcionar, mas você terá que fazer mais testes do que isso!

Shakira: Eu sei, eu sei. Aqui estão algumas outras seqüências que eu fiz.

(Ela mostra a Jamie as seguintes seqüências:)

2: habilitar • testar • [ler]^n • testar • desabilitar

3: [ler]^n

4: habilitar • desabilitar • [testar | ler]

Jamie: Então, deixe-me ver se entendi o objetivo dessas seqüências. 1 vai pela história de vida normal, uma espécie de uso convencional. 2 repete a operação de leitura *n* vezes e esse é um provável cenário. 3 tenta ler o sensor antes de ele estar habilitado... que deve produzir uma mensagem de erro de alguma espécie, certo? 4 habilita e desabilita o sensor e depois tenta lê-lo. Isso não é o mesmo que o teste 3?

Shakira: Na verdade, não. Em 4, o sensor foi habilitado. O que 4 realmente testa é se a operação desabilitar trabalha como deveria. Uma *ler()* ou *testar()* depois de *desabilitar()* deveria gerar uma mensagem de erro. Se isso não ocorrer, então tenho um erro na operação desabilitar.

Jamie: Bom. Lembre-se, porém, de que os quatro testes têm de ser aplicados a cada tipo de sensor, pois todas as operações podem ser sutilmente diferentes, dependendo do tipo do sensor.

Shakira: Não se preocupe. Esse é o plano.

14.8.2 Teste de Partição no Nível de Classe

O *teste de partição* reduz o número de casos de teste necessários para exercitar a classe de um modo semelhante ao da partição de equivalência (Seção 14.6.2) para software convencional. Entrada e saída são categorizadas e casos de teste são projetados para exercitar cada categoria. Mas como as categorias de partição são derivadas?

Document shared on www.docsity.com

? Que opções de teste estão disponíveis no nível de classe?

A *partição baseada em estado* categoriza as operações de classe com base na sua capacidade de mudar o estado da classe. Novamente, considerando a classe **Conta**, as operações de estado incluem *fazerDepósito()* e *fazerRetirada()*, enquanto as operações de consulta incluem *obterSaldo()*, *obterResumo()* e *obterLimiteDeCrédito()*. Os testes são projetados de modo que exercitem, separadamente, as operações que mudam o estado e aquelas que não mudam o estado. Assim,

Caso de teste p_1: **abrir • estabelecer • fazerDepósito • fazerDepósito • fazerRetirada • fazerRetirada • fechar**

Caso de teste p_2: **abrir • estabelecer • fazerDepósito • obterResumo • obterLimiteDeCrédito • fazerRetirada • fechar**

O caso de teste p_1 modifica o estado, enquanto o caso de teste p_2 exercita operações que não modificam o estado (além daquelas da seqüência mínima de teste).

A *partição baseada em atributo* categoriza operações de classe com base nos atributos que elas usam. Para a classe **Conta**, os atributos **saldo** e **limiteDeCrédito** podem ser usados para definir partições. As operações são divididas em três partições: (1) operações que usam **limiteDeCrédito**, (2) operações que modificam **limiteDeCrédito** e (3) operações que não usam nem modificam **limiteDeCrédito**. Seqüências de teste são então projetadas para cada partição.

A *partição baseada em categoria* categoriza as operações de classe com base na função genérica que cada uma realiza. Por exemplo, operações da classe **Conta** podem ser categorizadas em operações de inicialização — *abrir()*, *estabelecer()*, operações computacionais — (*fazerDepósito()*, *fazerRetiradas()*, consultas — *obterSaldo()*,*obterResumo()*,*obterLimiteDeCrédito()* e operações de término — *fechar()*.

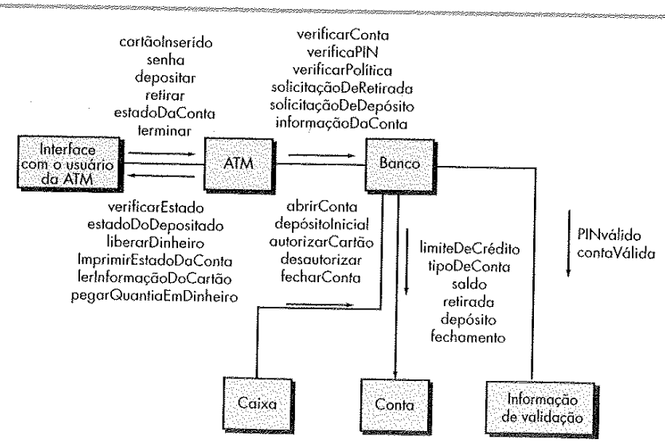
14.9 Projeto de Casos de Teste Interclasse

O projeto de casos de teste torna-se mais complicado quando a integração do sistema orientado a objetos tem início. É nesse estágio que deve começar o teste das colaborações entre classes. Para ilustrarmos "a geração de casos de teste interclasses" [KIR94], ampliamos o exemplo bancário introduzido na Seção 14.8 para incluir as classes e colaborações mencionadas na Figura 14.11. O sentido das setas na figura indica o sentido das mensagens, e os rótulos indicam as operações que são chamadas como conseqüência das colaborações implicadas pelas mensagens.

Como o teste das classes individuais, o teste de colaboração de classes pode ser conseguido aplicando-se os métodos aleatório e de particionamento, bem como o teste baseado em cenário e teste comportamental.

FIGURA 14.11

Diagrama de colaboração de classes para aplicação bancária (adapatada de [KIR94])



14.9.1 Teste de Várias Classes

Kirani e Tsai [KIR94] sugerem a seguinte seqüência de passos para gerar casos de teste aleatórios para várias classes:

1. Para cada classe cliente, use a lista de operações de classe para gerar uma série de seqüências de teste aleatório. As operações vão enviar mensagens para outras classes servidoras.
2. Para cada mensagem gerada, determine a classe colaboradora e a operação correspondente no objeto servidor.
3. Para cada operação no objeto servidor (que foi invocado por mensagens enviadas pelo objeto cliente), determine as mensagens que ela transmite.
4. Para cada uma das mensagens, determine o nível seguinte de operações que são invocadas e incorpore-as à seqüência de teste.

Para ilustrar [KIR94], considere uma seqüência de operações para a classe **Banco** relativa a uma classe **ATM** (Figura 14.11):

verificarConta • verificarPIN • [[verificarDiretrizes • requisiçãoDeRetirada] | requisiçãoDeDepósito | requisiçãoDeInformaçãoDeConta]^n

Um caso de teste aleatório para a classe **Banco** poderia ser

Caso de teste r_3 = **verificarConta • verificarPIN • requisiçãoDeDepósito**

A fim de considerar os colaboradores envolvidos nesse teste, as mensagens associadas com cada uma das operações mencionadas no caso de teste r_3 são apreciadas. **Banco** deve colaborar com **InformaçãoDeValidação** para executar *verificarConta()* e *verificarPIN()*. Banco precisa colaborar com **Conta** para executar *requisiçãoDeDepósito*. Assim sendo, um novo caso de teste que exercita essas colaborações é

Caso de teste r_4 = **verificarContaBanco[contaVálidaInformaçãoDeValidação] • verificarPINBanco • [PINválidoInformaçãoDeValidação] • requisiçãoDeDepósito • [depósitoconta]**

A abordagem para o teste de partição de várias classes é semelhante à abordagem usada para o teste de partição de classes individuais. Uma única classe é particionada, como discutido na Seção 14.8.2. No entanto, a seqüência de teste é expandida para incluir aquelas operações invocadas por meio de mensagens para as classes colaboradoras. Uma abordagem alternativa particiona os testes com base nas interfaces de uma classe particular. Observando a Figura 14.11, a classe **Banco** recebe mensagens das classes **ATM** e **Caixa**. Os métodos de **Banco** podem conseqüentemente ser testados pelo seu particionamento quanto aos que servem **ATM** e os que servem **Caixa**. O particionamento baseado em estado (Seção 14.8.2) pode ser usado para refinar mais as partições.

14.9.2 Testes Derivados dos Modelos de Comportamento

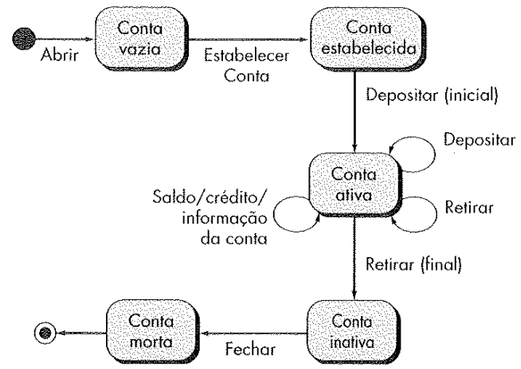
No Capítulo 8 discutimos o uso do diagrama de estados como o modelo que representa o comportamento dinâmico de uma classe. O diagrama de estado de uma classe pode ser usado para ajudar a originar as seqüências de testes que vão exercitar o comportamento dinâmico da classe (e daquelas classes que colaboram com ela). A Figura 14.12 [KIR94] ilustra um diagrama de estado para a classe **Conta** discutida anteriormente. Observando a figura, as transições iniciais movem-se entre os estados *conta vazia* e *conta estabelecida*. A maior parte de todo o comportamento para exemplos da classe ocorre enquanto se está no estado *conta ativa*. Uma retirada final e fechamento da conta para a classe **Conta** fazem que se estabeleçam transições para os estados *conta inativa* e *conta morta*, respectivamente.

Os testes a ser projetados devem cobrir todos os estados [KIR94]. Isto é, as seqüências de operações devem fazer a classe **Conta** realizar transição entre todos os estados permitidos:

Document shared on www.docsity.com

FIGURA 14.12

Diagrama de estado para a classe Conta (adaptada de [KIR94])



Caso de teste s_1: **abrir • estabelecerConta • fazerDepósito(inicial) • fazerRetirada (final) • fechar**

Deve-se notar que essa seqüência é idêntica à seqüência mínima de teste discutida na Seção 14.9.1. Acrescentando seqüências de teste à seqüência mínima:

Caso de teste s_2: **abrir • estabelecerConta • fazerDepósito(inicial) • fazerDepósito • obterSaldo • obterLimiteDeCrédito • fazerRetirada(final) • fechar**

Caso de teste s_3: **abrir • estabelecerConta • fazerDepósito(inicial) • fazerDepósito • fazerRetirada • informaçãoDaConta • fazerRetirada(final) • fechar**

Mais casos de teste poderiam ainda ser derivados para garantir que todos os comportamentos da classe tenham sido adequadamente exercitados. Em situações nas quais o comportamento da classe resulta em uma colaboração com uma ou mais classes, diagramas de estado múltiplos são usados para rastrear o fluxo de comportamento do sistema.

O modelo de estados pode ser percorrido em forma de "inclusão progressiva" [MGR94]. Nesse contexto, inclusão progressiva implica um caso de teste exercitar uma única transição e, quando uma nova transição tiver de ser testada, são usadas apenas aquelas previamente testadas.

Considere o objeto **CartãoDeCrédito** que é parte do sistema bancário. O estado inicial de **CartãoDeCrédito** é *indefinido* (isto é, nenhum número de cartão de crédito foi fornecido). Ao ler o cartão de crédito durante uma venda, o objeto assume um estado *definido;* isto é, os atributos **número do cartão** e **data da expiração**, juntamente com identificadores específicos do banco, são definidos. O cartão de crédito fica *submetido* quando é enviado para autorização e fica *aprovado* quando a autorização é recebida. A transição de **CartãoDeCrédito** de um estado para outro pode ser testada originando casos de teste que causem a ocorrência da transição. Uma abordagem de inclusão progressiva para esse tipo de teste não exercitaria *submetido* antes de exercitar *indefinido* e *definido*. Se o fizesse, estaria usando transições que não foram previamente testadas e, conseqüentemente, violaria o critério de inclusão progressiva.

14.10 TESTE DE AMBIENTES, ARQUITETURAS E APLICAÇÕES ESPECIALIZADAS

Os métodos de teste discutidos nas seção anteriores são geralmente aplicáveis em todos os ambientes, arquiteturas e aplicações, mas diretrizes e abordagens especiais para teste estão algumas vezes disponíveis. Nesta seção, consideramos diretrizes para teste de ambientes, arquiteturas e aplicações especializadas comumente encontradas por engenheiros de software.

AVISO

Uma estratégia de teste similar ao teste aleatório ou de partição (Seção 14.8) pode ser usada para projetar testes de IU.

14.10.1 Teste de IGU

Interfaces gráficas com o usuário (IGU, *graphical user interfaces*, GUI) apresentam desafios interessantes para os engenheiros de software. Devido a componentes reusáveis fornecidos como parte de ambientes de desenvolvimento de IGU, a criação da interface com o usuário tornou-se menos demorada e mais precisa (Capítulo 12). Mas, ao mesmo tempo, a complexidade das IGU cresceu, levando à mais dificuldade no projeto e na execução de casos de teste.

Como muitas IGU modernas têm a mesma aparência e funcionamento, uma série de testes padrão pode ser derivada. Grafos de modelagem de estados finitos podem ser usados para derivar uma série de testes que tratam de objetos de dados e programa específicos, relevantes para a IGU.

Devido ao grande número de permutações associadas com as operações IGU, o teste deve ser conduzido usando-se ferramentas automatizadas. Uma grande variedade de ferramentas de teste de IGU surgiu no mercado nos últimos anos. Para mais discussão, veja o Capítulo 12.

Veja na Web

Informação e recursos de teste cliente/servidor úteis podem ser encontrados em www.cst-technologies.com.

14.10.2 Teste de Arquiteturas Cliente/Servidor

Arquiteturas cliente/servidor representam um significativo desafio para os testadores de software. A natureza distribuída de ambientes cliente/servidor, os aspectos de desempenho, associados com processamento de transações, a presença potencial de várias plataformas de hardware diferentes, as complexidades de redes de comunicação, a necessidade de atender a muitos clientes por uma base de dados centralizada (ou em alguns casos distribuída) e os requisitos de coordenação impostos ao servidor, tudo se combina para tornar o teste de arquiteturas cliente/servidor e do software que nelas reside consideravelmente mais difícil do que de aplicações isoladas. De fato, estudos recentes na indústria indicam um significativo aumento de tempo e custo de teste quando ambientes cliente/servidor são desenvolvidos.

> "O tópico de teste é uma área na qual existe bastante coisa comum entre sistemas tradicionais e sistemas cliente/servidor."
>
> **Kelley Bourne**

Em geral, o teste de software cliente/servidor ocorre em três diferentes níveis: (1) aplicações clientes individuais são testadas no modo "não conectado"; a operação do servidor e a rede subjacente não são consideradas; (2) o software cliente e as aplicações do servidor associadas são testadas em conjunto, mas as operações da rede não são explicitamente exercitadas; (3) a arquitetura completa cliente/servidor, incluindo operações e desempenho da rede, é testada.

Embora muitos tipos diferentes de teste sejam conduzidos em cada um desses níveis de detalhe, as seguintes abordagens de teste são comumente encontradas para aplicações cliente/servidor:

? *Que tipos de testes são conduzidos em sistemas cliente/servidor?*

- **Teste de função da aplicação.** A funcionalidade de aplicações clientes é testada usando os métodos discutidos anteriormente neste capítulo. Na verdade, a aplicação é testada de modo isolado.
- **Testes de servidor.** As funções de coordenação e gestão de dados do servidor são testadas. O desempenho do servidor (tempo de resposta global e vazão de dados (*throughput*) é também considerado.
- **Testes de banco de dados.** A precisão e a integridade dos dados armazenados pelo servidor são testadas. As transações colocadas pelas aplicações cliente são examinadas para garantir que os dados sejam adequadamente armazenados, atualizados e recuperados. O arquivamento é também testado.
- **Testes de transação.** Uma série de testes é criada para garantir que cada classe de transações seja processada de acordo com os requisitos. Os testes enfocam a correção do processamento e também tópicos de desempenho (por exemplo, tempo de processamento de transação e volume de transação).

Document shared on www.docsity.com

- **Testes de comunicação em rede.** Esses testes verificam se a comunicação entre os nós da rede ocorre corretamente e se a passagem de mensagens, transações e tráfego de rede relacionado ocorre sem erro. Testes de segurança da rede podem também ser conduzidos como parte desses testes.

Para atingir essas abordagens de teste, Musa [MUS93] recomenda o desenvolvimento de *perfis operacionais* derivados de cenários de uso cliente/servidor.[9] Um perfil operacional indica como diferentes tipos de usuário interoperam com o sistema cliente/servidor. Isto é, os perfis fornecem um "padrão de uso" que pode ser aplicado quando os testes são projetados e executados.

14.10.3 Teste da Documentação e Dispositivos de Ajuda

O termo *teste de software* invoca imagens de grande número de casos de teste preparados para exercitar programas de computador e os dados que eles manipulam. Recordando a definição de *software* apresentada no primeiro capítulo deste livro, é importante notar que o teste deve também ser estendido para o terceiro elemento da configuração de software — documentação.

Erros na documentação podem ser tão devastadores para aceitação do programa quanto erros nos dados ou no código-fonte. Nada é mais frustrante que seguir exatamente um guia do usuário ou um documento de ajuda on-line e obter resultados ou comportamentos que não coincidem com aqueles previstos pela documentação. É por isso que o teste de documentação deve ser uma parte significativa de todo o plano de teste de software.

O teste de documentação pode ser abordado em duas fases. A primeira fase, revisão e inspeção (Capítulo 26), examina o documento quanto à clareza editorial. A segunda fase, teste ao vivo, usa a documentação em conjunto com o uso do programa real.

INFO

Teste de Documentação

As seguintes questões devem ser respondidas durante o teste de documentação e/ou de facilidades de ajuda:

- A documentação descreve precisamente como conseguir cada modo de uso?
- A descrição de cada seqüência de interação é precisa?
- Os exemplos são corretos?
- A terminologia, as descrições dos menus e as respostas do sistema são consistentes com o programa real? É relativamente fácil localizar diretrizes na documentação?
- A correção de defeitos pode ser realizada facilmente com a documentação?
- O sumário e o índice do documento são precisos e completos?
- O projeto do documento (leiautes, escolha de tipos, tabulação, gráficos) conduz ao entendimento e à rápida assimilação da informação?
- As mensagens de erro do software mostradas para o usuário são todas descritas em mais detalhes no documento? As ações a ser tomadas como conseqüência de uma mensagem de erro são claramente delineadas?
- Se são usadas ligações de hipertexto, elas são precisas e completas?
- Se é usado hipertexto, o projeto de navegação é apropriado à informação exigida?

O único modo viável de responder a essas questões é ter um parceiro independente (por exemplo, usuários selecionados) testando a documentação no contexto de uso do programa. Todas as discrepâncias são anotadas e as áreas de ambigüidade ou imperfeição do documento são definidas para serem reescritas novamente.

14.10.4 Teste de Sistemas de Tempo Real

A natureza dependente de tempo e assíncrona, de muitas aplicações em tempo real, adiciona um elemento novo e potencialmente difícil à mistura do teste — tempo. O projetista de casos de teste

9 Deve-se notar que os perfis operacionais podem ser usados no teste de todos os tipos de arquitetura de sistema, não apenas cliente/servidor.

não tem apenas que considerar casos de teste convencionais, mas também a manipulação de eventos (isto é, processamento de interrupções), a tempestividade dos dados e o paralelismo das tarefas (processos) que manipulam os dados. Em muitas situações, dados de teste fornecidos vão resultar em processamento adequado, quando um sistema de tempo real está em um estado, enquanto os mesmos dados fornecidos quando o sistema está em um estado diferente podem levar a erro.

Por exemplo, o software de tempo real que controla uma nova fotocopiadora aceita interrupções do operador (isto é, o operador da máquina pressiona teclas de controle tais como RESTAURAR ou ESCURECER), sem erro, quando a máquina está fazendo cópias (no estado *copiando*). Se essas mesmas interrupções do operador são efetuadas quando a máquina está no estado *emperrada*, a exibição de um código de diagnóstico (indicando a posição onde está emperrada) será perdida (um erro).

Além disso, a relação íntima que existe entre software de tempo real e seu ambiente de hardware pode também causar problemas de teste. Testes de software devem considerar o impacto de falhas do hardware no processamento do software. Tais falhas podem ser extremamente difíceis de simular com realismo.

? Qual é uma estratégia efetiva para teste de sistema de tempo real?

Métodos de projeto de casos de teste abrangentes para sistemas de tempo real continuam a evoluir. No entanto, uma estratégia de quatro passos pode ser proposta:

- **Teste de tarefa.** O primeiro passo no teste de software de tempo real é testar cada tarefa independentemente, isto é, testes convencionais são projetados e executados para cada tarefa. Cada tarefa é executada independentemente durante esses testes. O teste de tarefa detecta erros de lógica e de função, mas não de tempestividade ou de comportamento.
- **Teste comportamental.** Usando modelos do sistema criados com ferramentas automatizadas, é possível simular o comportamento de um sistema de tempo real e examinar seu comportamento como conseqüência de eventos externos. Essas atividades de análise podem servir de base para o projeto de casos de teste conduzidos depois de o software de tempo real ter sido construído.
- **Testes intertarefas.** Uma vez que erros em tarefas individuais e no comportamento do sistema foram isolados, o teste passa para erros relativos a tempo. Tarefas assíncronas que se comunicam umas com as outras são testadas com diferentes taxas de dados e carga de processamento para detectar se erros de sincronização intertarefas vão ocorrer. Além disso, tarefas que se comunicam por fila de mensagens ou depósito de dados são testadas para descobrir erros no tamanho dessas áreas de armazenamento.
- **Teste de sistema.** O software e o hardware são integrados e todo um conjunto de testes de sistema (Capítulo 13) é conduzido em uma tentativa de descobrir erros na interface software/hardware. A maioria dos sistemas de tempo real processa interrupções. Assim, o teste da manipulação desses eventos booleanos é essencial. Usando o diagrama de estados e a especificação de controle (Capítulo 8), o testador desenvolve uma lista de todas as possíveis interrupções e do processamento que ocorre como conseqüência das interrupções. Então, são projetados testes para avaliar as seguintes características do sistema:
  - — As prioridades de interrupção estão atribuídas adequadamente e propriamente manipuladas?
  - — O processamento para cada interrupção foi conduzido corretamente?
  - — O desempenho (por exemplo, tempo de processamento) de cada procedimento de manipulação de interrupção está de acordo com os requisitos?
  - — Um alto volume de interrupções chegando em tempos críticos cria problemas de função ou desempenho?

Além disso, áreas globais de dados, usadas para transferir informação como parte do processamento de interrupções, devem ser testadas para avaliar o potencial de geração de efeitos colaterais.

Document shared on www.docsity.com

14.11 Padrões de Teste

Veja na Web

Ponteiros para mais 70 padrões de teste podem ser encontrados em www.rbsc.com.

Em capítulos anteriores, discutimos o uso de padrões como um mecanismo para descrever blocos construtivos de software ou situações de engenharia de software. Esses blocos construtivos ou situações são encontrados repetidamente à medida que diferentes aplicações são construídas ou diferentes projetos são conduzidos. Como suas contrapartes em análise e projeto, *padrões de teste* descrevem blocos construtivos ou situações freqüentemente encontradas que testadores de software podem conseguir reusar quando eles abordam o teste de algum sistema novo ou revisado.

Padrões de teste não apenas fornecem aos engenheiros de software diretrizes úteis quando as atividades de teste começam, mas também fornecem três benefícios adicionais descritos por Marick [MAR02]

PONTO CHAVE

Padrões de teste podem ajudar uma equipe de software a se comunicar mais efetivamente sobre teste e entender melhor as influências que levam a abordagens de teste específicas.

1. Fornecem um vocabulário para resolvedores de problemas. "Você sabia que poderíamos usar um Objeto Nulo?"
2. Concentram a atenção nas influências por trás do problema. Isso permite que projetistas [de casos de teste] entendam melhor quando e por que uma solução se aplica.
3. Incentivam o raciocínio iterativo. Cada solução cria um contexto no qual novos problemas podem ser resolvidos.

Apesar de esses benefícios serem "tênues" (*soft*), eles não devem ser desprezados. Muito do teste de software, mesmo durante a década passada, foi uma atividade *ad hoc*. Se padrões de teste podem ajudar uma equipe de software a se comunicar mais efetivamente sobre teste, entender as influências motivadoras que levam a uma abordagem específica de teste e abordam o projeto de casos de teste como uma atividade evolutiva, eles conseguiram muito.

Padrões de teste são descritos de modo semelhante aos padrões de análise e projeto (Capítulos 7 e 9). Dezenas de padrões de teste têm sido propostos na literatura (por exemplo, [BIN99], [MAR02]). Os três padrões de teste seguintes (apresentados apenas de forma resumida) fornecem exemplos representativos:

Veja na Web

Padrões que descrevem organização, eficiência, estratégia e resolução de problemas de teste podem ser encontrados em www.agcs.com/supportv2/techpapers/patterns/papers/systestp.htm.

Nome do padrão: **teste aos pares**

Resumo: Um padrão orientado a processo, **teste aos pares,** descreve uma técnica análoga à programação aos pares (Capítulo 4), na qual dois testadores trabalham juntos para projetar e executar uma série de testes que pode ser aplicada a atividades de teste de unidade, integração ou validação.

Nome do padrão: **interface de teste separada**

Resumo: Há necessidade de testar cada classe em um sistema orientado a objetos, inclusive "classes internas" (classes que não expõem nenhuma interface para fora do componente que as usa). O padrão **interface de teste separada** descreve como criar "uma interface de teste que pode ser usada para descrever testes específicos das classes visíveis apenas internamente a um componente" [LAN01].

Nome do padrão: **teste de cenário**

Resumo: Uma vez conduzidos testes de unidade e de integração, há necessidade de determinar se o software terá desempenho que satisfaça aos usuários. O padrão **teste de cenário** descreve uma técnica para exercitar o software do ponto de vista do usuário. Uma falha nesse nível indica que o software deixou de satisfazer a um requisito visível ao [KAN01].

Uma abrangente discussão de padrões de teste está além do escopo deste livro. O leitor interessado pode ver [BIN99] e [MAR02] para mais informação sobre esse importante tópico.

14.12 Resumo

O objetivo principal do projeto de casos de teste é originar um conjunto de testes que tenha a maior probabilidade de detectar erros no software. Para alcançar esse objetivo, duas diferentes categorias de técnicas de projeto de casos de teste — aplicáveis a sistemas convencionais e orientados a objetos — são usadas: teste caixa-branca e teste caixa-preta.

O teste caixa-branca enfoca a estrutura de controle do programa. Casos de testes são originados para garantir que todos os comandos do programa tenham sido executados pelo menos uma vez durante o teste e que todas as condições lógicas tenham sido exercitadas. O teste de caminho básico, uma técnica caixa-branca, faz uso de grafos de programa (ou matrizes de grafos) para originar um conjunto de testes linearmente independentes que vão garantir a cobertura. Os testes de condição e de fluxo de dados exercitam adicionalmente a lógica do programa e os testes de ciclo complementam outras técnicas caixa-branca, fornecendo um procedimento para exercitar ciclos com vários graus de complexidade.

Os testes caixa-preta são projetados para validar os requisitos funcionais sem se prender ao funcionamento interno de um programa. As técnicas de teste caixa-preta focalizam o domínio de informação do software, originando casos pelo particionamento dos domínios de entrada e saída de um programa, de um modo que forneça rigorosa cobertura de teste. O particionamento de equivalência divide o domínio de entrada em classes de dados que provavelmente exercitam função específica do software. A análise de valor-limite investiga a habilidade do programa de manipular dados no limite de aceitabilidade. Teste de matriz ortogonal fornece um método eficiente e sistemático para testar sistemas com pequeno número de parâmetros de entrada.

Embora o objetivo global do teste orientado a objetos — encontrar o número máximo de erros com um mínimo de esforço — seja idêntico ao objetivo do teste de software convencional, a estratégia e a tática do teste OO diferem um pouco. A visão do teste se amplia para incluir a revisão tanto do modelo de análise quanto de projeto. Além disso, o enfoque do teste se afasta do componente procedimental (módulo) e vai em direção à classe.

O projeto de testes para uma classe usa uma variedade de métodos: teste baseado em erro, teste aleatório e teste de partição. Cada um desses métodos exercita as operações encapsuladas pela classe. Seqüências de testes são projetadas para garantir que as operações relevantes sejam exercitadas. O estado da classe, representado pelos valores dos seus atributos, é examinado para determinar se existem erros.

O teste de integração pode ser realizado usando-se uma estratégia baseada em uso. O teste baseado em uso constrói o sistema em camadas, começando com aquelas classes que não utilizam classes servidoras. Os métodos de projeto de casos de teste de integração podem também usar testes aleatórios e de partição. Além disso, o teste baseado em cenário e os testes derivados dos modelos de comportamento podem ser usados para testar uma classe e suas colaboradoras. Uma seqüência de testes rastreia o fluxo de operações ao longo das colaborações de classe.

Métodos especializados de teste abrangem uma grande variedade de capacidades do software e áreas de aplicação. O teste para interfaces gráficas com o usuário, arquitetura cliente/servidor, documentação e facilidades de ajuda, e sistemas de tempo real requerem em cada caso diretrizes e técnicas especializadas.

Desenvolvedores de software experientes freqüentemente dizem: "Teste nunca acaba, ele é simplesmente transferido de você [o engenheiro de software] para seu cliente. Toda vez que o seu cliente usa o programa, um teste está sendo conduzido". Pela aplicação de projeto de casos de teste, o engenheiro de software pode conseguir um teste mais completo e conseqüentemente descobrir e corrigir o maior número de erros antes que os "testes do cliente" comecem.

Referências Bibliográficas

[AMB95] Ambler, S., "Using Use Cases", *Software Development*, jul. 1995, p. 53-61.
[BEI90] Beizer, B., *Software Testing Techniques*, 2ª ed., Van Nostrand-Reinhold, 1990.

Document shared on www.docsity.com

[BEI95] _____, *Black-Box Testing*, Wiley, 1995.
[BIN94] Binder, R. V., "Testing Object-Oriented Systems: A Status Report", *American Programmer*, v. 7, n. 4, abr. 1994, p. 23-28.
[BIN99] Binder, R., *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley, 1999.
[DEU79] Deutsch, M., "Verification and Validation". In: *Software Engineering* (R. Jensen e Tonies, eds.), Prentice-Hall, 1979, p. 329-408.
[FRA88] Frankl, P. G. e Weyuker, E. J., "An Applicable Family of Data Flow Testing Criteria", *IEEE Trans. Software Engineering*, v. SE-14, n. 10, out. 1988, p. 1483-1498.
[FRA93] _____, e Weiss, S., "An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow", *IEEE Trans. Software Engineering*, v. SE-19, n. 8, ago. 1993, p. 770-787.
[KAN93] Kaner, C., Falk J., e Nguyen, H. Q., *Testing Computer Software*, 2ª ed., Van Nostrand-Reinhold, 1993.
[KAN01] _____, "Pattern. Scenario Testing", (draft), 2001, disponível em http://www.testing.com/test-patterns/patterns/pattern-scenario-testing-kaner.html.
[KIR94] Kirani, S. e Tsai, W. T., "Specification and Verification of Object-Oriented Programs", Technical Report TR 94-64, Computer Science Department, University of Minnesota, dez. 1994.
[LAN01] Lange, M., "It's Testing Time! Patterns for Testing Software", jun. 2001, disponível em http://www.testing.com/test-patterns/patterns/index.html.
[LIN94] Lindland, O. I. *et al.*, "Understanding Quality in Conceptual Modeling", *IEEE Software*, v. 11, n. 4, jul. 1994, p. 42-49.
[MAR94] Marick, B., *The Craft of Software Testing*, Prentice-Hall, 1994.
[MAR02] _____, "Software Testing Patterns", 2002, http://www.testing.com/test-patterns/index.html.
[MCC76] McCabe, T., "A Software Complexity Measure", *IEEE Trans. Software Engineering*, v. SE-2, dez. 1976, p. 308-320.
[MGR94] McGregor, J. D. e Korson, T. D., "Integrated Object-Oriented Testing and Development Processes", *CACM*, v. 37, n. 9, set. 1994, p. 59-77.
[MUS93] Musa, J., "Operational Profiles in Software Reliability Engineering", *IEEE Software*, mar. 1993, p. 14-32.
[MYE79] Myers, G., *The Art of Software Testing*, Wiley, 1979.
[NTA88] Ntafos, S. C., "A Comparison of Some Structural Testing Strategies", *IEEE Trans. Software Engineering*, v. SE-14, n. 6, jun. 1988, p. 868-874.
[PHA89] Phadke, M. S., *Quality Engineering Using Robust Design*, Prentice-Hall, 1989.
[PHA97] _____, "Planning Efficient Software Tests", *Crosstalk*, v. 10, n. 10, out. 1997, p. 11-15.
[TAI89] Tai, K. C., "What to Do Beyond Branch Testing", *ACM Software Engineering Notes*, v. 14, n. 2, abr. 1989, p. 58-61.

Problemas e Pontos a Considerar

14.1. Myers [MYE79] usa o seguinte programa para auto-avaliação de sua habilidade de especificar teste adequadamente. Um programa lê três valores inteiros. Os três valores são interpretados como representando os comprimentos dos lados de um triângulo. O programa imprime uma mensagem que declara se o triângulo é escaleno, isósceles ou equilátero. Desenvolva um conjunto de casos de teste que você ache adequado para testar esse programa.

14.2. Projete e implemente o programa (com manipulação de erros onde for apropriado) especificado no Problema 14.1. Crie um grafo de fluxo para o programa e aplique teste de caminho básico para desenvolver casos de teste que irão garantir que todos os comandos do programa sejam testados. Execute os casos e mostre seus resultados.

14.3. Você pode imaginar quaisquer outros objetivos de teste adicionais que não foram discutidos na Seção 14.1?

14.4. Selecione um componente de software que você tenha projetado e implementado recentemente. Projete um conjunto de casos de teste que garanta que todos os comandos tenham sido executados usando teste de caminho básico.

14.5. Especifique, projete e implemente uma ferramenta de software que calcule a complexidade ciclomática para a linguagem de programação de sua escolha. Use a matriz de grafo como a estrutura de dados do seu projeto.

14.6. Leia Beizer [BEI95] e determine como o programa que você desenvolveu no Problema 14.5 pode ser estendido para acomodar vários pesos de ligação. Estenda sua ferramenta para processar probabilidades de execução ou tempos de processamento de ligações.

14.7. Projete uma ferramenta automatizada que reconheça ciclos e os categorize como é indicado na Seção 14.5.3.

14.8. Estenda a ferramenta descrita no Problema 14.7 para gerar casos de teste para cada categoria de ciclo, que seja encontrada. Será necessário realizar essa função interativamente com o testador.

Document shared on www.docsity.com

14.9. Dê pelo menos três exemplos nos quais o teste caixa-preta pode dar a impressão de que "está tudo certo", enquanto testes caixa-branca poderiam detectar um erro. Dê pelo menos três exemplos nos quais o teste caixa-branca poderia dar a impressão de que "está tudo certo", enquanto testes caixa-preta poderiam descobrir um erro.

14.10. O teste exaustivo (mesmo que seja possível para programas muito pequenos) vai garantir que o programa esteja 100% correto?

14.11. Descreva com suas próprias palavras por que a classe é a menor unidade razoável de teste em um sistema OO.

14.12. Por que temos de testar novamente subclasses instanciadas a partir de uma classe existente, se a classe existente já foi testada rigorosamente? Podemos usar os casos de teste projetados para a classe existente?

14.13. Aplique o teste aleatório e o de partição a três classes definidas no projeto do sistema *CasaSegura*. Produza casos de teste que indiquem as seqüências de operação que serão invocadas.

14.14. Aplique teste de classes múltiplas e testes derivados do modelo comportamental ao projeto do *CasaSegura*.

14.15. Teste um manual de usuário (ou facilidades de ajuda) para uma aplicação que você use freqüentemente. Detecte pelo menos um erro na documentação.

Leituras e Fontes de Informação Adicionais

Entre dezenas de livros que apresentam métodos de projeto de casos de teste estão: Craig and Kaskiel *(Systematic Software Testing*, Artech House, 2002), Tamres *(Introducing Software Testing*, Addison-Wesley, 2002), Whittaker *(How to Break Software*, Addison-Wesley, 2002), Jorgensen *(Software Testing: A Craftman's Approach*, CRC Press, 2002), Splaine e seus colegas *(The Web Testing Handbook*, Software Quality Engineering Publishing, 2001), Patton *(Software Testing*, Sams Publishing, 2000), Kaner e seus colegas *(Testing Computer Software*, 2ª ed., Wiley, 1999). Além disso, Hutcheson *(Software Testing Methods and Metrics: The Most Important* Tests, McGraw-Hill, 1997) e Marick *(The Craft of Software Testing: Subsystem Testing Including Object-Based and Object-Oriented Testing*, Prentice-Hall, 1995) apresentam tratamentos de métodos e estratégias de teste.

Myers [MYE79] dá continuidade a um texto clássico, cobrindo técnicas de caixa-preta em considerável detalhe. Beizer [BEI90] fornece cobertura abrangente de técnicas caixa-branca, introduzindo um nível de rigor matemático que tem sido freqüentemente considerado carente em outros tratamentos de teste. Seu último livro [BEI95] apresenta uma abordagem concisa de importantes métodos. Perry *(Effective Methods for Software Testing*, Wiley-QED, 1995) e Friedman e Voas *(Software Assessment: Reliability, Safety, Testability*, Wiley, 1995) oferecem uma boa introdução às estratégias e táticas de teste. Mosley *(The Handbook of MIS Application Software Testing*, Prentice-Hall, 1993) discute tópicos de teste para grandes sistemas de informação e Marks *(Testing Very Big Systems*, McGraw-Hill, 1992) aborda os tópicos especiais que devem ser considerados em testes dos principais sistemas de programação.

Sykes e McGregor *(Practical Guide for Testing Object-Oriented Software*, Addison-Wesley, 2001), Bashir e Goel *(Testing Object-Oriented Software*, Springer-Verlag, 2000), Binder *(Testing Object-Oriented Systems*, Addison-Wesley, 1999), Kung e seus colegas *(Testing Object-Oriented Software*, IEEE Computer Society Press, 1998), Marick *(The Craft of Software Testing*, Prentice-Hall, 1997) e Siegel e Muller *(Object-Oriented Software Testing: A Hierarchical Approach*, Wiley, 1996) apresentam estratégias e métodos para teste de sistemas OO.

Teste de software é uma atividade intensa em termos de recurso. É por essa razão que muitas organizações automatizam parte do processo de teste. Os livros de Dustin, Rashka e Poston *(Automated Software Testing: Introduction, Management, and Performance*, Addison-Wesley, 1999), Graham e seus colegas *(Software Test Automation*, Addison-Wesley, 1999) e Poston *(Automating Specification-Based Software Testing*, IEEE Computer Society, 1996) discutem ferramentas, estratégias e métodos para teste automatizado.

Um certo número de obras considera métodos e estratégias de teste em áreas especializadas. Gardiner *(Testing Safety-Related Software: A Practical Handbook*, Springer-Verlag, 1999) editou um livro que trata do teste em sistemas de segurança crítica. Mosley *(Client/Server Software Testing on the Desk Top and the Web*, Prentice-Hall, 1999) discute o processo de teste para componentes clientes, servidores e de rede. Rubin *(Handbook of Usability Testing*, Wiley, 1994) escreveu um guia útil para aqueles que precisam exercitar interfaces humanas.

Binder [BIN99] descreve quase 70 padrões de teste que cobrem teste de métodos, classes, agregados, subsistemas, componentes reusáveis, frameworks e sistemas, bem como automação de teste e teste especializado em banco de dados. Uma lista desses padrões pode ser encontrada em *www.rbsc.com/pages/TestPatternList.htm*.

Uma ampla variedade de fontes de informação sobre métodos de projeto de casos de teste de software está disponível na Internet. Uma lista atualizada de referências da World Wide Web relevantes para técnicas de teste pode ser encontrada no site deste livro: **http://www.mhhe.com/pressman.**

CAPÍTULO

15 MÉTRICAS DE PRODUTOS PARA SOFTWARE

CONCEITOS-CHAVE

fatores de McCall 349
indicadores 352
medições 353
atributos 355
princípios 353
medidas 352
métricas
de modelo de análise .. 357
de código 370
de modelo de projeto .. 360
de manutenção 372
orientadas a objetos ... 363
de teste 371
paradigma GQM 354
pontos por função 357
qualidade 349

Um elemento-chave de qualquer processo de engenharia é a medição. Usamos medidas para entender melhor os atributos dos modelos que criamos e para avaliar a qualidade dos produtos ou sistemas submetidos à engenharia que construímos. Mas diferentemente de outras disciplinas de engenharia, a engenharia de software não está ancorada nas leis quantitativas básicas da física. Medidas diretas, tais como de massa, voltagem, velocidade ou temperatura são incomuns no mundo do software. Como medidas e métricas de software são freqüentemente indiretas, elas estão abertas a debate. Fenton [FEN91] trata desse assunto quando declara:

> Medição é o processo pelo qual números ou símbolos são associados aos atributos de entidades do mundo real de modo que os determinem de acordo com regras claramente definidas... Nas ciências físicas, medicina, economia e mais recentemente nas ciências sociais, estamos aptos agora a medir atributos que anteriormente pensávamos ser não-mensuráveis... Obviamente, tais medições não são tão refinadas quanto muitas medições nas ciências físicas..., mas elas existem (e decisões importantes são tomadas com base nelas). Achamos que a obrigação de tentar "medir o não-mensurável" a fim de melhorar nosso entendimento de entidades particulares é tão potente em engenharia de software como em qualquer disciplina.

Mas alguns membros da comunidade de software continuam a alegar que o software é "não-mensurável" ou que tentativas de mensurá-lo deveriam ser adiadas até que entendamos melhor o software e os atributos que deveriam ser usados para descrevê-lo. Isso é um erro.

PANORAMA

O que é? Por natureza, a engenharia é uma disciplina quantitativa. Engenheiros usam números para ajudá-los a projetar e avaliar o produto a ser construído. Até recentemente, engenheiros de software tinham pouca diretriz quantitativa em seu trabalho, — mas isso está mudando. Métricas de produto ajudam os engenheiros de software a ganhar profundidade na visão que têm do projeto e da construção do software que constroem. Diferentemente das métricas de processo e projeto que se aplicam ao projeto (ou processo) como um todo, as métricas de produto focalizam atributos específicos de produtos de trabalho de engenharia de software e são coletadas à medida que tarefas técnicas (análise, projeto, codificação e teste) estão sendo conduzidas.

Quem faz? Engenheiros de software usam métricas de produto para ajudá-los a construir software de alta qualidade.

Por que é importante? Sempre haverá um elemento qualitativo na criação de software de computador. O problema é que a avaliação qualitativa pode não ser suficiente. Um engenheiro de software precisa de critérios objetivos para ajudá-lo a dirigir o projeto dos dados, arquitetura, interfaces e componentes. O testador precisa de diretrizes quantitativas que vão ajudá-lo na seleção de casos de teste e seus alvos. Métricas de produto fornecem uma base a partir da qual análise, projeto, codificação e teste podem ser conduzidos mais objetivamente e avaliados mais quantitativamente.

Quais são os passos? O primeiro passo no processo de medição é derivar as medidas e métricas de software que são adequadas para a representação do software que está sendo considerado. A seguir, os dados necessários para derivar as métricas formuladas são coletados. Uma vez calculadas, as métricas adequadas são analisadas com base em diretrizes preestabelecidas e em dados anteriores. Os resultados da análise são interpretados para ganhar profundidade na visão da qualidade do software e os resultados da interpretação levam à modificação dos modelos de análise e projeto, código-fonte ou casos de teste. Em algumas instâncias pode também levar à modificação do processo de software propriamente dito.

Qual é o produto do trabalho? Métricas de produto que são calculadas a partir de dados coletados nos modelos de análise e projeto, código-fonte e casos de teste.

Como tenho certeza de que fiz corretamente? Você deve estabelecer os objetivos da medição antes que a coleta de dados tenha início, definindo cada métrica de produto de modo não-ambíguo. Defina apenas algumas métricas e depois use-as para ganhar profundidade de visão quanto à qualidade de um produto de trabalho da engenharia de software.

Apesar de as métricas de produto para software de computador serem freqüentemente não absolutas, elas nos dão um modo sistemático de avaliar qualidade com base em um conjunto de regras claramente definidas. Elas também dão ao engenheiro de software entendimento imediato, ao invés de *a posteriori*. Isso permite ao engenheiro descobrir e corrigir problemas potenciais, antes que se transformem em defeitos catastróficos.

Neste capítulo consideramos as medidas que podem ser usadas para avaliar a qualidade do produto à medida que ele está sendo construído. Essas medidas de atributos internos do produto dão ao engenheiro de software uma indicação em tempo real da eficácia dos modelos de análise, projeto e código; da efetividade dos casos de teste e da qualidade global do software em construção.

15.1 QUALIDADE DE SOFTWARE

Mesmo os mais sofridos desenvolvedores de software concordam que software de alta qualidade é uma meta importante. Mas como definimos qualidade? Em sentido mais geral, qualidade de software é a *satisfação de requisitos funcionais e de desempenho explicitamente declarados, normas de desenvolvimento explicitamente documentadas e características implícitas que são esperadas em todo o software desenvolvido profissionalmente*.

Há pouca dúvida de que esta definição poderia ser modificada ou estendida e debatida interminavelmente. Para a finalidade deste livro, a definição serve para enfatizar três pontos importantes:

1. Requisitos de software são a fundação a partir da qual a qualidade é medida. Falta de conformidade com os requisitos é falta de qualidade.[1]
2. Normas especificadas definem um conjunto de critérios de desenvolvimento que guiam o modo pelo qual o software é construído. Se os critérios não são seguidos, quase sempre vai resultar em falta de qualidade.
3. Há um conjunto de requisitos implícitos que freqüentemente não são mencionados (por exemplo, o desejo de facilidade de uso). Se o software satisfaz seus requisitos explícitos, mas deixa de satisfazer os requisitos implícitos, a qualidade do software é suspeita.

Qualidade de software é uma mistura complexa de fatores que variam com cada aplicação diferente e com os clientes que as encomendam. Nas seções seguintes, identificamos os fatores de qualidade de software e descreveremos as atividades humanas necessárias para alcançá-los.

15.1.1 Fatores de Qualidade de McCall

Os fatores que afetam a qualidade do software podem ser categorizados em dois amplos grupos: (1) fatores que podem ser medidos diretamente (por exemplo, defeitos por ponto por função) e (2) fatores que podem ser medidos apenas indiretamente (por exemplo, usabilidade ou manutenibilidade). Em cada caso devem ocorrer medições. Devemos comparar o software (documentos, programas e dados) a algum valor e chegar a uma indicação da qualidade.

1 É importante notar que a qualidade se estende às características técnicas dos modelos de análise e projeto e à realização em código-fonte desses modelos. Modelos que exibem alta qualidade (no sentido técnico) levam a software que exibem alta qualidade, do ponto de vista do cliente.

Document shared on www.docsity.com

FIGURA 15.1

Fatores de qualidade de software de McCall

PONTO CHAVE

É interessante notar que os fatores de qualidade de McCall são tão válidos hoje quanto eram na década de 1970. Assim, é razoável afirmar que os fatores que afetam a qualidade do software não mudam com o tempo.

McCall, Richards e Walters [MCC77] propõem uma categorização útil de fatores que afetam a qualidade do software. Esses fatores de qualidade do software, mostrados na Figura 15.1, concentram-se nos três aspectos importantes de um produto de software: suas características operacionais, sua habilidade de passar por modificações e sua adaptabilidade a novos ambientes.

Com referência aos fatores mencionados na Figura 15.1, McCall e seus colegas dão as seguintes descrições:

Correção. Quanto um programa satisfaz sua especificação e preenche os objetivos da missão do cliente.

Confiabilidade. Quanto se pode esperar que um programa realize a função pretendida com a precisão exigida. [Deve-se notar que outras definições mais completas de confiabilidade foram propostas (ver Capítulo 26).]

Eficiência. Quantidade de recursos de computação e código necessários para um programa realizar sua função.

Integridade. Quanto do acesso ao software ou dados por pessoas não-autorizadas pode ser controlado.

Usabilidade. O esforço necessário para aprender, operar, preparar entradas e interpretar saídas de um programa.

Manutenibilidade. O esforço necessário para localizar e consertar um erro em um programa. (Esta é uma definição muito limitada.)

Flexibilidade. O esforço necessário para modificar um programa operacional.

Testabilidade. Esforço necessário para testar um programa, a fim de garantir que ele realize a função esperada.

Portabilidade. Esforço necessário para transferir o programa de um ambiente de hardware ou software para outro.

Reutilização. Quanto de um programa (ou partes dele) pode ser reusado em outras aplicações — relativo ao empacotamento e escopo das funções que o programa realiza.

Interoperabilidade. Esforço necessário para acoplar um sistema a outro.

> "A qualidade de um produto é função de quanto ele muda o mundo para melhor."
>
> Tom DeMarco

É difícil, e em alguns casos impossível, desenvolver medidas[2] diretas desses fatores de qualidade. De fato, muitas das métricas definidas por McCall *et al.* podem ser medidas apenas subjetivamente.

2 Uma medida direta implica que exista um único valor contável que forneça uma indicação direta do atributo que está sendo examinado. Por exemplo, o "tamanho" de um programa pode ser medido diretamente pela contagem do número de linhas de código.

AVISO

Construa a sua própria lista de verificação usando esses fatores. Primeiro atribua a cada um a importância relativa para o seu projeto. Depois atribua graus aos seus produtos de trabalho para avaliar a qualidade do software que você está construindo.

As métricas podem estar em forma lista de verificação (checklist) que é usada para "atribuir grau" a atributos específicos do software [CAV78].

15.1.2 Fatores de Qualidade ISO 9126

A norma ISO 9126 foi desenvolvida em uma tentativa de identificar os atributos de qualidade para software de computador. A norma identifica seis atributos-chave de qualidade:

Funcionalidade. Grau em que o software satisfaz as necessidades declaradas, conforme indicado pelos seguintes subatributos: adequabilidade, precisão, interoperabilidade, atendibilidade e segurança.

Confiabilidade. Período de tempo em que o software está disponível para uso, conforme indicado pelos seguintes subatributos: maturidade, tolerância a falha, recuperabilidade.

Usabilidade. Grau em que o software é fácil de usar, conforme indicado pelos seguintes subatributos: inteligibilidade, facilidade de aprendizado, operabilidade.

Eficiência. Grau em que o software faz uso otimizado dos recursos do sistema, conforme indicado pelos seguintes subatributos: comportamento em relação ao tempo, comportamento em relação aos recursos.

Manutenibilidade. Facilidade com a qual podem ser feitos reparos no software, conforme indicado pelos seguintes subatributos: analisabilidade, mutabilidade, estabilidade, testabilidade.

Portabilidade. Facilidade com a qual o software pode ser transposto de um ambiente para outro, conforme indicado pelos seguintes subatributos: adaptabilidade, instalabilidade, conformidade, permutabilidade.

Como outros fatores de qualidade de software discutidos no Capítulo 9 e na Seção 15.1.1, os fatores ISO 9126 não necessariamente se prestam a medidas diretas. No entanto, fornecem uma base valiosa para medidas indiretas e uma excelente lista de verificação para avaliar a qualidade de um sistema.

> "Qualquer atividade torna-se criativa quando quem a executa se preocupa em fazê-la direito ou melhor."
>
> John Updike

15.1.3 Transição para Visão Quantitativa

Nas seções anteriores, um conjunto de fatores qualitativos para a "medição" da qualidade de software foi discutido. Lutamos para desenvolver medidas precisas da qualidade de software e ficamos algumas vezes frustrados pela natureza subjetiva da atividade. Cavano e McCall [CAV78] discutem essa situação:

> A determinação da qualidade é um fator-chave nos eventos diários — concursos de degustação de vinhos, eventos esportivos (por exemplo, ginástica), concursos de talento etc. Nessas situações, a qualidade é julgada da maneira mais fundamental e direta: comparação lado a lado de objetos sob condições idênticas e com conceitos predeterminados. O vinho pode ser julgado de acordo com a clareza, a cor, o odor, o sabor etc. No entanto, esse tipo de julgamento é muito subjetivo; para ter algum valor ele deve ser feito por um especialista.
>
> Subjetividade e especialização também se aplicam à determinação da qualidade de software. Para ajudar a resolver esse problema, é necessária uma definição de qualidade de software mais precisa, bem como um modo de derivar medições quantitativas da qualidade de software para análise objetiva...

Nas seções seguintes, examinamos um conjunto de métricas de software que pode ser aplicado à avaliação quantitativa da qualidade de software. Em todos os casos, as métricas representam medidas indiretas; isto é, nunca realmente medimos qualidade, mas em vez disso, alguma manifestação da qualidade. O fator complicativo é a relação precisa entre a variável que é medida e a qualidade do software.

Document shared on www.docsity.com

> "Assim como a medição da temperatura começou com o dedo indicador... e evoluiu para escalas, ferramentas e técnicas sofisticadas, a medição do software também está amadurecendo..."
>
> **Shari Pfleeger**

15.2 Um Arcabouço para Métricas de Produto

Como mencionamos na introdução a este capítulo, a medição atribui números ou símbolos aos atributos de entidades no mundo real. Para conseguir isso, é necessário um modelo de medidas, que inclui um conjunto de regras consistentes. Apesar de a teoria da medida (por exemplo, [KYB84]) e sua aplicação ao software de computador (por exemplo, [DEM81], [BRI96], [ZUS97]) serem tópicos que estão além do escopo deste livro, vale a pena estabelecer um arcabouço fundamental e um conjunto de princípios básicos para a medição de métricas de produto de software.

15.2.1 Medidas, Métricas e Indicadores

Apesar de os termos medida, medição e métrica serem freqüentemente usados intercambiadamente, é importante notar as diferenças sutis entre eles. A palavra *medida* pode ser usada tanto como substantivo quanto como verbo, tornando a definição do termo confusa. No contexto de engenharia de software, uma medida fornece uma indicação quantitativa da extensão, quantidade, dimensão, capacidade ou tamanho de algum atributo de um produto ou processo. *Medição* é o ato de determinar uma medida. O *IEEE Standard Glossary* [IEE93] define métrica como "uma medida quantitativa do grau em que um sistema, componente ou processo possui um determinado atributo".

Quando um único ponto de dados foi coletado (por exemplo, o número de erros descoberto em um único componente de software), uma medida foi estabelecida. Medição ocorre como resultado da coleção de um ou mais pontos de dados (por exemplo, um certo número de revisões de componente e testes de unidade são investigados para coletar medidas do número de erros em cada um). Uma métrica de software relaciona as medidas individuais de algum modo (por exemplo, o número médio de erros encontrados por revisão ou número médio de erros encontrados por teste de unidade).

Um engenheiro de software coleta medidas e desenvolve métricas de modo que indicadores sejam obtidos. Um *indicador* é uma métrica ou combinação de métricas que fornece profundidade na visão do processo de software, projeto de software ou produto em si. Um indicador fornece profundidade de visão que permite ao gerente do projeto ou engenheiros de software ajustar o processo, projeto ou produto para tornar as coisas melhores.

> "Uma ciência é tão madura quanto seus instrumentos de medição."
>
> **Louis Pasteur**

15.2.2 O Desafio das Métricas Técnicas

Durante as três últimas décadas, muitos pesquisadores tentaram desenvolver uma métrica única que fornecesse uma medida abrangente da complexidade do software. Fenton [FEN94] caracteriza essa pesquisa como a busca pelo "impossível santo Graal". Apesar de dezenas de medidas de complexidade terem sido propostas [ZUS90], cada uma adota um ponto de vista um tanto diferente sobre o que é complexidade e que atributos de um sistema levam à complexidade. Por analogia, considere uma métrica para avaliar um automóvel atrativo. Alguns observadores poderiam enfatizar o projeto da carroceria, outros poderiam considerar características mecânicas, outros poderiam ainda valorizar o custo, ou o desempenho, ou a economia de combustível, ou a capacidade de reciclar, quando o carro é descartado. Como qualquer uma dessas características pode se contrapor às outras, é difícil originar um único valor para "atração". O mesmo problema ocorre com o software de computador.

Veja na Web

Volumosa informação sobre métricas de produto foi compilada por Horst Zuse em irb.cs.tuberlin.de/~zuse/.

No entanto, há necessidade de medir e controlar a complexidade de software. E se um valor único para essa métrica de qualidade é difícil de derivar, deve ser possível desenvolver medidas de diferentes atributos internos de programa (por exemplo, modularidade efetiva, independência funcional e outros atributos discutidos nos Capítulos 9 a 12). Essas medidas e as métricas delas derivadas podem ser usadas como indicadores independentes da qualidade dos modelos de análise e projeto. Mas aí também surge um problema. Fenton [FEN94] menciona isso quando afirma: "O perigo de tentar encontrar medidas que caracterizam tantos atributos diferentes é que inevitavelmente as medidas têm de satisfazer objetivos conflitantes. Isso é contra a teoria representacional da medida". Apesar de a afirmação de Fenton estar correta, muitas pessoas alegam que a medição de produto conduzida durante os primeiros estágios do processo de software fornece aos engenheiros de software um mecanismo consistente e objetivo para avaliar a qualidade.

É justo perguntar, no entanto, quão válidas são as métricas de produto. Isto é, quão próximas as métricas de produto estão da confiabilidade e da qualidade a longo prazo de um sistema baseado em computador? Fenton [FEN91] trata dessa questão do seguinte modo:

> Apesar das conexões intuitivas entre a estrutura interna de produtos de software [métricas de produto] e seus atributos externos de produto e processo, têm havido realmente muito poucas tentativas científicas de estabelecer relações específicas. Há várias razões para isso; a mais comumente citada é a impraticabilidade da condução de experimentos relevantes.

Cada um dos "desafios" aqui mencionados é causa de precaução, mas não é razão para descartar as métricas de produto.[3] A medição é essencial quando se quer alcançar qualidade.

15.2.3 Princípios de Medição

? Quais são os passos de um processo de medição efetivo?

Antes de introduzirmos uma série de métricas de produto que (1) ajuda na avaliação dos modelos de análise e projeto, (2) fornece uma indicação da complexidade de projetos procedimentais e código-fonte e (3) facilita o projeto de teste mais efetivo, é importante entender princípios básicos de medição. Roche [ROC94] sugere um processo de medição que pode ser caracterizado por cinco atividades:

- *Formulação.* A derivação de medidas e métricas de software adequadas para a representação do software que está sendo considerado.
- *Coleta.* Mecanismo usado para acumular os dados necessários para derivar as métricas formuladas.
- *Análise.* Cálculo de métricas e aplicação das ferramentas matemáticas.
- *Interpretação.* Avaliação das métricas em um esforço para ganhar profundidade na visão da qualidade de representação.
- *Realimentação.* Recomendações derivadas da interpretação das métricas de produto transmitidas à equipe de software.

AVISO

Na realidade, muitas métricas de produto em uso hoje em dia não satisfazem esses princípios tão bem quanto deveriam. Mas, isso não significa que elas não tenham valor — apenas tome cuidado quando você as usa, entendendo que elas se destinam a fornecer profundidade de visão, não verificação científica precisa.

Métricas de software serão úteis apenas se forem caracterizadas efetivamente e validadas de modo que seu valor seja provado. Os seguintes princípios [LET03] são representativos de muitos que podem ser propostos para caracterização e validação de métricas:

- *Uma métrica deve ter propriedades matemáticas desejáveis.* Isto é, o valor da métrica deve estar em um intervalo significativo (por exemplo, zero a um, em que zero na realidade significa ausência, um indica o valor máximo e 0,5 representa o "ponto médio"). Também, uma métrica que se propõe a estar em uma escala racional não deve ser composta por componentes que são apenas medidos em uma escala ordinal.

3 Apesar de a crítica de métricas específicas ser comum na literatura, muitos críticos enfocam tópicos esotéricos e perdem o objetivo principal das métricas no mundo real: ajudar o engenheiro de software a estabelecer um modo sistemático e objetivo de ganhar profundidade na visão do seu trabalho e melhorar a qualidade do produto como resultado.

Document shared on www.docsity.com

- *Quando uma métrica representa uma característica de software que aumenta quando acontecimentos positivos ocorrem ou diminui quando acontecimentos indesejáveis são encontrados, o valor da métrica deve aumentar ou diminuir do mesmo modo.*
- *Cada métrica deve ser validada empiricamente em uma ampla variedade de contextos antes de ser publicada ou usada para tomar decisões.* Uma métrica deve medir o fator de interesse, independentemente de outros fatores. Ela deve se "adequar" a sistemas grandes e funcionar em uma variedade de linguagens de programação e domínios de sistemas.

Apesar de a formulação, caracterização e validação serem críticas, a coleta e a análise são as atividades que guiam o processo de medição. Roche [ROC94] sugere as seguintes diretrizes para essas atividades: (1) sempre que possível a coleta de dados e a análise devem ser automatizadas; (2) técnicas estatísticas válidas devem ser aplicadas para estabelecer relações entre os atributos internos do produto e as características externas de qualidade (por exemplo, se o nível de complexidade arquitetural está correlacionado com o número de defeitos relatados no uso em produção); (3) diretrizes e recomendações interpretativas devem ser estabelecidas para cada métrica.

15.2.4 Medições de Software Orientadas a Objetivo

Veja na Web

Uma discussão útil de GQM pode ser encontrada em www.thedocs.com/GoldPractices/practices/gqma.html.

O paradigma GQM (*Goal/Question/Metric* — Objetivo/Questão/Métrica) foi desenvolvido por Basili e Weiss [BAS84] como uma técnica para identificação de métricas significativas para qualquer parte do processo de software. GQM enfatiza a necessidade de (1) estabelecer um objetivo (*goal*) explícito de medição que seja específico da atividade do processo ou característico do produto que deve ser avaliado; (2) definir um conjunto de *questões* (*question*) que precisam ser respondidas a fim de alcançar o objetivo; e (3) identificar *métricas* (*metric*) bem formuladas que ajudam a responder a essas questões.

Um *gabarito de definição de objetivo* (*goal definition template*) [BAS84] pode ser usado para definir cada objetivo de medição. O gabarito tem a forma:

> **Analise** {o nome da atividade ou atributo a ser medido} **com a finalidade de** {o objetivo global da análise[4]} **com relação a** {o tópico da atividade ou atributo que é considerado} **do ponto de vista de** {o pessoal que tem interesse na medição} **no contexto de** {o ambiente no qual a medição tem lugar}.

Como exemplo, considere um gabarito de definição de objetivo para o *CasaSegura:*

> **Analise** a arquitetura do software do *CasaSegura* **com a finalidade de** avaliar componentes arquiteturais **com respeito à** habilidade de tornar o *CasaSegura* mais extensível **do ponto de vista dos** engenheiros de software que estão realizando o trabalho **no contexto de** aperfeiçoamento do produto durante os próximos três anos.

Com um objetivo de medição explicitamente definido, um conjunto de questões é desenvolvido. Respostas a essas questões ajudam a equipe de software (ou outros interessados) a determinar se o objetivo de medição foi atingido. Entre as questões que poderiam ser formuladas estão:

> Q_1: Os componentes arquiteturais são caracterizados de um modo que compartimentalize a função e dados relacionados?
> Q_2: É a complexidade de cada componente dentro dos limites que vai facilitar a modificação e extensão?

Cada uma dessas questões deve ser respondida quantitativamente, usando uma ou mais medidas e métricas. Por exemplo, uma métrica que fornece uma indicação da coesão (Capítulo 9) de um componente arquitetural poderia ser útil na resposta a Q_1. A complexidade ciclomática e métricas discutidas nas Seções 15.4.1 ou 15.4.2 poderiam aprofundar a visão de Q_2.

Na verdade, pode haver um certo número de metas de medição com as questões e métricas relacionadas. Em cada caso, as métricas escolhidas (ou derivadas) devem satisfazer aos princípios

4 Van Solingen e Berghout [SOL99] sugerem que o objetivo é quase sempre "entender, controlar ou aperfeiçoar" a atividade do processo ou atributo do produto.

de medição discutidos na Seção 15.2.3 e aos atributos de medição discutidos na Seção 15.2.5. Para mais informação sobre GQM, o leitor interessado deve ver [SHE98] ou [SOL99].

15.2.5 Os Atributos de Métricas de Software Efetivas

? Como devemos avaliar a qualidade de uma métrica de software proposta?

AVISO

A experiência indica que uma métrica de produto somente será usada se for intuitiva e fácil de calcular. Se dúzias de "contagens" tiverem de ser feitas e cálculos complexos forem necessários, é improvável que a métrica seja amplamente adotada.

Centenas de métricas têm sido propostas para software de computador, mas nem todas fornecem apoio prático ao engenheiro de software. Algumas exigem medições que são muito complexas, outras são tão restritas que poucos profissionais do mundo real têm qualquer esperança de entendê-las e outras violam as noções básicas intuitivas do que o software de alta qualidade realmente é.

Ejiogu [EJI91] define um conjunto de atributos que deve ser abrangido para métricas de software efetivas. As métricas derivadas e as medidas que levam a elas devem ser:

- *Simples e computáveis.* Deve ser relativamente fácil aprender como derivar a métrica e seu cálculo não deve exigir esforço ou tempo exagerado.
- *Empíricas e intuitivamente persuasivas.* A métrica deve satisfazer às noções intuitivas do engenheiro sobre o atributo do produto que está sendo considerado.
- *Consistentes e objetivas.* A métrica deve produzir sempre resultados que não sejam ambíguos.
- *Consistentes no uso de unidades e dimensões.* O cálculo matemático da métrica deve usar medidas que não levam a combinações de unidades bizarras.
- *Independentes da linguagem de programação.* Métricas devem ser baseadas no modelo de análise, modelo de projeto ou na estrutura do programa propriamente dita.
- *Mecanismo efetivo para realimentação de alta qualidade.* Isto é, a métrica deve levar a um produto final da mais alta qualidade.

Apesar de a maioria das métricas de software satisfazer a esses atributos, algumas métricas comumente usadas podem deixar de satisfazer a um ou dois entre eles. Um exemplo é pontos por função (discutido na Seção 15.3.1) — uma medida da "funcionalidade" entregue pelo software. Pode ser alegado[5] que o atributo *consistente e objetivo* falha, porque um terceiro independente pode não conseguir derivar o mesmo valor de pontos por função que um colega, usando a mesma informação sobre o software. Devemos assim rejeitar a medida FP (ponto por função)? A resposta é: "Sem dúvida que não!". FP oferece entendimento útil e assim fornece valor importante, mesmo se deixar de satisfazer perfeitamente a um atributo.

15.2.6 A Paisagem da Métrica de Produto

Apesar de uma ampla variedade de taxonomia de métricas ter sido proposta, a abordagem seguinte trata das mais importantes áreas de métricas:

Métricas para o modelo de análise. Essas métricas tratam de vários aspectos do modelo de análise e incluem:

Funcionalidade entregue — fornece uma medida indireta da funcionalidade que é empacotada com o software.

Tamanho do sistema — mede o tamanho global do sistema definido em termos de informação disponível como parte do modelo de análise.

Qualidade da especificação — fornece uma indicação da especificidade e completeza de uma especificação de requisitos.

Métricas para o modelo de projeto. Essas métricas quantificam atributos do projeto de um modo que permita ao engenheiro de software avaliar a qualidade do projeto. A métrica inclui:

Métrica arquitetural — fornece uma indicação da qualidade do projeto arquitetural.

5 Um contra-argumento igualmente vigoroso pode ser feito. Tal é a natureza das métricas de software.

Document shared on www.docsity.com

Métrica no nível de componente — mede a complexidade dos componentes de software e outras características que têm influência na qualidade.

Métricas de projeto de interface — focalizam principalmente a usabilidade.

Métricas especializadas em projeto OO — medem características de classes e suas características de comunicação e colaboração.

Métricas para código-fonte. Essas métricas medem o código-fonte e podem ser usadas para avaliar sua complexidade, manutenibilidade e testabilidade, entre outras características.

Métricas de Halstead — controversas, mas não obstantemente fascinantes, essas métricas fornecem medidas singulares de um programa de computador.

Métricas de complexidade — medem a complexidade lógica do código-fonte (podem também ser consideradas como métricas de projeto no nível de componente).

Métricas de comprimento — fornecem uma indicação do tamanho do software.

Métricas de teste. Essas métricas ajudam o projeto de casos de teste efetivos e avaliam a eficácia do teste.

Métricas de cobertura de comando e desvio — levam ao projeto de casos de teste que fornecem cobertura do programa.

Métricas relacionadas a defeito — enfocam erros encontrados, em vez dos testes em si.

Efetividade de teste — fornece uma indicação em tempo real da efetividade dos testes que foram conduzidos.

Métricas em processo — métricas relacionadas a processo que podem ser determinadas à medida que o teste é conduzido.

Em muitos casos, métricas para um modelo podem ser usadas em atividades posteriores de engenharia de software. Por exemplo, métricas de projeto podem ser usadas para estimar o esforço necessário para gerar código-fonte. Além disso, métricas de projeto podem ser usadas no planejamento de teste e no projeto de casos de teste.

CASASEGURA

Debate sobre Métricas de Produto

A cena: Cubículo de Vinod.

Os personagens: Vinod, Jamie e Ed — membros da equipe de engenharia de software do *CasaSegura*, que continuam trabalhando no projeto no nível de componente e no projeto de casos de teste.

A conversa:

Vinod: Doug [Doug Miller, gerente de engenharia de software] comentou que nós todos deveríamos usar métricas de produto, mas ele foi um pouco vago. Ele também disse que não iria forçar o uso... cabia a nós decidir o que fazer.

Jamie: Está certo, porque não há meio de eu ter tempo para começar a medir coisas. Já estamos lutando para cumprir o cronograma como as coisas estão.

Ed: Eu concordo com Jamie. Nós somos contra... não há tempo.

Vinod: É, eu sei, mas há provavelmente algum mérito em usá-las.

Jamie: Eu não estou discutindo isso, Vinod. É uma questão de tempo. . . e quanto a mim, eu não tenho nenhum tempo disponível.

Vinod: Mas, e se a medição poupar tempo?

Ed: Errado, ela toma tempo e como Jamie disse. . .

Vinod: Não, espere. . . e se isso nos poupar tempo?

Jamie: Como?

Vinod: Retrabalho . . . está aí como. Se uma métrica que nós usamos nos ajuda a evitar um problema importante ou mesmo moderado, e isso nos poupa ter que refazer uma parte do sistema, nós ganhamos tempo, não é?

Ed: É possível, eu acho, mas você pode garantir que alguma métrica de produto vai nos ajudar a encontrar um problema?

Vinod: Você pode garantir que não vai?

Jamie: Então o que você está propondo?

Vinod: Eu acho que deveríamos selecionar algumas métricas de projeto, provavelmente orientadas a classe, e usá-las como parte de nosso processo de revisão para cada componente que desenvolvemos.

Ed: Eu não estou realmente familiarizado com métricas orientadas a classe.

Vinod: Eu vou empregar algum tempo verificando-as e fazendo uma recomendação . . . está certo com vocês?

(Ed e Jamie concordam com a cabeça sem muito entusiasmo.)

15.3 MÉTRICAS PARA O MODELO DE ANÁLISE

Embora relativamente poucas métricas de análise e especificação tenham aparecido na literatura, é possível adaptar métricas que são freqüentemente usadas para estimativa de projetos e aplicá-las nesse contexto. Essas métricas examinam o modelo de análise com a intenção de prever o "tamanho" do sistema resultante. Tamanho é algumas vezes (mas, não sempre) um indicador da complexidade do projeto e é quase sempre um indicador do aumento de esforço de codificação, integração e teste.

15.3.1 Métricas Baseadas em Função

Veja na Web

Muita informação sobre ponto por função pode ser obtida em www.ifpug.org ou em www.functionpoints.com.

A *métrica ponto por função* (*Function Point* — FP), inicialmente proposta por Albrecht [ALB79], pode ser usada efetivamente como um meio para medir a funcionalidade entregue por um sistema.[6] Usando dados históricos, o FP pode então ser usado para (1) estimar o custo ou esforço necessário para projetar, codificar e testar o software; (2) prever o número de erros que vão ser encontrados durante o teste; e (3) prever o número de componentes e/ou o número de linhas de código projetadas no sistema implementado.

Pontos por função são derivados usando uma relação empírica baseada em medidas de contagem (direta) do domínio de informação do software e avaliação da complexidade do software. Valores do domínio de informação são definidos da seguinte maneira:[7]

Número de entradas externas (*External Inputs* — EIs). Cada *entrada externa* se origina de um usuário ou é transmitida de outra aplicação e fornece dados distintos orientados à aplicação do software ou informação de controle. Entradas são freqüentemente usadas para atualizar *arquivos lógicos internos (Internal Logical File — ILFs)*. Entradas devem ser distinguidas de consultas, que são contadas separadamente.

Número de saídas externas (*External Output* — EOs). Cada *saída externa* é derivada de dentro da aplicação e fornece informação para o usuário. Nesse contexto, saída externa refere-se a relatórios, telas, mensagens de erro etc. Itens de dados individuais dentro de um relatório não são contados separadamente.

Número de consultas externas (*External Inquiries* — EQs). Uma *consulta externa* é definida como uma entrada on-line, que resulta na geração de alguma resposta imediata do software sob a forma de uma saída on-line (freqüentemente recuperada de um ILF).

Número de arquivos lógicos internos (*Internal Logical Files* — ILFs). Cada *arquivo lógico interno* é um agrupamento lógico de dados que reside dentro das fronteiras da aplicação e é mantido por entradas externas.

Número de arquivos de interface externa (*External Interface Files* — EIFs). Cada *arquivo de interface externa* é um agrupamento lógico de dados que reside externamente à aplicação, mas fornece dados que podem ser úteis para a aplicação.

Uma vez coletados esses dados, a tabela da Figura 15.2 é completada e um valor de complexidade é associado com cada contagem. Organizações que usam os métodos de pontos por função desenvolvem critérios para determinar se uma entrada particular é simples, média ou complexa. Apesar disso, a determinação da complexidade é um tanto subjetiva.

Para calcular os pontos por função (*function points* — FP), a seguinte relação é usada:

$$FP = \text{total de contagem} \times [0{,}65 + 0{,}01 \times \Sigma\,(F_i)] \qquad (15\text{-}1)$$

em que o total da contagem é a soma de todas as entradas de FP obtidas da Figura 15.2.

6 Desde o trabalho original de Albrecht, centenas de livros, trabalhos e artigos têm sido escritos sobre FP. Uma bibliografia que vale a pena pode ser encontrada em [IFP03].

7 Na realidade, a definição dos valores do domínio da informação e o modo pelo qual eles são contados são um pouco mais complexos. O leitor interessado poderá ver [IFP01] para mais detalhes.

Document shared on www.docsity.com

FIGURA 15.2

Cálculo de pontos por função

Valor do Domínio da Informação	Contagem		Fator de Ponderação: Simples	Médio	Complexo		
Entradas Externas (EIs)		×	3	4	6	=	
Saídas Externas (EOs)		×	4	5	7	=	
Consultas Externas (EQs)		×	3	4	6	=	
Arquivos Lógicos Internos (ILFs)		×	7	10	15	=	
Arquivos de Interface Externa (EIFs)		×	5	7	10	=	
Contagem total							

Os F_i (i = 1 a 14) são fatores de ajuste de valor (*Value Adjustmente Factors* — VAF), baseados nas respostas às seguintes perguntas [LON02]:

PONTO CHAVE

Fatores de ajuste de valor são usados para fornecer uma indicação da complexidade do problema.

1. O sistema requer salvamento (backup) e recuperação (recovery)?
2. Comunicações de dados especializadas são necessárias para transferir informação para ou da aplicação?
3. Existem funções de processamento distribuído?
4. O desempenho é crítico?
5. O sistema será executado em um ambiente operacional existente, intensamente utilizado?
6. O sistema requer entrada de dados on-line?
7. A entrada de dados on-line exige que a transação de entrada seja construída por meio de várias telas ou operações?
8. Os ILFs são atualizados on-line?
9. As entradas, saídas, arquivos ou consultas são complexos?
10. O processamento interno é complexo?
11. O código é projetado para ser reusado?
12. A conversão e a instalação estão incluídas no projeto?
13. O sistema está projetado para instalações múltiplas em diferentes organizações?
14. A aplicação está projetada para facilitar modificações o uso do usuário?

Cada uma dessas questões é respondida usando uma escala que varia entre 0 (não-importante ou aplicável) e 5 (absolutamente essencial). Os valores constantes na Equação (15-1) e os fatores de ponderação, aplicados às contagens do domínio de informação, são determinados experimentalmente.

Para ilustrar o uso da métrica FP nesse contexto, consideramos uma representação simples do modelo de análise, ilustrada na Figura 15.3. Observando a figura, um diagrama de fluxo de dados (Capítulo 8) para uma função do software *CasaSegura* é representado. A função gerencia a interação com o usuário, aceitando uma senha do usuário para ativar ou desativar o sistema e permite consultas sobre o estado de zonas de segurança e de vários sensores de segurança. A função mostra uma série de mensagens de informação e envia sinais de controle apropriados aos vários componentes do sistema de segurança.

Veja na Web

Uma calculadora FP on-line pode ser encontrada em irb.cs.unimagdeburg.de/sw-eng/us/java/fp/.

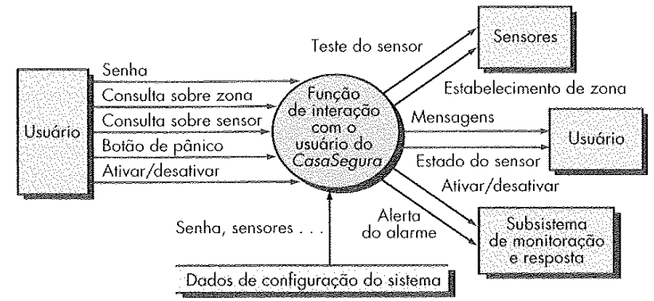
O diagrama de fluxo de dados é avaliado para determinar as medidas-chave exigidas para cálculo da métrica de pontos por função. Três entradas externas — **senha**, **botão de pânico** e **ativar/desativar** — são mostradas na figura junto com duas consultas externas — **consulta de zona** e **consulta de sensor**. Um arquivo (**arquivo de configuração do sistema**) é mostrado. Duas saídas externas (**mensagens** e **estado do sensor**) e quatro EIFs (**teste do sensor**, **estabelecimento de zona**, **ativar/desativar** e **alerta do alarme**) também estão presentes. Esses dados, juntamente com a complexidade adequada, são mostrados na Figura 15.4.

A contagem total mostrada na Figura 15.4 deve ser ajustada usando a equação (15-1):

$$FP = \text{contagem total} \times [0{,}65 + 0{,}01 \times \Sigma\,(F_i)]$$

FIGURA 15.3

Um modelo de fluxo de dados para o software do *CasaSegura*



em que contagem total é a soma de todas as entradas de FP obtidas na Figura 15.4 e F_i (i = 1 a 14) são valores de ajuste de complexidade. Para a finalidade desse exemplo, consideramos que o somatório $\Sigma\,(F_i)$ é 46 (um produto moderadamente complexo). Assim,

$$FP = 50 \times [0{,}65 + (0{,}01 \times 46)] = 56$$

Com base no valor projetado de FP, derivado do modelo de análise, a equipe de projeto pode estimar o tamanho total implementado da função de interação com o usuário do *CasaSegura*. Considere que dados anteriores indicam que um FP traduz-se em 60 linhas de código (uma linguagem orientada a objetos deve ser usada) e que 12 FPs são produzidos para cada pessoa-mês de esforço. Esses dados históricos fornecem ao gerente do projeto informação de planejamento importante, que é baseada no modelo de análise em vez de nas estimativas preliminares. Considere também que projetos anteriores encontraram em média três erros por ponto por função durante as revisões de análise e projeto e quatro erros por ponto por função durante o teste de unidade e de integração. Esses dados podem ajudar os engenheiros de software a avaliar a completeza de suas atividades de revisão e teste.

Uemura e seus colegas [UEM99] sugerem que pontos por função podem também ser calculados a partir dos diagramas UML de classe e seqüência (Capítulos 8 e 10). O leitor interessado pode ver [UEM99] para detalhes.

> "Em vez de só ficar pensando sobre que 'métrica nova' deve se aplicar... deveríamos estar nos perguntando a questão mais básica, 'o que vamos fazer com métricas'?"
>
> **Michael Mah e Larry Putnam**

15.3.2 Métricas de Qualidade de Especificação

Davis e seus colegas [DAV93] propõem uma lista de características que podem ser usadas para avaliar a qualidade do modelo de análise e da correspondente especificação de requisitos: *especi-*

FIGURA 15.4

Cálculo de pontos por função

Valor do Domínio da Informação	Contagem		Fator de Ponderação: Simples	Médio	Complexo		
Entradas Externas (EIs)	3	×	3	4	6	=	9
Saídas Externas (EOs)	2	×	4	5	7	=	8
Consultas Externas (EQs)	2	×	3	4	6	=	6
Arquivos Lógicos Internos (ILFs)	1	×	7	10	15	=	7
Arquivos de Interface Externa (EIFs)	4	×	5	7	10	=	20
Contagem total							50

Document shared on www.docsity.com

ficidade (falta de ambigüidade), *completeza, correção, inteligibilidade, verificabilidade, consistência interna e externa, atingibilidade, concisão, rastreabilidade, permutabilidade, precisão* e *reusabilidade*. Além disso, os autores [DAV93] indicam que especificações de alta qualidade são eletronicamente armazenadas, executáveis, ou pelo menos interpretáveis, anotadas quanto à importância relativa, estáveis, com versões definidas, organizadas, com referências cruzadas e especificadas no nível de detalhe correto.

PONTO CHAVE

Pela medição de características da especificação, é possível obter conhecimento aprofundado de especificidade e completeza.

Apesar de muitas dessas características parecerem ser de natureza qualitativa, Davis *et al.* [DAV93] sugerem que cada uma pode ser representada usando uma ou mais métricas. Por exemplo, assumimos que há n_r requisitos em uma especificação, tais que

$$n_r = n_f + n_{nf}$$

em que n_f é o número de requisitos funcionais e n_{nf} é o número de requisitos não-funcionais (por exemplo, desempenho).

Para determinar a *especificidade* (falta de ambigüidade) dos requisitos, Davis *et al.* sugerem uma métrica que é baseada na consistência da interpretação dos revisores de cada requisito:

$$Q_1 = n_{ui} / n_r$$

em que n_{ui} é o número de requisitos para os quais todos os revisores têm interpretações idênticas. Quanto mais perto de 1 for o valor de Q, mais baixa a ambigüidade da especificação.

A *completeza* dos requisitos funcionais pode ser determinada calculando a razão

$$Q_2 = n_u / [n_i \times n_s]$$

em que n_u é o número de requisitos funcionais únicos, n_i é o número de entradas (estímulos) definido ou implícito na especificação, e n_s é o número de estados especificado. A razão Q_2 mede a porcentagem de funções necessárias que foram especificadas para o sistema. No entanto, não trata de requisitos não-funcionais. Para incorporar esses últimos em uma métrica global para completeza, precisamos considerar o grau em que os requisitos foram validados:

$$Q_3 = n_c / [n_c + n_{nv}]$$

em que n_c é o número de requisitos que já foram validados como corretos e n_{nv} é o número de requisitos que ainda não foram validados.

> "Meça o que é mensurável e o que não é mensurável torne mensurável."
>
> Galileo

15.4 Métricas para o Modelo de Projeto

É inconcebível que o projeto de um novo avião, de um novo chip de computador, ou de um novo edifício de escritórios fosse conduzido sem definir medidas de projeto, determinando métricas para os vários aspectos da qualidade do projeto e usando-as para guiar a forma pela qual o projeto evolui. E, no entanto, o projeto de sistemas complexos baseados em software freqüentemente prossegue sem virtualmente nenhuma medição. A ironia disso é que métricas de projeto de software estão disponíveis, mas a grande maioria dos engenheiros de software continua a ignorar sua existência.

Métricas de projeto para software de computador, como todas as outras métricas de software, não são perfeitas. O debate sobre sua eficácia e sobre a forma pela qual elas deveriam ser aplicadas continua. Muitos especialistas argumentam que mais experimentação é necessária antes que as medidas de projeto possam ser usadas. E, no entanto, projetos sem medição são uma alternativa inaceitável.

Document shared on www.docsity.com

Nas seções seguintes, examinaremos algumas das métricas de projeto para software de computador mais comuns. Cada uma pode fornecer ao projetista visão mais aprofundada e todas podem ajudar o projeto a evoluir para o mais alto nível de qualidade.

15.4.1 Métricas de Projeto Arquitetural

Métricas de projeto arquitetural focalizam as características da arquitetura do programa (Capítulo 10) com ênfase na estrutura arquitetural e na efetividade dos módulos ou componentes dentro da arquitetura. Essas métricas são "caixa-preta", no sentido de não exigirem nenhum conhecimento do funcionamento interno de um componente de software particular.

Card e Glass [CAR90] definem três medidas de complexidade do projeto de software: complexidade estrutural, complexidade de dados e complexidade de sistema.

Para arquiteturas hierárquicas (por exemplo, arquiteturas de chamada e retorno), a *complexidade estrutural* de um módulo *i* é definida da seguinte maneira:

$$S(i) = f^2_{out}(i) \qquad (15\text{-}2)$$

em que $f_{out}(i)$ é o fan-out[8] do módulo *i*.

PONTO CHAVE

Métricas podem dar entendimento da complexidade estrutural, dos dados e do sistema associados com o projeto arquitetural.

Complexidade dos dados fornece uma indicação da complexidade na interface interna de um módulo *i* e é definida como:

$$D(i) = v(i) / [f_{out}(i) + 1] \qquad (15\text{-}3)$$

em que *v(i)* é o número de variáveis de entrada e saída que são passadas para e do módulo *i*.

Finalmente, *complexidade de sistema* é definida como a soma da complexidade estrutural e de dados, especificada como:

$$C(i) = S(i) + D(i) \qquad (15\text{-}4)$$

À medida que os valores dessas complexidades aumentam, a complexidade arquitetural global do sistema também aumenta. Isso leva a uma maior probabilidade de aumento do esforço de teste e de integração.

Fenton [FEN91] sugere um número de métricas simples de *morfologia* (por exemplo, forma) que permite que diferentes arquiteturas de programa sejam comparadas usando um conjunto de dimensões diretas. Observando a arquitetura de chamada e retorno da Figura 15.5, as seguintes métricas podem ser definidas:

$$\text{Tamanho} = n + a$$

em que *n* é o número de nós e *a* o número de arcos. Para a arquitetura mostrada na Figura 15.5,

tamanho = 17 + 18 = 35,
profundidade = 4, maior caminho entre o nó-raiz (de topo) para um nó-folha.
largura = 6, máximo número de nós em qualquer nível da arquitetura.
razão arco-nó, $r = a/n$,

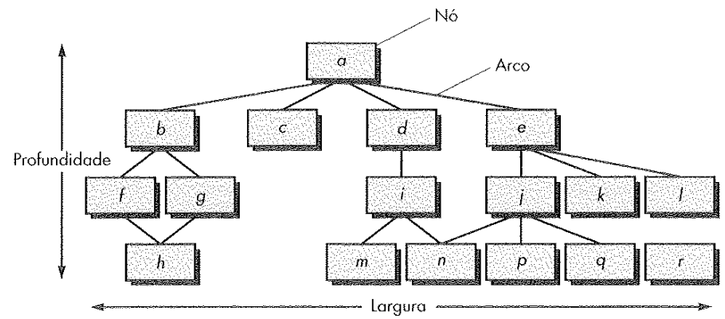
que mede a densidade de conectividade da arquitetura e pode fornecer uma indicação simples do acoplamento da arquitetura. Para a arquitetura mostrada na Figura 15.5, $r = 18/17 = 1{,}06$.

O Comando de Sistemas da Força Aérea Americana [USA87] desenvolveu alguns indicadores de qualidade de software, que são baseados em características de projeto mensuráveis de um programa de computador. Usando conceitos semelhantes àqueles propostos na norma IEEE Std.982.1-1988 [IEE94], a Força Aérea usa informação obtida do projeto de dados e arquitetural para derivar um índice de qualidade da estrutura do projeto (*design structure quality index* — DSQI) que vai de 0 a 1. Os seguintes valores devem ser estabelecidos para calcular o DSQI [CHA89]:

8 *Fan-out* é definido como o número de módulos imediatamente subordinados ao módulo *i*; isto é, o número de módulos que são diretamente invocados (acionados) pelo módulo *i*. *Fan-in* é definido como o número de módulos que invocam diretamente o módulo *i*.

FIGURA 15.5

Métricas morfológicas



S_1 = número total de módulos definidos na arquitetura do programa
S_2 = número de módulos cujo funcionamento correto depende da fonte de entrada de dados ou que produz dados a serem usados em outro lugar (de modo geral, módulos de controle, entre outros, não seriam contados como parte de S_2)
S_3 = número de módulos cujo funcionamento correto depende de processamento anterior
S_4 = número de itens na base de dados (inclui objetos de dados e todos os atributos que definem os objetos)
S_5 = número total de itens únicos na base de dados
S_6 = número de segmentos (registros ou objetos individuais diferentes) da base de dados
S_7 = número de módulos com uma única entrada e saída (processamento de exceção não é considerado saída múltipla)

Uma vez determinados os valores S_1 a S_7 para um programa de computador, os seguintes valores intermediários podem ser calculados:

Estrutura de programa: D_1, em que D_1 é definido como segue: se o projeto arquitetural foi desenvolvido usando um método distinto (por exemplo, projeto orientado a fluxo de dados ou projeto orientado a objetos) então $D_1 = 1$, senão $D_1 = 0$.

Independência modular: $D_2 = 1 - (S_2/S_1)$

Módulos não-dependentes de processamento anterior: $D_3 = 1 - (S_3/S_1)$

Tamanho da base de dados: $D_4 = 1 - (S_5/S_4)$

Compartimentalização da base de dados: $D_5 = 1 - (S_6/S_4)$

Característica de entrada/saída do módulo: $D_6 = 1 - (S_7/S_1)$

Com esses valores intermediários determinados, o DSQI é calculado do seguinte modo:

$$DSQI = \Sigma\, w_i D_i \qquad (15\text{-}5)$$

em que $i = 1$ a 6, w_i é o peso relativo da importância de cada um dos valores intermediários e $\Sigma w_i = 1$ (se todos os D_i têm peso igual, então $w_i = 0{,}167$).

O valor de DSQI de projetos anteriores pode ser determinado e comparado com o do projeto que está atualmente em desenvolvimento. Se o DSQI é significativamente menor do que a média, mais esforço de projeto e revisão é indicado. Analogamente, se grandes modificações precisam ser feitas em um projeto existente, o efeito dessas modificações no DSQI pode ser calculado.

> "A medição pode ser vista como um desvio. Esse desvio é necessário, porque os seres humanos, em geral, não conseguem tomar decisões claras e objetivas [sem apoio quantitativo]."
>
> **Horst Zuse**

15.4.2 Métricas para o Modelo de Projeto OO

Muito do que diz respeito ao projeto orientado a objetos é subjetivo — um projetista experiente "sabe" como caracterizar um sistema OO, de modo que implemente efetivamente os requisitos do cliente. Mas, à medida que o modelo de projeto cresce em tamanho e complexidade, uma visão mais objetiva das características do projeto pode beneficiar tanto o projetista experiente (que ganha experiência adicional) quanto o novato (que obtém uma indicação de qualidade que, de outro modo, não estaria disponível).

Em um tratamento detalhado das métricas de software para sistemas OO, Whitmire [WHI97] descreve nove características distintas e mensuráveis de um projeto OO:

? Que características podem ser medidas quando avaliamos um projeto OO?

Tamanho. O tamanho é definido em termos de quatro perspectivas: população, volume, comprimento e funcionalidade. *População* é medida pela contagem estática das entidades OO, tais como classes ou operações. Medidas de *volume* são idênticas a medidas de população, mas são coletadas dinamicamente — em um determinado instante de tempo. *Comprimento* é a medida de uma cadeia de elementos de projeto interconectados (por exemplo, a profundidade de uma árvore de herança é uma medida de comprimento). Métricas de *funcionalidade* fornecem uma indicação indireta do valor entregue ao cliente por uma aplicação OO.

Complexidade. Como o tamanho, há muitas perspectivas diferentes de complexidade de software [ZUS97]. Whitmire focaliza a complexidade em termos de características estruturais, examinando como as classes de um projeto OO estão inter-relacionadas umas com as outras.

Acoplamento. As conexões físicas entre elementos de um projeto OO (por exemplo, o número de colaborações entre classes ou o número de mensagens passadas entre objetos) representam o acoplamento dentro de um sistema OO.

Suficiência. Whitmire define suficiência como "o grau das características exigidas de uma abstração ou o grau das características que um componente de projeto possui na sua abstração, do ponto de vista da aplicação corrente". Dito de outro modo, perguntamos: Que propriedades essa abstração (classe) precisa possuir para me ser útil? [WHI97]. De fato, um componente de projeto (por exemplo, uma classe) é suficiente se reflete plenamente todas as propriedades do objeto do domínio de aplicação que está modelando — isto é, se a abstração (classe) possui as características dela requeridas.

> "Muitas das decisões, para as quais tive que confiar no folclore e no mito, podem ser tomadas agora usando dados quantitativos."
>
> **Scott Whitmire**

Completeza. A única diferença entre completeza e suficiência é "o conjunto de características com o qual comparamos a abstração ou o componente de projeto [WHI97]". A suficiência compara a abstração do ponto de vista da aplicação corrente. A completeza considera múltiplos pontos de vista, formulando a pergunta: Que propriedades são necessárias para representar plenamente o objeto do domínio do problema? Como o critério de completeza considera diferentes pontos de vista, tem implicação indireta no grau em que a abstração ou o componente de projeto pode ser reusado.

Coesão. Um componente OO deve ser projetado de modo que tenha todas as operações trabalhando juntas para atingir um propósito único, bem-definido, como seu correspondente no software convencional. A coesão de uma classe é determinada pelo exame do grau em que "o conjunto de propriedades que ela possui é parte do problema ou do domínio do projeto" [WHI97].

Primitividade. Característica semelhante a simplicidade, a primitividade (aplicada tanto a operações quanto a classes) é o grau em que uma operação é atômica — isto é, a operação não pode ser construída a partir de uma seqüência de outras operações contidas na classe. Uma classe que exibe um alto grau de primitividade encapsula apenas operações primitivas.

Similaridade. É o grau em que duas ou mais classes são semelhantes em termos de sua estrutura, função, comportamento ou finalidade.

Volatilidade. Como vimos anteriormente neste livro, podem ocorrer modificações de projeto quando os requisitos são alterados ou quando ocorrem modificações em outras partes de uma aplicação, que resultam na adaptação obrigatória do componente de projeto em questão. A volatilidade de um componente de projeto OO mede a probabilidade de que uma modificação venha a ocorrer.

Document shared on www.docsity.com

Na realidade, podem ser aplicadas métricas de produto para sistemas OO não apenas ao modelo de projeto, mas também ao modelo de análise. Nas próximas seções, exploramos métricas que fornecem uma indicação da qualidade no nível de classe OO e no nível de operação.

15.4.3 Métricas Orientadas a Classe — O Conjunto de Métricas CK

A classe é a unidade fundamental de um sistema OO. Conseqüentemente, medidas e métricas para uma classe individual, para a hierarquia de classes e para as colaborações entre classes serão de grande valor para um engenheiro de software que precisa avaliar a qualidade do projeto. Nos capítulos anteriores, vimos que a classe encapsula operações (processamento) e atributos (dados). A classe é freqüentemente "pai" de subclasses (algumas vezes chamadas de *filhas*) que herdam seus atributos e operações. A classe colabora freqüentemente com outras classes. Cada uma dessas características pode ser usada como base para medição.[9]

Um dos conjuntos de métricas de software OO mais amplamente referenciado foi proposto por Chidamber e Kemerer [CHI94]. Freqüentemente conhecido como *conjunto de métricas CK*, os autores propuseram seis métricas de projeto para sistemas OO baseados em classe.[10]

PONTO CHAVE

O número de métodos e sua complexidade estão diretamente correlacionados com o esforço necessário para testar uma classe.

Métodos ponderados por classe (*weighted methods per class* — WMC). Considere que n métodos de complexidade $c_1, c_2, \ldots, c_n$ são definidos para uma classe **C**. A métrica de complexidade específica escolhida (por exemplo, complexidade ciclomática) deve ser normalizada de modo que a complexidade nominal de um método assuma o valor de 1,0.

$$WMC = \Sigma c_i$$

para $i = 1$ a n. O número de métodos e sua complexidade são indicadores razoáveis da quantidade de esforço necessária para implementar e testar a classe. Além disso, quanto maior o número de métodos mais complexa é a árvore de herança (todas as classes subordinadas herdam os métodos de seus pais). Finalmente, à medida que o número de métodos para uma determinada classe cresce, é provável que ela se torne mais e mais específica da aplicação, limitando conseqüentemente o reuso em potencial. Por todas essas razões, o WMC deve ser mantido razoavelmente baixo.

Apesar de parecer relativamente direto desenvolver uma contagem do número de métodos de uma classe, o problema é, na realidade, mais complexo do que parece. Uma abordagem consistente de contagem para métodos deveria ser desenvolvida [CHU95].

AVISO

Herança é uma característica extremamente poderosa que pode colocá-lo em apuros se você a usar sem cuidado. Use DIT e outras métricas relacionadas para ter uma idéia da complexidade da hierarquia de classes.

Profundidade da árvore de herança (*depth of the inheritance tree* — DIT). Essa métrica é "o comprimento máximo do nó até a raiz da árvore" [CHI94]. Observando a Figura 15.6, o valor de DIT para a hierarquia de classes mostrada é 4. À medida que DIT cresce, é provável que as classes de nível mais baixo herdem muitos métodos. Isso leva a dificuldades em potencial quando se tenta prever o comportamento de uma classe. Uma hierarquia de classes profunda (DIT grande) leva também a maior complexidade de projeto. Do lado positivo, valores grandes de DIT implicam que muitos métodos podem ser reusados.

Número de filhos (*number of children* — NOC). As subclasses imediatamente subordinadas a uma classe, na hierarquia de classes, são denominadas *filhas*. Observando a Figura 15.6, a classe $\mathbf{C_2}$ tem três filhas — subclasses $\mathbf{C_{21}}$, $\mathbf{C_{22}}$ e $\mathbf{C_{23}}$. À medida que o número de filhas cresce, o reuso também cresce, mas, por outro lado, à medida que NOC aumenta, a abstração representada pela classe pai pode ser diluída. Isto é, alguns dos filhos podem não ser realmente membros adequados da classe pai. À medida que NOC aumenta, a quantidade de teste (necessária para exercitar cada filha no seu contexto operacional) também vai aumentar.

Acoplamento entre as classes de objetos (*coupling between object classes* — CBO). O modelo CRC (Capítulo 8) pode ser usado para determinar o valor de CBO. De fato, CBO é o número de colaborações listadas para uma classe no seu cartão de indexação CRC.[11] É provável que a reusabilidade de uma classe diminua à medida que o CBO aumenta. Valores altos de CBO também

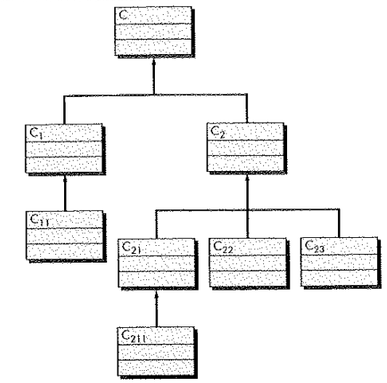
9 Deve-se ressaltar que a validade de algumas das métricas discutidas neste capítulo está atualmente em debate na literatura técnica. Alguns dos simpatizantes da teoria da medição exigem um grau de formalismo que algumas das métricas OO não fornecem. No entanto, é razoável declarar que todas as métricas mencionadas fornecem conhecimento útil para o engenheiro de software.

10 Chidamber e Kemerer usam o termo *métodos* em vez de *operações*. O uso desse termo está refletido nesta seção.

11 Se cartões indexados CRC são desenvolvidos manualmente, completeza e consistência devem ser avaliadas antes que CBO possa ser determinada com confiança.

FIGURA 15.6

Uma hierarquia de classes



complicam as modificações e os testes, que são necessários quando são feitas modificações. Em geral, os valores de CBO para cada classe devem ser mantidos razoavelmente baixos. Isso está consistente com a diretriz geral de reduzir o acoplamento no software convencional.

Resposta de uma classe (*response for a class* — RFC). O conjunto de respostas de uma classe é "um conjunto de métodos que podem ser executados potencialmente em resposta a uma mensagem recebida por um objeto daquela classe" [CHI94]. RFC é o número de métodos do conjunto-resposta. À medida que RFC aumenta, o esforço necessário para teste também aumenta, porque a seqüência de testes (Capítulo 14) cresce. E mais, à medida que RFC aumenta, a complexidade geral do projeto da classe aumenta.

AVISO

Os conceitos de acoplamento e coesão aplicam-se tanto ao software convencional quanto ao OO. Mantenha o acoplamento de classes baixo e a coesão de classes e operações alta.

Falta de coesão em métodos (*lack of cohesion in methods* — LCOM). Cada método de uma classe **C** tem acesso a um ou mais atributos (também chamados *instâncias de variáveis*). LCOM é o número de métodos que têm acesso a um ou mais dos mesmos atributos.[12] Se nenhum método tem acesso ao mesmo atributo, então LCOM = 0. Para ilustrar o caso em que LCOM ≠ 0, considere uma classe com seis métodos. Quatro dos métodos têm um ou mais atributos em comum (por exemplo, eles têm acesso a atributos comuns). Então, LCOM = 4. Se LCOM é alto, métodos podem estar acoplados uns aos outros por meio de atributos. Isso aumenta a complexidade do projeto da classe. Apesar de haver casos nos quais um alto valor de LCOM é justificável, manter a coesão alta, isto é, manter LCOM baixo, é desejável.[13]

CASASEGURA

Aplicação de Métricas CK

A cena: Cubículo de Vinod.

Os personagens: Vinod, Jamie, Shakira e Ed — membros da equipe de engenharia de software do *CasaSegura*, que continuam a trabalhar no projeto no nível de componente e no projeto de casos de teste.

A conversa:

Vinod: Vocês tiveram a oportunidade de ler a descrição do conjunto de métricas CK que eu lhes mandei na quarta-feira e fizeram aquelas medições?

Shakira: Não foi muito complicado. Eu voltei aos meus diagramas de classe e de seqüência UML como você

12 A definição formal é um tanto mais complexa. Veja [CHI94] para detalhes.

13 A métrica LCOM fornece uma visão útil mais aprofundada em algumas situações, mas pode ser enganosa em outras. Por exemplo, manter acoplamento encapsulado dentro de uma classe aumenta a coesão do sistema como um todo. Assim, no mínimo um sentimento importante, LCOM mais alto sugere que uma classe pode ter alta coesão, não mais baixa.

Document shared on www.docsity.com

sugeriu, e consegui contagens grosseiras de DIT, RFC e LCOM. Eu não consegui encontrar o modelo CRC, assim não contei CBO.

Jamie (rindo): Você não conseguiu encontrar o modelo CRC porque eu estava com ele.

Shakira: Isso é o que eu amo nesta equipe, comunicação perfeita.

Vinod: Eu fiz minhas contagens. . . vocês desenvolveram números para as métricas CK?

(Jamie e Ed acenam afirmativamente.)

Jamie: Como eu tinha os cartões CRC, dei uma olhada na CBO, e ela pareceu bastante uniforme ao longo da maioria das classes. Houve uma exceção que eu anotei.

Ed: Existem umas poucas classes em que RFC é bastante alto, comparado com as médias. . . talvez porque nós deveríamos tentar simplificá-las.

Jamie: Talvez sim, talvez não. Ainda estou preocupada com o tempo e não quero consertar coisa que não está, na realidade, quebrada.

Vinod: Eu concordo com isso. Talvez devêssemos procurar classes que tenham maus números em pelo menos duas ou mais métricas CK. Uma espécie de dois golpes e você é modificado.

Shakira (espiando a lista de classes de alto RFG de Ed): Olhe, vê essa classe? Ela tem um alto LCOM e um alto RFC. Dois golpes?

Vinod: É, eu acho que sim . . . vai ser difícil de implementar por causa da complexidade e difícil de testar pela mesma razão. Provavelmente valha a pena projetar duas classes separadas para conseguir o mesmo comportamento.

Jamie: Você acha que modificá-la nos poupa tempo?

Vinod: A longo prazo, sim.

15.4.4 Métricas Orientadas à Classe — o Conjunto de Métricas MOOD

Harrison, Counsell e Nithi [HAR98] propõem um conjunto de métricas para o projeto orientado a objetos que fornece indicadores quantitativos para as características do projeto OO. Uma pequena amostra das métricas MOOD é a seguinte:

Fator de herança de métodos (*method inheritance factor* — MIF). O grau em que a arquitetura de classes de um sistema OO faz uso de herança, tanto para métodos (operações) quanto atributos, é definido como

$$\text{MIF} = \Sigma\, M_i(C_i) / \Sigma\, M_a(C_i)$$

em que o somatório ocorre de $i = 1$ até T_c. T_c é definido como o número total de classes na arquitetura, C_i é uma classe dentro da arquitetura e

$$M_a(C_i) = M_d(C_i) + M_i(C_i)$$

em que

$M_a(C_i)$ = número de métodos que podem ser invocados em associação com C_i.
$M_d(C_i)$ = número de métodos declarados na classe C_i.
$M_a(C_i)$ = número de métodos herdados (e não-redefinidos) em C_i.

O valor de MIF — o fator de herança de atributos (*attribute inheritance factor* — AIF) é definido de modo análogo — fornece uma indicação do impacto da herança no software OO.

> "A análise de software OO, com o propósito de avaliar sua qualidade, está se tornando cada vez mais importante, à medida que o paradigma [OO] continua a aumentar em popularidade."
>
> **Rachel Harrison *et al.***

Fator de acoplamento (*coupling factor* — CF). Anteriormente, neste capítulo, mencionamos que o acoplamento é uma indicação das conexões entre elementos do projeto OO. O conjunto de métricas MOOD define o acoplamento da seguinte maneira:

$$\text{CF} = \Sigma_i \Sigma_j\, é_cliente\,(C_i, C_j) / (T_c^2 - T_c)$$

em que os somatórios ocorrem de $i = 1$ a T_c e $j = 1$ a T_c. A função

$$é_cliente = \begin{cases} 1, & \text{se e somente se existe uma relação entre a classe cliente } C_c \text{ e a classe servidora } C_s \text{ e } C_c \neq C_s \\ 0, & \text{caso contrário} \end{cases}$$

Apesar de muitos fatores afetarem a complexidade do software, sua inteligibilidade e sua manutenibilidade, é razoável concluir que à medida que o valor de CF aumenta, a complexidade do software OO também vai aumentar, e a inteligibilidade, a manutenibilidade e o potencial de reuso podem piorar como resultado.

Harrison e seus colegas [HAR98] apresentam uma análise detalhada de MIF e CF junto com outras métricas e examinam sua validade para uso na avaliação da qualidade do projeto.

15.4.5 Métricas Propostas por Lorenz e Kidd

Em seu livro sobre métricas OO, Lorenz e Kidd [LOR94] dividem métricas baseadas em classe em quatro amplas categorias, cada uma tendo um comportamento no projeto no nível de componente: tamanho, herança, aspectos internos e aspectos externos. Métricas orientadas a tamanho para uma classe de projeto OO focalizam a contagem de atributos e operações para uma classe individual e valores médios para todo o sistema OO. Métricas baseadas em herança focalizam um modo pelo qual as operações são reusadas ao longo da hierarquia de classes. Métricas para os tópicos internos da classe focalizam a coesão e tópicos orientados a código, e métricas externas examinam acoplamento e reuso. A seguir, um exemplo de métricas propostas por Lorenz e Kidd:

AVISO

Durante a revisão do modelo de análise, os cartões de indexação CRC fornecerão uma indicação razoável do valor esperado de CS. Se você encontrar uma classe com um número grande de responsabilidades, considere a possibilidade de particioná-la.

Tamanho da classe (*class size* — CS). O tamanho global de uma classe pode ser medido pela determinação das seguintes medidas:

- Número total de operações (tanto operações herdadas quanto de instância privada) que são encapsuladas na classe.
- Número de atributos (tanto herdados quanto atributos de instância privada) que são encapsulados na classe.

A métrica WMC proposta por Chidamber e Kemerer (Seção 15.4.3) também é uma medida ponderada do tamanho de classe. Como mencionamos anteriormente, valores grandes de CS indicam que uma classe pode ter responsabilidade demais. Isso reduzirá a reusabilidade da classe e complicará a implementação e o teste. Em geral, operações e atributos herdados ou públicos devem ser mais fortemente ponderados na determinação do tamanho da classe [LOR94]. Operações e atributos privados permitem especialização e são mais localizados no projeto. Também podem ser computadas médias para o número de atributos e operações de classe. Quanto mais baixa a média dos valores de CS, mais provavelmente as classes no sistema podem ser reusadas amplamente.

Número de operações adicionadas por uma subclasse (*number of operations added by a subclass* — NOA). Subclasses são especializadas pela adição de operações e atributos privados. À medida que o valor de NOA aumenta, a subclasse se afasta da abstração implicada pela superclasse. Em geral, à medida que a profundidade da hierarquia de classes aumenta (DIT torna-se grande), o valor de NOA nos níveis mais baixos da hierarquia deve diminuir.

15.4.6 Métricas de Projeto em Nível de Componente

Métricas de projeto em nível de componente para componentes de software convencional focalizam as características internas de um componente de software e incluem medidas dos "três C" — coesão, acoplamento e complexidade do módulo (*module cohesion, coupling and complexity*). Essas medidas podem ajudar o engenheiro de software a julgar a qualidade de um projeto em nível de componente.

As métricas apresentadas nesta seção são "caixa de vidro", no sentido de exigir conhecimento do funcionamento interno do módulo em consideração. Métricas de projeto em nível de componentes podem ser aplicadas, uma vez desenvolvido o processo procedimental. Alternativamente podem ser adiadas até que o código-fonte esteja disponível.

Document shared on www.docsity.com

Métricas de coesão. Bieman e Ott [BIE94] definem uma coleção de métricas que fornecem indicação da coesividade (Capítulo 9) de um módulo. As métricas são definidas especificando cinco conceitos e medidas:

PONTO CHAVE

É possível calcular medidas da independência funcional — acoplamento e coesão — de um componente e usá-las para avaliar a qualidade do projeto.

Fatia de dados (data slice). Na verdade, uma fatia de dados é um caminho retroativo ao longo de um módulo, que procura valores de dados que afetam a posição no módulo em que o caminho teve início. Deve-se notar que tanto fatias de programa (que se concentram em comandos e condições) quanto fatias de dados podem ser definidas.

Fichas de dados (data tokens). As variáveis especificadas para um módulo podem ser definidas como fichas de dados para o módulo.

Fichas aglutinantes (glue tokens). Esse conjunto de fichas de dados está situado em uma ou mais fatias de dados.

Fichas superaglutinantes (superglue tokens). Essas fichas de dados são comuns a todas as fatias de dados de um módulo.

Aglutinação (stickiness). A aglutinação relativa de uma ficha aglutinante é diretamente proporcional ao número de fatias de dados que ela aglutina.

Bieman e Ott desenvolveram métricas para *coesão funcional forte (strong functional cohesion — SFC), coesão funcional fraca (weak functional cohesion — WFC)* e *adesividade (adhesiveness)* (grau relativo em que fichas aglutinantes juntam fatias de dados). Essas métricas podem ser interpretadas do seguinte modo [BIE94]:

> Todas essas métricas de coesão variam de 0 a 1. Têm o valor 0 quando um procedimento tem mais de uma saída e não exibe nenhum dos atributos de coesão indicados por uma métrica particular. Em um procedimento sem fichas superaglutinantes, nenhuma ficha é comum a todas as fatias de dados, tem coesão funcional forte zero — não há fichas de dados que contribuem para todas as saídas. Um procedimento sem fichas aglutinantes, isto é, sem fichas comuns a mais que uma fatia de dados (em procedimentos com mais de uma fatia de dados), exibe coesão funcional fraca zero e adesividade zero — não há fichas de dados que contribuem para mais de uma saída.

Coesão funcional forte e adesividade são encontradas quando as métricas de Bieman e Ott assumem o valor máximo de 1.

Métricas de acoplamento. O acoplamento de módulos fornece a indicação da "conectividade" de um módulo a outros módulos, dados globais e ambiente exterior. No Capítulo 9, o acoplamento foi discutido em termos qualitativos.

Dhama [DHA95] propôs uma métrica para acoplamento de módulos que engloba acoplamento de dados e de fluxo de controle, acoplamento global e acoplamento ambiental. As medidas necessárias para calcular acoplamento de módulos são definidas em termos de cada um dos três tipos de acoplamento mencionados anteriormente. Para acoplamento de dados e de fluxo de controle,

d_i = número de parâmetros de dados de entrada
c_i = número de parâmetros de controle de entrada
d_o = número de parâmetros de dados de saída
c_o = número de parâmetros de controle de saída

Para acoplamento global,

g_d = número de variáveis globais usadas como dados
g_c = número de variáveis globais usadas como controle

Para acoplamento ambiental,

w = número de módulos chamados (*fan-out*)
r = número de módulos que chamam o módulo sendo considerado (*fan-in*)

Usando essas medidas, um indicador de acoplamento de módulo, m_c, é definido do seguinte modo:

$$m_c = k/M$$

em que k é uma constante de proporcionalidade e

$$M = d_i + (a \times c_i) + d_o + (b \times c_o) + g_d + (c \times g_c) + w + r \qquad (15\text{-}6)$$

Os valores para k, a, b e c devem ser derivados empiricamente.

À medida que o valor de m_c, aumenta, o acoplamento global do módulo diminui. A fim de ter a métrica de acoplamento aumentando conforme o grau de acoplamento aumenta, uma métrica de acoplamento revisada pode ser definida como

$$C = 1 - m_c$$

em que o grau de acoplamento aumenta à medida que as medidas da Equação (15-6) aumentam.

Métricas de complexidade. Diversas métricas de software podem ser calculadas para determinar a complexidade do fluxo de controle do programa. Muitas delas são baseadas no diagrama de fluxo. Como discutimos no Capítulo 14, um grafo é uma representação composta de nós e ligações (também chamadas arestas). Quando as ligações (arestas) são dirigidas, o grafo de fluxo é um grafo dirigido.

McCabe e Watson [MCC94] identificam alguns usos importantes para métricas de complexidade:

> Métricas de complexidade podem ser usadas para prever informação crítica sobre confiabilidade e manutenibilidade de sistemas de software a partir da análise automática do código-fonte [ou informação do projeto procedimental]. Métricas de complexidade também fornecem realimentação durante o projeto de software para ajudar a controlar a [atividade de projeto]. Durante o teste e manutenção, elas fornecem informação detalhada sobre os módulos de software para ajudar a localizar áreas de instabilidade potencial.

PONTO CHAVE

Complexidade ciclomática é apenas uma de uma grande quantidade de métricas de complexidade.

A métrica de complexidade mais amplamente usada (e debatida) para software de computador é a complexidade ciclomática, originalmente desenvolvida por Thomas McCabe [MCC76], [MCC89] e discutida em detalhes no Capítulo 14.

Zuse ([ZUS90], [ZUS97]) apresenta uma discussão enciclopédica de não menos que 18 categorias diferentes de métricas de complexidade de software. O autor apresenta as definições básicas para métricas de cada categoria (por exemplo, há algumas variações da métrica de complexidade ciclomática) e depois analisa e critica cada uma. O trabalho de Zuse é o mais abrangente publicado até hoje.

15.4.7 Métricas Orientadas a Operação

Como a classe é a unidade dominante em sistemas OO, menos métricas têm sido propostas para operações que residem dentro de uma classe. Churcher e Shepperd [CHU95] discutem isso quando afirmam: "Resultados de estudos recentes indicam que métodos tendem a ser pequenos tanto em termos de número de comandos quanto em complexidade lógica [WIL93], sugerindo que a estrutura de conectividade de um sistema pode ser mais importante do que o conteúdo de módulos individuais". No entanto, alguns aprofundamentos de visão podem ser obtidos pelo exame da média de características para os métodos (operações). Três métricas simples propostas por Lorenz e Kidd [LOR94] são adequadas:

Tamanho médio de operação (average operation size — OS_{avg}). Embora linhas de código possam ser usadas como indicador para o tamanho da operação, a medida LOC sofre de um conjunto de problemas discutido no Capítulo 22. Por essa razão, o número de mensagens enviado pela operação fornece uma alternativa para o tamanho da operação. À medida que o número de mensagens enviadas por uma única operação cresce, é provável que responsabilidades não tenham sido bem alocadas dentro da classe.

Complexidade da operação (operation complexity — OC). A complexidade de uma operação pode ser calculada usando qualquer das métricas de complexidade propostas para software convencional [ZUS90]. Como operações devem ser limitadas a uma responsabilidade específica, o projetista deve lutar para manter OC tão baixa quanto possível.

Document shared on www.docsity.com

Número médio de parâmetros por operação (average number of parameters per operation – NP_{avg}). Quanto maior o número de parâmetros da operação, mais complexa é a colaboração entre objetos. Em geral, NP_{avg} deve ser mantido tão baixo quanto possível.

15.4.8 Métricas de Projeto de Interface

Apesar de haver literatura significativa sobre o projeto de interface homem/computador (ver Capítulo 12), pouca informação tem sido relativamente publicada sobre métricas, que forneceriam uma visão aprofundada da qualidade e utilização da interface.

Sears [SEA93] sugere que a *adequação de leiaute* (*layout appropriateness*, LA) é uma métrica de projeto que vale a pena para interfaces humano/computador. Uma IGU típica usa entidades de leiaute — ícones gráficos, textos, menus, janelas e análogos — para ajudar o usuário a completar tarefas. Para realizar uma dada tarefa usando uma IGU, o usuário deve se deslocar de uma entidade de leiaute para a próxima. A posição absoluta e relativa de cada entidade de leiaute, a freqüência com que cada uma é usada e o "custo" da transição de uma entidade de leiaute para a próxima, tudo contribui para a adequação da interface.

> "Você pode aprender pelo menos um princípio de projeto de interface com o usuário carregando uma lavadora de pratos: se você encher demais, nada fica muito limpo."
>
> **Autor desconhecido**

AVISO

Métricas de projeto de interface são boas, mas, acima de tudo, fique absolutamente certo de que seus usuários finais gostam da interface e se sentem confortáveis com as interações necessárias.

Kokol e seus colegas [KOK95] definem uma métrica de coesão para telas de IU que medem a conexão relativa do conteúdo de uma tela com o conteúdo de outra tela. Se os dados (ou outro conteúdo) apresentados em uma tela pertencem a um único objeto principal (como definido no modelo de análise), a coesão da IU para aquela tela é alta. Se muitos tipos diferentes de dados ou conteúdos são apresentados e esses dados estão relacionados a diferentes objetos de dados, a coesão da IU é baixa. Os autores fornecem modelos empíricos para coesão [KOK95].

Além disso, medidas diretas de interação de IU podem enfocar medição do tempo necessário para alcançar um cenário ou uma operação específicos, tempo necessário para recuperar-se de uma condição de erro, contagem de operações específicas ou tarefas necessárias para satisfazer um caso de uso, o número de objetos de dados ou conteúdo apresentados na tela, densidade e tamanho do texto e muitos outros. No entanto, essas medidas diretas devem ser organizadas para criar métricas significativas de IU que vão levar à melhoria da qualidade da IU e/ou melhoria de usabilidade.

É importante notar que a seleção de um projeto de IGU pode ser guiada por métricas tais como LA ou coesão de telas de IU, mas o árbitro final deve ser a opinião do usuário, baseada em protótipos da IGU. Nielsen e Levy [NIE94] relatam que "há uma possibilidade de sucesso razoavelmente grande, se a escolha entre [projetos] de interface for feita com base apenas nas opiniões dos usuários. O desempenho médio dos usuários nas tarefas e sua satisfação subjetiva com uma IGU estão altamente correlacionados".

15.5 Métricas de Código-Fonte

A teoria de Halstead, da "ciência de software" [HAL77], propôs as primeiras "leis" analíticas para software de computador.[14] Halstead associou leis quantitativas ao desenvolvimento de software de computador, usando um conjunto de medidas primitivas que podem ser originadas após o código ser gerado, ou estimadas quando o projeto é completado. As medidas são:

n_1 = número de operadores distintos que aparece em um programa
n_2 = número de operandos distintos que aparece em um programa
N_1 = número total de ocorrências de operador
N_2 = número total de ocorrências de operando

14 Deve-se notar que as "leis" de Halstead geraram substancial controvérsia e muitos acreditam que a teoria subjacente tem falhas. No entanto, a verificação experimental para linguagens de programação selecionadas tem sido executada (por exemplo, [FEL89]).

AVISO

Operadores incluem todas as construções de fluxo de controle, condicionais e de operações matemáticas. Operandos abrangem todas as variáveis e constantes de programas.

Halstead usa essas medidas primitivas para desenvolver expressões para o tamanho total do programa, o volume potencial mínimo de um algoritmo, o volume real (número de bits necessário para especificar um programa), o nível do programa (medida da complexidade de software), o nível da linguagem (constante para uma certa linguagem) e outras características tais como esforço de desenvolvimento, tempo de desenvolvimento e mesmo o número projetado de falhas do software.

Halstead mostra que o tamanho N pode ser estimado

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

e o volume do programa pode ser definido

$$V = N \log_2 (n_1 + n_2)$$

Deve-se notar que V varia de acordo com a linguagem de programação e representa o volume de informação (em bits) necessário para especificar um programa.

> "O cérebro humano segue um conjunto de regras mais rígido [no desenvolvimento de algoritmos] do que se tem consciência."
>
> **Maurice Halstead**

Teoricamente deve existir um volume mínimo para um algoritmo particular. Halstead define uma razão de volume L como a razão do volume da forma mais compacta de um programa para o volume do programa real. Na realidade, L deve ser sempre menor que 1. Em termos de medidas primitivas, a razão de volume pode ser expressa como:

$$L = 2/n_1 \times n_2/N_2$$

O trabalho de Halstead é passível de verificação experimental e um grande volume de pesquisa tem sido conduzido para investigar a ciência de software. Para mais informações ver [ZUS90], [FEN91] e [ZUS97].

15.6 Métricas para Teste

Apesar de muito ter sido escrito sobre métricas de teste para software (por exemplo, [HET93]), a maioria das métricas propostas focaliza o processo de teste, não as características técnicas dos testes propriamente ditos. Em geral, os testadores devem se apoiar nas métricas de análise, projeto e código, para guiá-los no projeto e execução de casos de teste.

PONTO CHAVE

Métricas de teste enquadram-se em duas categorias amplas: (1) métricas que tentam prever o número de testes provável necessário nos vários níveis de teste e (2) métricas que focalizam a cobertura de teste para um dado componente.

Métricas baseadas em função (Seção 15.3.1) podem ser usadas como previsão para o esforço global de teste. Várias características de projeto (por exemplo, esforço e tempo de teste, erros descobertos, número de casos de teste produzidos) para projetos anteriores, podem ser coletadas e correlacionadas com o número de pontos por função produzidos por uma equipe de projeto. A equipe pode então projetar "valores esperados" dessas características para o projeto em andamento.

As métricas de projeto arquitetural fornecem informação sobre a facilidade ou dificuldade associada ao teste de integração (Capítulo 13) e a necessidade de teste de software especializado (por exemplo, pseudocontroladores e pseudocontrolados). A complexidade ciclomática (uma métrica de projeto em nível de componente) fica situada no âmago do teste de caminho básico, o método de projeto de casos de teste apresentado no Capítulo 14. Além disso, a complexidade ciclomática pode ser usada para escolher módulos candidatos a testes de unidade extensivos. Módulos com alta complexidade ciclomática são mais sujeitos a erros que módulos cuja complexidade ciclomática é mais baixa. Por esse motivo, o testador deve empregar mais do que o esforço médio para descobrir erros em tais módulos antes de os integrar.

15.6.1 Métricas de Halstead Aplicadas ao Teste

O esforço de teste também pode ser estimado usando métricas derivadas das medidas de Halstead (Seção 15.5). Usando as definições de volume de programa, V, e nível de programa PL (*program level*), o esforço de Halstead, e, pode ser calculado como

Document shared on www.docsity.com

$$PL = 1/[(n_1/2) \times (N_2/n_2)] \quad (15\text{-}7a)$$

$$e = V/PL \quad (15\text{-}7b)$$

A porcentagem do esforço total de teste a ser alocada a um módulo k pode ser estimada usando a seguinte relação:

$$\text{porcentagem do esforço de teste } (k) = e(k)/\Sigma\, e(i) \quad (15\text{-}8)$$

em que $e(k)$ é calculado para o módulo k usando as Equações (15-7) e o somatório no denominador da Equação (15-8) é a somatória do esforço da ciência de software ao longo de todos os módulos do sistema.

15.6.2 Métricas para Teste Orientado a Objetos

As métricas de projeto OO mencionadas na Seção 15.4 fornecem uma indicação da qualidade do projeto. Elas também dão uma indicação geral da quantidade de esforço de teste necessária para exercitar um sistema OO.

Binder [BIN94] sugere um amplo conjunto de métricas de projeto que tem influência direta na "testabilidade" de um sistema OO. As métricas consideram os tópicos de encapsulamento e herança. Segue-se uma amostra:

AVISO

O teste OO pode ser bastante complexo. Métricas podem ajudá-lo a concentrar os recursos de teste em linhas de ação, cenários e aglomerados de classe que são "suspeitos", com base nas características medidas. Use-as.

Falta de coesão em métodos (*lack of cohesion in methods* — LCOM)[15]. Quanto maior o valor de LCOM, mais estados devem ser testados para garantir que os métodos não geram efeitos colaterais.

Porcentagem pública e protegida (*percent public and protected* — PAP). Essa métrica indica a porcentagem dos atributos da classe que são públicos ou protegidos. Valores altos de PAP aumentam a probabilidade de efeitos colaterais entre classes porque atributos públicos e protegidos levam a alto acoplamento potencial (Capítulo 9).[16] Devem ser projetados testes para garantir que tais efeitos colaterais sejam descobertos.

Acesso público a membros de dados (*public access to data members* — PAD). Essa métrica indica o número de classes (ou métodos) que podem acessar os atributos de outra classe, que é uma violação do encapsulamento. Valores altos de PAD podem levar a efeitos colaterais entre classes. Os testes precisam ser projetados para garantir que tais efeitos colaterais sejam descobertos.

Número de classes raiz (*number of root classes* — NOR). Essa métrica é uma contagem de hierarquias de classes distintas, que são descritas no modelo de projeto. Seqüências de teste para cada classe raiz e a correspondente hierarquia de classes precisam ser desenvolvidas. À medida que NOR aumenta, o esforço de teste também aumenta.

Convergência (*fan-in* — FIN). Quando usado no contexto OO, *fan-in* para a hierarquia de herança é uma indicação de herança múltipla. FIN > 1 indica que uma classe herda seus atributos e operações de mais de uma classe raiz. FIN > 1 deve ser evitado sempre que possível.

Número de filhos (*number of children* — NOC) e profundidade da árvore de herança (*depth of the inheritance tree* — DIT).[17] Como discutimos no Capítulo 14, os métodos da superclasse terão de ser retestados em cada subclasse.

15.7 Métricas de Manutenção

Todas as métricas de software introduzidas neste capítulo podem ser usadas para o desenvolvimento de software novo e para a manutenção de software existente. No entanto, métricas projetadas explicitamente para as atividades de manutenção têm sido propostas.

15 Veja a Seção 15.4.3 para uma descrição de LCOM.
16 Algumas pessoas criam projetos em que os atributos não são públicos nem privados; isto é, FAP = 0. Isso implica que todos os atributos devem ser acessados via método em outras classes.
17 Veja a Seção 15.4.3 para uma descrição de NOC e DIT.

A norma IEEE Std. 982.1-1988 [IEE94] sugere um *índice de maturidade de software* (*software maturity index* — SMI) que fornece indicação da estabilidade de um produto de software (com base nas modificações que ocorrem em cada versão do produto). A seguinte informação é determinada:

M_T = número de módulos na versão corrente

F_c = número de módulos na versão corrente que foram modificados

F_a = número de módulos na versão corrente que foram adicionados

F_d = número de módulos na versão anterior que foram descartados na versão corrente

O índice de maturidade de software é calculado da seguinte maneira:

$$SMI = [M_t - (F_a + F_c + F_d)]/M_T \quad (19\text{-}15)$$

À medida que SMI se aproxima de 1,0, o produto começa a se estabilizar. SMI também pode ser usado como métrica para o planejamento das atividades de manutenção de software. O tempo médio para produzir uma versão de um produto de software pode ser correlacionado com o SMI e modelos empíricos para o esforço de manutenção podem ser desenvolvidos.

FERRAMENTAS DE SOFTWARE

Métricas de Produto

Objetivo: Apoiar engenheiros de software no desenvolvimento de métricas significativas para avaliar os produtos de trabalho produzidos durante a modelagem de análise e projeto, geração de código-fonte e teste.

Mecânica: Ferramentas dessa categoria abrangem um amplo conjunto de métricas que são implementadas quer como aplicações isoladas ou (mais comumente) como funcionalidade que existe dentro de ferramentas para análise e projeto, codificação ou teste. Na maioria dos casos, a ferramenta de métricas analisa uma representação do software (por exemplo, um modelo UML ou código-fonte) e desenvolve uma ou mais métricas como resultado.

Ferramentas Representativas[18]

Krakatau Metrics, desenvolvida por Power Software (www.powersoftware.com/products), calcula métricas de complexidade, Halstead, outras relacionadas com C/C++ e Java.

Metrics4C, desenvolvida por +1 Software Engineering (www.plus-one.com/Metrics4C_fact_sheet.html), calcula uma variedade de métricas arquiteturais, de projeto e orientadas a código, bem como métricas orientadas a serviço.

Rational Rose, desenvolvida por Rational Corporation (www.rational.com), é um conjunto de ferramentas abrangente para modelagem UML que inclui um certo número de características de análise de métricas.

RSM, desenvolvida por M-Squared Technologies (msquaredtechnologies.com/m2rsm/index.html), calcula uma grande variedade de métricas orientadas a código C, C++ e Java.

Understand, desenvolvida por Scientific Toolworks, Inc. (www.scitools.com), calcula métricas orientadas para uma variedade de linguagens de programação.

15.8 Resumo

Métricas de software fornecem uma maneira quantitativa de avaliar a qualidade de atributos internos do produto, habilitando assim o engenheiro de software a avaliar a qualidade antes de o produto ser construído. As métricas fornecem a visão aprofundada necessária para criar modelos efetivos de análise e projeto, código sólido e testes rigorosos.

Para ser útil no contexto do mundo real, uma métrica de software precisa ser simples e calculável, persuasiva, consistente e objetiva. Deve ser independente da linguagem de programação e fornecer efetiva realimentação para o engenheiro de software.

18 As ferramentas mencionadas não representam uma recomendação, mas em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

Document shared on www.docsity.com

As métricas para o modelo de análise focalizam a função, os dados e o comportamento — as três componentes do modelo de análise. As métricas de projeto consideram tópicos da arquitetura, projeto no nível de componentes e projeto de interface. As métricas do projeto arquitetural consideram os tópicos estruturais do modelo de projeto. As métricas de projeto no nível de componentes fornecem indicação da qualidade do módulo, estabelecendo medidas indiretas de coesão, acoplamento e complexidade. As métricas de projeto de interface com o usuário fornecem indicação da facilidade com a qual uma IGU pode ser usada.

As métricas para sistemas OO focalizam medições que podem ser aplicadas à classe e às características do projeto — localização, encapsulamento, ocultamento da informação, herança e técnicas de abstração de objetos — que tornam a classe singular.

Halstead fornece um interessante conjunto de métricas no nível do código-fonte. Usando o número de operadores e operandos presentes no código, uma variedade de métricas é desenvolvida para avaliar a qualidade do programa.

Poucas métricas de produto têm sido propostas para uso direto no teste e manutenção de software. No entanto, muitas outras métricas de produto podem ser usadas para guiar o processo de teste e como mecanismo para avaliar a manutenibilidade de um programa de computador. Uma grande variedade de métricas OO tem sido proposta para avaliar a testabilidade de um sistema OO.

Referências Bibliográficas

[ALB79] Albrecht, A. J., "Measuring Application Development Productivity", *Proc. IBM Application Development Symposium*, Monterey, CA, out. de 1979, p. 83-92.

[ALB83] Albrecht, A. J. e Gaffney, J. E., "Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation", *IEEE Trans. Software Engineering*, nov. de 1983, p. 639-648.

[BAS84] Basili, V. R. e Weiss, D. M., "A Methodology for Collecting Valid Software Engineering Data", *IEEE Trans. Software Engineering*, vol. SE-10, 1984, p. 728-738.

[BER95] Berard, E., "Metrics for Object-Oriented Software Engineering", Colocado na Internet em comp.software-eng, 28-1-1995.

[BIE94] Bieman, J. M. e Ott, L. M., "Measuring Functional Cohesion", *IEEE Trans. Software Engineering*, vol. SE-20, n. 8, agos. de 1994, p. 308-320.

[BIN94] Binder, R. V., "Object-Oriented Software Testing", *CACM*, vol. 37, n. 9, set. de 1994, p. 29.

[BRI96] Briand, L. C., S. Morasca e Basili, V. R., "Property-Based Software Engineering Measurement", *IEEE Trans. Software Engineering*, vol. SE-22, n. 1, jan. de 1996, p. 68-85.

[CAR90] Card, D. N. e Glass, R. L., *Measuring Software Design Quality*, Prentice-Hall, 1990.

[CAV78] Cavano, J. P. e McCall, J. A., "A Framework for the Measurement of Software Quality", *Proc. ACM Software Quality Assurance Workshop*, nov. de 1978, p. 133-139.

[CHA89] Charette, R. N., *Software Engineering Risk Analysis and Management*, McGraw-Hill/ Intertext, 1989.

[CHI94] Chidamber, S. R. e Kemerer, C. F., "A Metrics Suite for Object-Oriented Design", *IEEE Trans. Software Engineering*, vol. SE-20, n. 6, jun. de 1994, p. 476-493.

[CHI98] Chidamber, S. R., Darcy, D. P., e C. F. Kemerer, "Management Use of Metrics for Object-Oriented Software: An Exploratory Analysis", *IEEE Trans. Software Engineering*, vol. SE-24, n. 8, agos. de 1998, p. 629-639.

[CHU95] Churcher, N. I. e Shepperd, M. J., "Towards a Conceptual Framework for Object-Oriented Metrics", *ACM Software Engineering Notes*, vol. 20, n. 2, abril de 1995, p. 69-76.

[CUR80] Curtis, W., "Management and Experimentation in Software Engineering", *Proc. IEEE*, vol. 68, n. 9, set. de 1980.

[DAV93] Davis, A. *et al.*, "Identifying and Measuring Quality in a Software Requirements Specification", *Proc. First Intl. Software Metrics Symposium*, IEEE, Baltimore, MD, maio de 1993, p. 141-152.

[DEM81] DeMillo, R. A. e Lipton, R. J., "Software Project Forecasting," in *Software Metrics* (A. J. Perlis, F. G. Sayward e M. Shaw, eds.), MIT Press, 1981, p. 77-89.

[DEM82] DeMarco, T., Controlling Software Projects, Yourdon Press, 1982.[DHA95] Dhama, H., "Quantitative Models of Cohesion and Coupling in Software", *Journal of Systems and Software*, vol. 29, n. 4, abril de 1995.

[EJI91] Ejiogu, L., *Software Engineering with Formal Metrics*, QED Publishing, 1991.

[FEL89] Felican, L. e Zalateu, G., "Validating Halstead's Theory for Pascal Programs", *IEEE Trans. Software Engineering*, vol. SE-15, n. 2, dez de 1989, p. 1630-1632.

[FEN91] Fenton, N., *Software Metrics*, Chapman and Hall, 1991.

[FEN94] Fenton, N., "Software Measurement: A Necessary Scientific Basis", *IEEE Trans. Software Engineering*, vol. SE-20, n. 3, março de 1994, p. 199-206.

[GRA87] Grady, R. B. e Caswell, D. L., *Software Metrics: Establishing a Company-Wide Program*, Prentice-Hall, 1987.

[HAL77] Halstead, M., *Elements of Software Science*, North-Holland, 1977.

[HAR98] Harrison, R., Counsell, S. J., e Nithi, R. V., "An Evaluation of the MOOD Set of Object-Oriented Software Metrics", *IEEE Trans. Software Engineering*, vol. SE-24, n. 6, jun. de 1998, p. 491-496.

[HET93] Hetzel, B., *Making Software Measurement Work*, QED Publishing, 1993.

[IEE93] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE, 1993.

[IEE94] *Software Engineering Standards*, 1994 edition, IEEE, 1994.

[IFP01] *Function Point Counting Practices Manual*, Release 4.1.1, International Function Point Users Group, 2001, disponível em http://www.ifpug.org/publications/ manual.htm.

[IFP03] Function Point Bibliography/Reference Library, International Function Point Users Group, 2003, disponível em http://www.ifpug.org/about/bibliography. htm.

[KOK95] Kokol, P., I. Rozman e Venuti, V., "User Interface Metrics", *ACM SIGPLAN Notices*, vol. 30, no. 4, abril 1995, pode ser obtido em: http://portal.acm.org/.

[KYB84] Kyburg, H. E., *Theory and Measurement*, Cambridge University Press, 1984.

[LET03] Lethbridge, T., Comunicação particular sobre métricas de software, jun. de 2003.

[LON02] Longstreet, D., "Fundamental of Function Point Analysis", Longstreet Consulting, Inc, 2002, disponível em http:// www.ifpug.com/fpafund.htm.

[LOR94] Lorenz, M. e Kidd, J., *Object-Oriented Software Metrics*, Prentice-Hall, 1994.

[MCC76] McCabe, T. J., "A Software Complexity Measure", *IEEE Trans. Software Engineering*, vol. SE-2, dez de 1976, pp. 308-320.

[MCC77] McCall, J., P. Richards e Walters, G., "Factors in Software Quality", três volumes, NTIS AD-A049-014, 015, 055, nov. de 1977.

[MCC89] McCabe, T. J., e Butler, C. W., "Design Complexity Measurement and Testing", CACM, vol. 32, n. 12, dez. 1989, p. 1415-1425.

[MCC94] McCabe, T. J. e Watson, A. H., "Software Complexity", *Crosstalk*, vol. 7, n. 12, dez. 1994, p. 5-9.

[NIE94] Nielsen, J. e Levy, J., "Measuring Usability: Preference vs. Performance", *CACM*, vol. 37, n. 4, abr. de 1994, p. 65-75.

[ROC94] Roche, J. M., "Software Metrics and Measurement Principles", *Software Engineering Notes*, ACM, vol. 19, n. 1, jan. de 1994, p. 76-85.

[SEA93] Sears, A., "Layout Appropriateness: A Metric for Evaluating User Interface Widget Layout", *IEEE Trans. Software Engineering*, vol. SE-19, n. 7, jul. de 1993, p. 707-719.

[SHE98] Sheppard, M., Goal, *Question, Metric*, 1998, disponível em http://dec. bournemouth.ac.uk/ESERG/mshepperd/ SEMGQM.html.

[SOL99] Van Solingen, R. e Berghout, E., *The Goal/Question/Metric Method*, McGraw-Hill, 1999.

[UEM99] Uemura, T., S. Kusumoto e Inoue, K., "A Function Point Measurement Tool for UML Design Specifications", *Proc. of Sixth International Symposium on Software Metrics*, IEEE, November 1999, p. 62-69.

[USA87] *Management Quality Insight*, AFCSP 800-14 (U.S. Air Force), jan. de 20, 1987.

[WHI97] Whitmire, S., *Object-Oriented Design Measurement*, Wiley, 1997.

[WIL93] Wilde, N. e Huitt, R., "Maintaining Object-Oriented Software", *IEEE Software*, jan. de 1993, p. 75-80.

[ZUS90] Zuse, H., *Software Complexity: Measures and Methods*, DeGruyter, 1990.

[ZUS97] Zuse, H., *A Framework of Software Measurement*, DeGruyter, 1997.

Problemas e Pontos a Considerar

15.1. A teoria de medição é um tópico avançado que se apóia fortemente em métricas de software. Usando [ZUS97], [FEN91], [ZUS90], [KYB84] ou alguma outra fonte, redija um trabalho resumido que descreva os principais pontos da teoria de medição. Projeto individual: Desenvolva uma apresentação sobre o assunto e faça essa apresentação para a sua classe.

15.2. Os fatores de qualidade de McCall foram desenvolvidos durante os anos 70. Quase todos os tópicos da computação sofreram mudanças radicais desde a época em que foram desenvolvidos, no entanto, os fatores de McCall continuam a se aplicar ao software moderno. Você pode tirar alguma conclusão com base nesse fato?

15.3. Por que não pode ser desenvolvida uma métrica única, abrangente para a complexidade ou a qualidade de um programa?

Document shared on www.docsity.com

15.4. Tente melhorar uma medida ou métrica da vida diária que viola os atributos de métricas efetivas de software definidas na Seção 15.2.5.

15.5. Um sistema tem 12 entradas externas, 24 saídas externas, 30 diferentes campos de consultas externas, gerencia 4 arquivos lógicos internos e interfaces com 6 diferentes sistemas legados (6EIFs). Todos esses dados são de complexidade média e o sistema global é relativamente simples. Calcule o FP para o sistema.

15.6. O Software para o Sistema *X* tem 24 requisitos funcionais individuais e 14 requisitos não funcionais. O que é a especificidade dos requisitos? E completeza?

15.7. Um sistema de informação importante tem 1.140 módulos. Há 96 módulos que realizam funções de controle e coordenação e 490 módulos cuja função depende de processamento anterior. O sistema processa aproximadamente 220 objetos de dados, cada um dos quais tem em média três atributos. Há 140 itens únicos e 90 segmentos diferentes no banco de dados. Finalmente, 600 módulos têm pontos de entrada e saída únicos. Calcule o DSQI para esse sistema.

15.8. Uma classe **X** tem 12 operações. A complexidade ciclomática foi calculada para todas as operações do sistema OO e o valor médio da complexidade de módulo é 4. Para a classe **X**, a complexidade das operações 1 a 12 é 5, 4, 3, 3, 6, 8, 2, 2, 5, 5, 4, 4, respectivamente. Calcule os métodos ponderados por classe.

15.9. Desenvolva uma ferramenta de software que calcule a complexidade ciclomática de um módulo para uma linguagem de programação. Você pode escolher a linguagem.

15.10. Desenvolva uma pequena ferramenta de software que realiza a análise de Halstead em código-fonte, na linguagem de programação de sua escolha.

15.11. Um sistema herdado tem 940 módulos. A última versão exigiu que 90 desses módulos fossem modificados. Além disso, 40 novos módulos foram adicionados e 12 módulos antigos foram removidos. Calcule o índice de maturidade de software do sistema.

Leituras e Fontes de Informação Adicionais

Há um número surpreendentemente grande de livros que são dedicados a métricas de software, apesar de a maioria focalizar métricas de processo e de projeto, deixando de lado as métricas técnicas. Kan *(Metrics and Models in Software Quality Engineering,* Addison-Wesley, second edition, 2002), Fenton e Pfleeger *(Software Metrics: A Rigourous and Practical Approach,* Brooks-Cole Publishing, 1998) e Zuse [ZUS97] escreveram o mais profundo tratado de métricas de produto.

Os livros de Card e Glass [CAR90], Zuse [ZUS90], Fenton [FEN91], Ejiogu [EJI91], Moeller e Paulish *(Software Metrics,* Chapman e Hall, 1993) e Hetzel [HET93] tratam todos de métricas técnicas em algum detalhe. Oman e Pfleeger *(Applying Software Metrics,* IEEE Computer Society Press, 1997) editaram uma antologia de trabalhos importantes sobre métricas de software. Além disso, os seguintes livros merecem ser examinados:

Conte, S. D., Dunsmore, H. E. e Shen, V. Y. *Software Engineering Metrics and Models*, Benjamin/Cummings, 1984.

Grady, R. B., *Practical Software Metrics for Project Management and Process Improvement*, Prentice-Hall, 1992.

Sheppard, M., *Software Engineering Metrics,* McGraw-Hill, 1992.

A teoria de medição de software é apresentada por Denvir, Herman e Whitty em uma coleção editada de trabalho *(Proceedings of the International BCS-FACS Workshop: Formal Aspects of Measurement,* Springer-Verlag, 1992). Shepperd *(Foundations of Software Measurement,* Prentice-Hall, 1996) também trata da teoria das medições em algum detalhe. Pesquisa atual é apresentada nos *Proceedings of the Symposium on Software Metrics* (IEEE, publicados anualmente).

Um resumo abrangente de dezenas de métricas de software úteis é apresentado em [IEE94]. Em geral, uma discussão de cada métrica foi destilada até as "primitivas" essenciais necessárias para calcular a métrica e os relacionamentos adequados para efetuar o cálculo. Um apêndice fornece discussão e muitas referências.

Whitmire [WHI97] apresenta a abordagem mais abrangente e sofisticada matematicamente das métricas OO publicada até hoje. Lorenz e Kidd [LOR94] e Hendersen-Sellers *(Object-Oriented Metrics: Measures of Complexity,* Prentice-Hall, 1996) oferecem os únicos outros livros dedicados a métricas OO. Hutcheson *(Software Testing Fundamentals: Methods and Metrics,* Wiley, 2003) apresenta diretrizes úteis para aplicação e uso de métricas para teste de software.

Uma ampla variedade de fontes de informação sobre métricas de software está disponível na Internet. Uma lista atualizada de referências relevantes para métricas de software pode ser encontrada no site deste livro: **http://www.mhhe.com/pressman**.

Document shared on www.docsity.com

PARTE

TRÊS

APLICAÇÃO DE ENGENHARIA DA WEB

Nesta parte de *Engenharia de Software* você aprenderá sobre os princípios, conceitos e métodos que são usados para criar aplicações de alta qualidade baseadas na Web. Estas questões são tratadas nos capítulos que se seguem:

- As aplicações baseadas na Web (WebApps) são diferentes de outros tipos de software?
- O que é engenharia da Web e que elementos da prática de engenharia de software ela pode adotar?
- Quais são os elementos de um processo de engenharia da Web?
- Como é formulado e planejado um projeto de engenharia da Web?
- Como os requisitos de WebApps são analisados e modelados?
- Quais conceitos e princípios guiam um profissional no projeto de WebApps?
- Como conduzir o projeto de arquitetura, interface e navegação de WebApps?
- Que técnicas de construção podem ser aplicadas para implementar o modelo de projeto?
- Que conceitos, princípios e métodos de teste são aplicáveis à engenharia da Web?

Uma vez respondidas essas questões, você vai estar mais bem preparado para desenvolver a engenharia de aplicações de alta qualidade baseadas na Web.

CAPÍTULO 16

ENGENHARIA DA WEB

CONCEITOS-CHAVE

arcabouço de processo 383
critérios de qualidade 386
engenharia da Web
ferramentas 382
métodos 382
processo 381
melhores práticas 385
questões básicas 385
WebApp
atributos 379
categorias 381

A World Wide Web e a Internet, que lhe dá poder, são indiscutivelmente os desenvolvimentos mais importantes da história da computação. Essas tecnologias nos colocaram (com bilhões mais que irão eventualmente nos seguir) na era da informação. Elas se tornaram integradas à vida diária na primeira década do século XXI.

Para aqueles que ainda se lembram de um mundo sem a Web, o crescimento caótico da tecnologia nos remete ao passado, a outra época — os primeiros dias do software. Era uma época de pouca disciplina, mas de enorme entusiasmo e criatividade. Era uma época em que programadores freqüentemente juntavam sistemas — alguns bons, outros ruins. A atitude prevalente parecia ser "Faça rápido e ponha em prática, vamos melhorar (e entender melhor o que realmente precisávamos construir) à medida que avançarmos". Parece familiar?

Em uma mesa-redonda virtual, publicada no *IEEE Software* [PRE98], marquei a minha posição a respeito da engenharia da Web:

> Parece-me que praticamente todo produto ou sistema importante merece passar por engenharia. Antes de começar a construí-lo é melhor que você entenda o problema, projete uma solução que funcione, implemente-a de um modo sólido e teste-a em profundidade. Você provavelmente deve controlar também as modificações realizadas à medida que trabalha, e deve ter algum mecanismo que assegure a qualidade do resultado final. Muitos desenvolvedores da Web não discutem isso; eles apenas pensam que seu mundo é realmente diferente e que as abordagens convencionais de engenharia de software simplesmente não se aplicam.

PANORAMA

O que é? Sistemas e aplicações baseados na Web (WebApps) produzem uma complexa matriz de conteúdo e funcionalidade para uma ampla população de usuários finais. A engenharia da Web (WebE) é o processo usado para criar WebApps de alta qualidade. A WebE não é um clone perfeito da engenharia de software, mas toma emprestados muitos dos conceitos e princípios fundamentais da engenharia de software. Além disso, o processo WebE enfatiza atividades técnicas e de gestão similares. Há diferenças sutis no modo pelo qual essas atividades são conduzidas, mas a filosofia dominante determina uma abordagem disciplinada para o desenvolvimento de um sistema baseado em computador.

Quem faz? Engenheiros da Web e desenvolvedores de conteúdo não-técnico criam a WebApp.

Por que é importante? À medida que as WebApps se tornam cada vez mais integradas nas estratégias de negócio, para empresas pequenas e grandes (por exemplo, comércio eletrônico — e-commerce), a necessidade de construir sistemas confiáveis, usáveis e adaptáveis cresce em importância. É por isso que é necessária uma abordagem disciplinada para o desenvolvimento de WebApps.

Quais são os passos? Como qualquer disciplina de engenharia, a WebE aplica uma abordagem genérica que é combinada com estratégias, táticas e métodos especializados. O processo deWebE começa com uma formulação do problema a ser resolvido pela WebApp. O projeto WebE é planejado e os requisitos e o projeto da WebApp são modelados. O sistema é construído usando tecnologias e ferramentas especializadas associadas com a Web. É então entregue aos usuários finais e avaliado usando tanto critérios técnicos quanto de negócio. Como as WebApps evoluem continuamente, devem ser estabelecidos mecanismos para controle de configuração, garantia de qualidade e suporte continuado.

Qual é o produto do trabalho? Diversos produtos do trabalho de WebE são produzidos. O resultado final é a WebApp operacional.

Como tenho certeza que fiz corretamente? Algumas vezes é difícil ter certeza até que os usuários finais exercitem a WebApp. No entanto, práticas de SQA podem ser aplicadas para garantir a qualidade dos modelos WebE, no conteúdo e funções, usabilidade, desempenho e segurança de todo o sistema.

Isso nos leva a uma questão importante: *os princípios de engenharia de software, seus conceitos e métodos podem ser aplicados ao desenvolvimento na Web?* Muitos deles podem, mas sua aplicação de certo modo pode exigir uma abordagem diferente.

Mas, e se a abordagem atual para o desenvolvimento na Web continuar? Na ausência de um processo disciplinado para o desenvolvimento de sistemas baseados na Web, há uma preocupação crescente de que poderemos enfrentar sérios problemas no desenvolvimento, na implantação e na manutenção bem-sucedidos. De fato, a infra-estrutura de aplicação que estamos criando hoje pode levar a algo que poderia ser chamado "Web entrelaçada", à medida que avançamos neste novo século. Essa frase representa um amontoado de aplicações maldesenvolvidas baseadas na Web, que têm uma probabilidade muito alta de falha. Pior, à medida que os sistemas baseados na Web tornam-se mais complexos, uma falha em um deles pode e vai propagar problemas de base ampla para muitos outros. Quando isso acontecer, a confiança em toda a Internet pode ser abalada irreparavelmente. Pior ainda, pode levar a uma regulamentação governamental desnecessária e malconcebida, causando dano irreparável a essas tecnologias singulares.

Para evitar uma Web entrelaçada e alcançar maior sucesso no desenvolvimento e na aplicação de sistemas complexos e de grande escala baseados na Web, há uma necessidade premente de abordagens disciplinadas de engenharia da Web, e novos métodos e ferramentas para o desenvolvimento, a implantação e a avaliação de sistemas e aplicações baseados na Web. Tais abordagens e técnicas devem levar em conta as características especiais do novo meio, os ambientes e cenários operacionais, e a multiplicidade de perfis de usuários, que adicionam desafios ao desenvolvimento de aplicações baseadas na Web.

A engenharia da Web (*Web Engineering* — WebE) aplica "princípios científicos sólidos, de engenharia e de gestão, e abordagens disciplinadas e sistemáticas para o bem-sucedido desenvolvimento, implantação e manutenção de sistemas e aplicações de alta qualidade baseados na Web." [MUR99].

16.1 ATRIBUTOS DE SISTEMAS E APLICAÇÕES BASEADOS NA WEB

Nos primeiros dias da World Wide Web (entre 1990 e 1995), os "sites Web" eram formados de pouco mais do que um conjunto de arquivos de hipertexto ligados que apresentavam informação usando texto e um pouco de gráficos. Com o passar do tempo, a HTML foi crescendo com ferramentas de desenvolvimento (por exemplo, XML, Java) que habilitaram os engenheiros Web a fornecer capacidade computacional junto com informação. Sistemas e aplicações[1] baseados na Web (vamos nos referir a esses, coletivamente, como WebApps) nasceram. Hoje em dia, WebApps evoluíram para ferramentas computacionais sofisticadas que não fornecem somente funções isoladas para o usuário final, mas também são integradas com banco de dados coorporativos e aplicações de negócio.

> "Na hora em que virmos qualquer espécie de estabilização, a Web terá se tornado algo completamente diferente."
>
> Louis Monier

Há pouca discussão sobre o fato de que WebApps *são* diferentes de muitas outras categorias de software de computador, discutidas no Capítulo 1. Powell resume as principais diferenças, quando afirma que sistemas baseados na Web "envolvem uma mistura de publicação impressa e desenvolvimento de software, de comercialização e computação, de comunicações internas e relações externas, e de arte e tecnologia" [POW98]. Os seguintes atributos são encontrados na vasta maioria de WebApps:

Concentração em Redes. Uma WebApp reside em uma rede e precisa servir às necessidades de uma comunidade diversificada de clientes. Uma WebApp pode residir na Internet (conseqüentemente, permitindo comunicação aberta ao mundo todo). Alternativamente, uma aplicação pode

1 No contexto deste capítulo, o termo "aplicação Web" (WebApp) engloba tudo, de uma simples página Web que poderia ajudar um consumidor a calcular o pagamento do aluguel de um automóvel a um abrangente site Web que fornece serviços completos de viagem para pessoas de negócios e em férias. Estão incluídos nessa categoria sites Web completos, funcionalidade especializada dentro de sites e aplicações de processamento de informação que residem na Internet, em uma intranet ou extraNet.

Document shared on www.docsity.com

AVISO

Pode-se argumentar que uma aplicação tradicional em qualquer domínio de software, discutido no Capítulo 1, pode exibir essa lista de atributos. No entanto, WebApps quase sempre o fazem.

ser colocada em uma intranet (implementando as comunicações dentro de uma organização) ou em uma extranet (comunicação entre redes).

Concorrência. Um grande número de usuários pode ter acesso à WebApp ao mesmo tempo. Em muitos casos, os padrões de utilização entre os usuários finais vão variar muito.

Carga imprevisível. O número de usuários da WebApp pode variar por ordens de magnitude de um dia para o outro. Cem usuários podem aparecer na segunda-feira; dez mil podem usar o sistema na terça-feira.

Desempenho. Se um usuário da WebApp tem de esperar muito (para acesso, para processamento do lado do servidor, para formatação ou exibição do lado do cliente), ele ou ela pode decidir ir para outro lugar.

Disponibilidade. Embora a expectativa de 100% de disponibilidade seja irracional, usuários de WebApps populares freqüentemente querem acesso na base de "24/7/365". Usuários da Austrália ou da Ásia podem querer acesso durante horários em que as aplicações domésticas de software tradicional na América do Norte poderiam estar fora do ar para a manutenção.

Voltada a dados. A função principal de muitas WebApps é usar hipermídia para apresentar conteúdos de texto, gráficos, áudio e vídeo ao usuário final. Além disso, WebApps são comumente usadas para dar acesso a informação que existe em bancos de dados que não eram originariamente parte integral de um ambiente baseado na Web (por exemplo, comércio eletrônico — e-commerce — ou aplicações financeiras).

Sensível ao conteúdo. A qualidade e natureza estética do conteúdo permanecem como um considerável determinante da qualidade de uma WebApp.

Evolução continuada. Diferentemente do software de aplicação convencional, que evolui ao longo de uma série de versões planejadas e cronologicamente espaçadas, as aplicações Web evoluem continuamente. Não é incomum que algumas WebApps (especificamente, seu conteúdo) sejam atualizadas por meio de um cronograma minuto a minuto ou que o conteúdo seja independentemente calculado para cada solicitação. Alguns alegam que a evolução contínua de WebApps torna o trabalho realizado nelas análogo à jardinagem. Lowe [LOW99] discute isso quando escreve:

> Engenharia está relacionada com a adoção de uma abordagem consistente e científica, misturada com um contexto prático específico para o desenvolvimento e o comissionamento de sistemas ou aplicações. O desenvolvimento de sites Web, está, com freqüência, muito mais relacionado à criação de uma infra-estrutura (dispondo as linhas mestras do jardim) e depois ao "cultivo" da informação, que cresce e floresce dentro desse jardim. Ao longo do tempo, o jardim (por exemplo, o site) vai continuar a evoluir, a se modificar e a crescer. Uma boa arquitetura inicial deve permitir que esse crescimento ocorra de um modo controlado e consistente...

Cuidado e alimentação contínua permitem que um site Web cresça (em robustez e importância). Mas, diferentemente de um jardim, aplicações da Web devem servir (e se adaptar) às necessidades de outros, além do jardineiro.

Imediatismo. Embora *imediatismo* — a necessidade premente de obter software para ser colocado rapidamente no mercado — seja uma característica de muitos domínios de aplicação, as WebApps freqüentemente exibem um prazo de colocação no mercado que pode ser questão de alguns dias ou semanas[2]. Os engenheiros Web precisam usar métodos de planejamento, análise, projeto, implementação e teste que tenham sido adaptados aos cronogramas de tempo reduzido, requeridos para o desenvolvimento de WebApp.

Segurança. Como as WebApps estão disponíveis por meio de acesso em rede, é difícil, senão impossível, limitar a população de usuários finais que podem ter acesso à aplicação. A fim de proteger conteúdo reservado e fornecer modos seguros de transmissão de dados, fortes medidas de segurança precisam ser implementadas em toda a infra-estrutura que apóia uma WebApp e na aplicação propriamente dita.

Estética. Uma inegável parte da atração de uma WebApp é o seu aspecto. Quando uma aplicação é projetada para o mercado ou para vender produtos ou idéias, a estética pode ter tanto a ver com o sucesso quanto o projeto técnico.

2 Com ferramentas modernas, páginas Web sofisticadas podem ser produzidas em apenas algumas horas.

Esses atributos gerais aplicam-se a todas as WebApps, mas com diferentes graus de influência.

Mas, e quanto às WebApps em si? De quais problemas que elas tratam? As seguintes categorias de aplicação são mais comumente encontradas no trabalho de WebE [DAR99]:

? Que categorias de WebApps são encontradas no trabalho de WebE?

- *Informacional* — conteúdo somente de leitura é fornecido com navegação e ligações (links) simples.
- *Para baixar* — um usuário baixa informação (faz download) de um servidor adequado.
- *Adaptável* — o usuário adapta o conteúdo a necessidades específicas.
- *Interação* — a comunicação entre uma comunidade de usuários ocorre por intermédio de salas de bate-papo, quadros de aviso ou mensagens instantâneas.
- *Entrada do usuário* — entrada baseada em formulários é o principal mecanismo para comunicar a necessidade.
- *Orientada a transação* — o usuário faz uma solicitação (por exemplo, coloca um pedido) que é atendida pela WebApp.
- *Orientada a serviços* — a aplicação fornece um serviço ao usuário (por exemplo, ajuda o usuário a calcular um pagamento de hipoteca).
- *Portal* — a aplicação orienta o usuário para outros conteúdos ou serviços da Web fora do domínio de aplicação do portal.
- *De acesso a banco de dados* — o usuário consulta uma grande base de dados e extrai informação.
- *Armazém de dados* — o usuário consulta uma coleção de grandes bancos de dados e extrai informação.

Os atributos mencionados anteriormente nesta seção e as categorias de aplicação mencionadas representam fatos da vida importantes para os engenheiros da Web. A solução é conviver com as restrições impostas por esses atributos e ainda assim produzir uma WebApp bem-sucedida.

16.2 As Camadas de Engenharia da WebApp

O desenvolvimento de sistemas e aplicações baseados na Web incorpora modelos de processo especializados, métodos de engenharia de software adaptados às características do desenvolvimento de WebApp e um conjunto importante de tecnologias de autorização. Processos, métodos e tecnologias (ferramentas) fornecem uma abordagem em camadas para WebE que é conceitualmente idêntica às camadas de software descritas na Figura 2.1.

> "Engenharia da Web trata de abordagens disciplinadas e sistemáticas para o desenvolvimento, a implantação e a manutenção de sistemas e aplicações baseados na Web."
>
> Yogesh Deshpande

AVISO

O processo de WebE é freqüentemente ágil e é quase sempre incremental. Observe, no entanto, que o modelo ágil pode não ser escolhido para a maioria dos principais projetos de engenharia Web.

16.2.1 Processo

Os modelos de processo WebE (discutidos em detalhe na Seção 16.3) adotam a filosofia do desenvolvimento ágil (Capítulo 4). Desenvolvimento ágil enfatiza uma abordagem de desenvolvimento simples que incorpora ciclos rápidos de desenvolvimento. Aoyama [AOY98] descreve a motivação para a abordagem ágil da seguinte forma:

> A Internet modificou a principal prioridade do desenvolvimento de software do *o quê* para *quando*. O tempo reduzido de colocação no mercado tornou-se o aspecto competitivo pelo qual as principais empresas lutam. Assim, reduzir o ciclo de desenvolvimento é agora uma das missões mais importantes dos engenheiros de software.

Document shared on www.docsity.com

Mesmo quando ciclos de tempo rápido dominam o raciocínio de desenvolvimento, é importante reconhecer que o problema deve ainda ser analisado, um projeto deve ser desenvolvido, implementação deve ser procedida de modo incremental e uma abordagem organizada de teste deve ser iniciada. No entanto, essas atividades de arcabouço devem ser definidas em um processo que (1) acolhe modificações, (2) encoraja a criatividade e independência da equipe de desenvolvimento e a forte interação com os interessados nas WebApps, (3) constrói sistemas usando pequenas equipes de desenvolvimento, e (4) enfatiza o desenvolvimento evolutivo ou incremental usando ciclos curtos de desenvolvimento [MCD01].

16.2.2 Métodos

O panorama de métodos da WebE engloba um conjunto de tarefas técnicas que habilitam um engenheiro Web a entender, caracterizar e então construir uma WebApp de alta qualidade. Os métodos WebE (discutidos em detalhe nos capítulos 18 a 20) podem ser categorizados da seguinte maneira:

AVISO

É importante notar que muitos métodos da WebE foram adaptados diretamente de seus correlatos da engenharia de software. Outros estão em seus estágios de formação. Alguns deles vão sobreviver, outros serão descartados à medida que melhores abordagens forem sugeridas.

Métodos de comunicação — definem a abordagem usada para facilitar a comunicação entre os engenheiros Web e todos os outros interessados na WebApp (por exemplo, usuários finais, clientes de negócio, especialistas no domínio do problema, projetistas de conteúdo, líderes de equipe, gerentes de projeto). As técnicas de comunicação são particularmente importantes durante a coleta de requisitos e sempre que um incremento da WebApp precisa ser avaliado.

Métodos de análise de requisitos — fornecem uma base para o entendimento do conteúdo a ser entregue por uma WebApp, a função a ser fornecida para o usuário final e os modos de interação que cada classe de usuários irá requerer à medida que a navegação pela WebApp ocorre.

Métodos de projeto — abrangem uma série de técnicas de projeto que cuidam do conteúdo, arquitetura da aplicação e da informação, projeto da interface e estrutura de navegação da WebApp.

Métodos de teste — incorporam revisões técnicas formais do conteúdo e modelo de projeto e uma ampla variedade de técnicas de teste, que tratam de tópicos no nível de componente e arquitetural, testes de navegação, testes de usabilidade, testes de segurança e testes de configuração.

É importante notar que embora métodos de WebE adotem muitos dos mesmos conceitos e princípios subjacentes aos métodos de engenharia de software descritos na Parte 2 deste livro, os mecanismos de análise, projeto e teste devem ser adaptados para acomodar as características especiais das WebApps.

Além dos métodos técnicos que acabaram de ser esboçados, uma série de atividades guarda-chuva (com métodos associados) é essencial para o sucesso da engenharia da Web, que inclui técnicas de gestão de projeto (por exemplo, estimativas, cronogramação, análise de riscos), técnicas de gestão de configuração de software e técnicas de revisões.

Veja na Web

Excelentes recursos para tecnologia WebE podem ser encontrados em **webdeveloper.com** e **www.eborcom.com/webmaker.**

16.2.3 Ferramentas e Tecnologia

Uma vasta gama de ferramentas e tecnologia tem evoluído desde a década passada à medida que as WebApps têm se tornado mais sofisticadas e difundidas. Essas tecnologias englobam uma grande variedade de descrição de conteúdo e linguagens de modelagem (por exemplo, HTML, VRML, XML), linguagens de programação (por exemplo, Java), recursos de desenvolvimento baseado em componentes (por exemplo, CORBA, COM, ActiveX, .NET), navegadores, ferramentas multimídia, ferramentas de autoria de sites, ferramentas de conectividade de banco de dados, ferramentas de segurança, servidores e utilitários de servidor, e ferramentas de gestão e análise de sites.

Uma discussão abrangente sobre ferramentas e tecnologia para engenharia Web está fora do escopo deste livro. O leitor interessado poderá visitar um ou mais dos seguintes sites Web: *Web Developer's Virtual Encyclopedia* (www.wdlv.com), *WebDeveloper* (www.webdeveloper.com), *Developer Shed* (www.devshed.com), *Webknowhow.net* (www.webknowhow.net), ou *WebReference* (www.webreference.com).

16.3 O Processo de Engenharia Web

Os atributos de sistemas e aplicações baseados na Web têm uma profunda influência no processo WebE que é escolhido. No Capítulo 3 mencionamos que um engenheiro de software escolhe um modelo de processo com base nos atributos do software que deve ser desenvolvido. O mesmo continua valendo para um engenheiro da Web.

Se imediatismo e evolução contínua são atributos principais de uma WebApp, uma equipe de engenharia da Web pode escolher um modelo de processo ágil (Capítulo 4) que produz versões WebApp em uma seqüência rápida. Por outro lado, se uma WebApp precisar ser desenvolvida durante um período de tempo maior (por exemplo, uma grande aplicação de e-commerce), um modelo de processo incremental (Capítulo 3) pode ser escolhido.

> "O desenvolvimento Web está na adolescência. . . Como a maioria dos adolescentes, quer ser aceito como adulto quando tenta se afastar de seus pais. Se estiver caminhando para atingir todo o seu potencial, precisa aprender algumas lições do mundo mais maduro de desenvolvimento de software."
>
> **Doug Wallace *et al.***

A natureza de concentração em rede do domínio das aplicações sugere uma população de usuários que é diversa (e, portanto, fazendo demanda especial quanto à elicitação e modelagem de requisitos) e uma arquitetura de aplicação que pode ser altamente especializada (e, portanto, fazendo exigências sobre o projeto). Como as WebApps são freqüentemente voltadas a conteúdo com ênfase na estética, é provável que atividades de desenvolvimento paralelo sejam planejadas no processo WebE e envolvam uma equipe com pessoal técnico e não técnico (por exemplo, redatores e projetistas gráficos).

16.3.1 Definição do Arcabouço

Qualquer um dos modelos ágeis de processo (por exemplo, Programação Extrema, Desenvolvimento de Software Adaptativo, SCRUM) apresentados no Capítulo 4, pode ser aplicado com sucesso em um processo WebE. O arcabouço de processo que é apresentado aqui é uma fusão dos princípios e idéias discutidos no Capítulo 4.

Para ser eficaz, qualquer processo de engenharia precisa ser adaptável. Isto é, a organização da equipe de projeto, os modos de comunicação entre os membros da equipe, as atividades de engenharia e as tarefas a serem realizadas, a informação que é coletada e criada, e os métodos usados para produzir um produto de alta qualidade precisam todos ser adaptados ao pessoal que está fazendo o trabalho, ao cronograma e às restrições do projeto, e ao problema a ser resolvido. Antes de definirmos um arcabouço de processo para WebE devemos reconhecer que:

PONTO CHAVE

O modelo de processo da WebE está fixado em três pontos: entrega incremental, modificações contínuas e cronogramas curtos.

1. *WebApps com freqüência são entregues incrementalmente.* Isto é, as atividades de arcabouço vão ocorrer repetidamente à medida que cada incremento é submetido à engenharia e entregue.
2. *Modificações ocorrerão freqüentemente.* Essas modificações podem ocorrer como resultado da avaliação de um incremento entregue ou como conseqüência de condições de negócio mutáveis.
3. *Cronogramas são curtos.* Isso alivia a criação e revisão de volumosa documentação de engenharia, mas não despreza a simples realidade de que análise, projeto e teste crítico devem ser registrados de algum modo.

Além disso, os princípios definidos como parte do "Manifesto de Desenvolvimento Ágil de Software" (Capítulo 4) devem ser aplicados. No entanto, os princípios não são os Dez Mandamentos. É algumas vezes razoável adotar o espírito desses princípios sem necessariamente abrir mão do que diz o manifesto.

Com esses tópicos em mente, discutimos o processo WebE no arcabouço de processo genérico apresentado no Capítulo 2:

Document shared on www.docsity.com

PONTO CHAVE

O modelo de processo genérico (introduzido no Capítulo 2) é aplicável à engenharia Web.

Comunicação com o cliente. No processo WebE, a comunicação com o cliente é caracterizada por duas tarefas principais: análise e formulação de negócio. *Análise de negócio* define o contexto do negócio/organizacional para a WebApp. Além disso, os interessados são identificados, modificações potenciais no ambiente ou requisitos do negócio são previstos e a integração entre a WebApp e outras aplicações de negócio, bancos de dados e funções são definidas. *Formulação* é uma atividade de coleta de requisitos que envolve todos os interessados. O objetivo é descrever o problema que a WebApp deve resolver (junto com os requisitos básicos para a WebApp) usando a melhor informação disponível. Além disso, uma tentativa é feita para identificar áreas de incerteza e onde potenciais modificações vão ocorrer.

Planejamento. O plano de projeto para o incremento da WebApp é criado. O plano consiste de uma definição de tarefa e de um cronograma para o período de tempo (normalmente medido em semanas) projetado para o desenvolvimento do incremento da WebApp.

Modelagem. As tarefas de análise e projeto de engenharia de software convencional são adaptadas ao desenvolvimento da WebApp, intercaladas, e então fundidas na atividade de modelagem da WebE (Capítulos 18 e 19). O objetivo é desenvolver modelos de análise e projeto "rápidos" que definam requisitos e ao mesmo tempo representem uma WebApp que vai satisfazê-los.

Construção. Ferramentas e tecnologias da WebE são aplicadas para construir a WebApp que foi modelada. Uma vez construído o incremento da WebApp, uma série de testes rápidos é conduzida para garantir que erros no projeto (isto é, conteúdo, arquitetura, interface e navegação) sejam descobertos. Teste adicional cuida de outras características da WebApp.

Implantação. A WebApp é configurada para o seu ambiente operacional, entregue aos usuários finais e então começa um período de avaliação. Realimentações da avaliação são apresentadas à equipe de WebE, e o incremento é modificado como necessário.

Essas cinco atividades de arcabouço da WebE são aplicadas usando um fluxo de processo incremental como mostrado na Figura 16.1

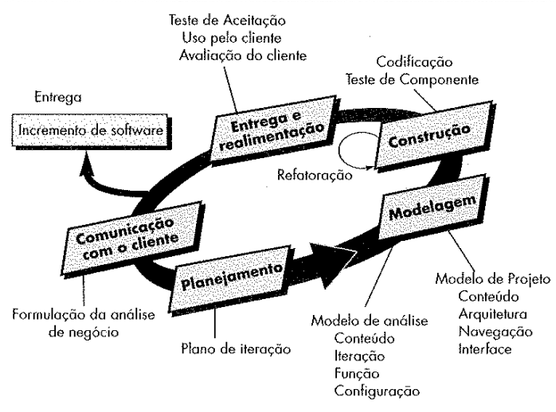
16.3.2 Refinamento do Arcabouço

Já notamos que o modelo de processo WebE deve ser adaptável. Isto é, uma definição de tarefas de engenharia necessárias para refinar cada atividade de arcabouço é deixada ao critério da equipe de engenharia Web. Em alguns casos, uma atividade de arcabouço é conduzida informalmente, em outros, uma série de tarefas distintas será definida e conduzida pelos membros da equipe. Em qualquer caso, a equipe tem a responsabilidade de produzir um incremento WebApp de alta qualidade dentro do período de tempo alocado.

É importante enfatizar que tarefas associadas às atividades de arcabouço WebE podem ser modificadas, eliminadas ou estendidas com base nas características do problema, do produto, do projeto e das pessoas da equipe de engenharia Web.

FIGURA 16.1

O processo WebE



INFO

Engenharia da Web — Questões Básicas

A engenharia de qualquer produto envolve sutilezas que não são imediatamente óbvias para aqueles que não têm experiência considerável. As características das WebApps forçam os engenheiros da Web a responderem a uma variedade de questões que deverão ser atendidas durante as atividades iniciais de arcabouço. Questões estratégicas relacionadas às necessidades do negócio e objetivos do produto são atendidas durante a formulação. Questões de requisitos relacionadas às características e funções precisam ser consideradas durante a modelagem de análise. Questões de projeto de base ampla relacionadas à arquitetura, características de interface e tópicos navegacionais da WebApp são consideradas à medida que o modelo de projeto evolui. Finalmente, um conjunto de tópicos humanos, relacionados à maneira pela qual um usuário realmente interage com a WebApp, é tratado continuamente. Susan Weinshenk [WEI02] sugere um conjunto de questões que deve ser considerado à medida que a análise e o projeto progridem. Um pequeno subconjunto (adaptado) é aqui mencionado:

- Quão importante é uma página principal de um site Web? Ela deve conter informação útil ou uma simples lista de links que levam um usuário a ter mais detalhe em níveis mais baixos?
- Qual é o leiaute de página mais efetivo (por exemplo, menu no topo, à direita ou à esquerda?), e ele varia dependendo de tipo da WebApp que está sendo desenvolvido?
- Que opções de mídia têm maior impacto? Os gráficos são mais efetivos do que o texto? O vídeo (ou áudio) é uma opção eficaz? Quando as várias opções de mídia devem ser escolhidas?
- Quanto de trabalho pode-se esperar que um usuário faça quando ele ou ela está procurando a informação? Quantos cliques as pessoas estão dispostas a fazer?
- Quão importantes são os apoios navegacionais quando as WebApps são complexas?
- Quão complexo pode ser o formulário de entrada antes que ele se torne irritante para o usuário? Como os formulários de entrada podem ser manipulados?
- Quão importantes são os recursos de busca? Que porcentagem de usuários navega, e que porcentagem usa buscas específicas? Quão importante é estruturar cada página de um modo que aceite um *link* de alguma fonte externa?
- A WebApp será projetada para que seja acessível àqueles que têm deficiências físicas ou outras?

Não há respostas definitivas a questões como essas, e, no entanto, elas devem ser tratadas à medida que a WebE progride. Consideraremos respostas potenciais nos capítulos 17 a 20.

> "Existem aqueles entre nós que acreditam que as melhores práticas para desenvolvimento de software são práticas e merecem implementação. E depois há aqueles entre nós que acreditam que as melhores práticas são interessantes de um ponto de vista acadêmico, mas não o são para o mundo real... muito obrigado."
>
> Warren Keuffel

16.4 MELHORES PRÁTICAS DE ENGENHARIA WEB

Todo desenvolvedor de WebApp usará o arcabouço de processo WebE e o conjunto de tarefas definido na Seção 16.3? Provavelmente, não. As equipes de engenharia Web estão, algumas vezes, sob enorme pressão de tempo e tentam pegar atalhos (mesmo que estes sejam imprudentes e resultem em *mais* esforços de desenvolvimento, não menos). Mas, um conjunto fundamental de melhores práticas — adotado por meio das práticas de engenharia de software discutidas ao longo da Parte 2 deste livro — deveria ser aplicado se WebApps de qualidade industrial vão ser construídas.

AVISO

Certifique-se de que a necessidade do negócio de uma WebApp tenha sido claramente enunciada por alguém. Se ela não foi, seu projeto WebE corre risco.

1. *Empregue tempo para entender as necessidades do negócio e os objetivos do produto, mesmo se os detalhes da WebApp forem vagos.* Muitos desenvolvedores de WebApp erroneamente acreditam que requisitos vagos (que são muito comuns) os aliviam da necessidade de se certificarem de que o sistema que eles estão construindo tem um objetivo legítimo em termos do negócio. O resultado final é (muito freqüentemente) bom trabalho técnico que resulta em um sistema errado construído por razões erradas para público errado. Se os interessados não podem enunciar uma necessidade de negócio para a WebApp, prossiga com extremo

Document shared on www.docsity.com

cuidado. Se os interessados se esforçam para identificar um conjunto de objetivos claros para o produto (WebApp), não prossiga até que eles possam fazê-lo.

2. Descreva como os usuários irão interagir com a WebApp usando uma abordagem baseada em cenário. Os interessados precisam ser convencidos a desenvolver casos de uso (discutidos ao longo da Parte 2 deste livro) para refletir como vários atores irão interagir com a WebApp. Esses cenários podem ser então usados (1) para planejar e acompanhar o projeto, (2) para guiar a modelagem de análise e projeto, e (3) como entrada importante para o projeto de testes.
3. Desenvolva um plano de projeto, mesmo que ele seja muito abreviado. Baseie o plano em um arcabouço de processo predefinido que seja aceitável a todos os interessados. Como os prazos do projeto são muito curtos, a granularidade do cronograma deve ser fina, isto é, em muitas instâncias, o projeto deve ser cronogramado e acompanhado diariamente.
4. Empregue algum tempo modelando o que você está querendo construir. Geralmente, modelos de análise e projeto abrangentes não são desenvolvidos durante a engenharia da Web. No entanto, classes e diagramas de seqüência UML juntamente com outras notações UML selecionadas (por exemplo, diagramas de estado) podem fornecer uma visão inestimável.
5. Revise os modelos quanto à consistência e qualidade. Revisões técnicas formais (Capítulo 26) devem ser conduzidas durante um projeto WebE. O tempo gasto com revisões paga importantes dividendos porque freqüentemente elimina o retrabalho e resulta em uma WebApp que exibe alta qualidade — aumentando, assim, a satisfação do cliente.
6. Use ferramentas e tecnologia que lhe possibilitem construir o sistema com tantos componentes reusáveis quanto possível. Uma ampla variedade de ferramentas está disponível virtualmente para todos os tópicos de construção da WebApp. Muitas dessas ferramentas possibilitam a um engenheiro da Web construir partes significativas da aplicação usando componentes reusáveis.
7. Não confie nos primeiros usuários para depurar a WebApp — projete testes abrangentes e execute-os antes de entregar o sistema. Usuários de uma WebApp vão freqüentemente lhe dar uma chance. Se ela falhar na execução, eles vão embora — para nunca mais voltar. É essa a razão que "teste primeiro, depois implante" deve ser a filosofia prevalente, mesmo se os prazos finais precisarem ser esticados.

INFO

Critérios de Qualidade/Diretrizes para WebApps

A WebE esforça-se para produzir WebApps de alta qualidade. Mas, o que é "qualidade" nesse contexto, e quais diretrizes estão disponíveis para alcançá-la? Em seu artigo sobre garantia de qualidade para sites Web, Quibeldey-Cirkel [QUI01] sugere um conjunto abrangente de recursos *on-line* que trata desses tópicos:

W3C: Diretrizes de Estilo para Hipertextos On-line (W3C: Style Guide for Online Hypertext) www.w3.org/Provider/Style

O Guia de Sevloid para Projeto Web (The Sevloid Guide to Web Design) www.sev.com.au/webzone/design/guide.asp

Páginas da Web que Atraem (Web Pages That Suck) www.webpagesthatsuck.com/index.html

Recursos de Estilos Web (Resources on Web Style) www.westegg.com/unmaintained/badpages

Ferramenta da Gartner para Avaliação Web (Gartner's Web Evaluation Tool) www.gartner.com/ebusiness/website-ings

IBM Corp: Diretrizes Web (IBM Corp: Web Guidelines) www-3.ibm.com/ibm/easy/eou_ext.nsf/Publish/572

Usabilidade na World Wide Web (World Wide Web Usability) ijhcs.open.ac.uk

Sala da Vergonha da Interface (Interface Hall of Shame) www.iarchitect.com/mshame.htm

Arte e Zen de Sites Web (Art and the Zen of Web Sites) www.tlc-systems.com/webtips.shtml

Projeto para Web: Estudos Empíricos (Designing for the Web: Empirical Studies) www.microsoft.com/usability/webconf.htm

Useit.com de Nielsen (Nielsen's useit.com) www.useit.com

Qualidade da Experiência (Quality of Experience) www.qualityofexperience.org

Criação de Sites Web Excepcionais (Creating Killer Web Sites) www.killersites.com/core.html

Todas as Coisas da Web (All Things at Web) www.pantos.org/atw

Novos Projetos Web para SUN (SUN's New Web Design) www.sun.com/980113/sunonnet

Página de Bruce Tognazzini (Tognazzini, Bruce: Homepage) www.asktog.com

Macaco da Web (Webmonkey) hotwired.lycos.com/webmonkey/design/?tw=design

Os Melhores Sites Web do Mundo (World's Best WebSites) www.worldbestwebsites.com

Guia de Estilo Web da Yale (Yale University: Yale Web-Style Guide) info.med.yale.edu/caim/manual

16.5 Resumo

O impacto de sistemas e aplicações baseados na Web é discutivelmente o evento único mais significativo da história da computação. À medida que as WebApps crescem em importância, uma abordagem disciplinada de WebE — adaptada dos princípios, conceitos, processos e métodos da engenharia de software — começou a surgir.

WebApps são diferentes de outras categorias de software de computador. São intensamente voltadas para redes, guiadas por conteúdo e evoluem continuamente. O imediatismo que preside seu desenvolvimento, a necessidade prevalente de segurança na sua operação e a demanda por estética, bem como a distribuição de conteúdo funcional, são fatores de diferenciação adicionais. Como outros tipos de software, WebApps podem ser avaliadas usando uma variedade de critérios de qualidade que incluem: usabilidade, funcionalidade, confiabilidade, eficiência, manutenibilidade, segurança, disponibilidade, escalabilidade e tempo de colocação no mercado.

WebE pode ser descrita em três categorias — processo, métodos e ferramentas/tecnologia. O processo WebE adota a filosofia de desenvolvimento ágil que enfatiza uma abordagem de engenharia "simples" que leva à entrega incremental do sistema a ser construído. Um arcabouço genérico de processo — comunicação, planejamento, modelagem, construção e implantação — é aplicável à WebE. Essas atividades de arcabouço são refinadas em um conjunto de tarefas WebE que são adaptadas às necessidades de cada projeto. Um conjunto de atividades guarda-chuva análogas àquelas aplicadas durante o trabalho de engenharia de software SQA (*Software Quality Assurance*), SCM (*Software Configuration Management*) e gestão de projeto (*project management*) se aplica a todos os projetos WebE.

Referências Bibliográficas

[AOY98] Aoyama, M., "Web-Based Agile Software Development", *IEEE Computer*, nov./dez. de 1998, p. 56-65.

[DAR99] Dart, S., "Containing the Web Crisis Using Configuration Management", *Proc. First ICSE Workshop on Web Engineering*, ACM, Los Angeles, maio 1999. (*The Proceedings of the First ICSE Workshop on Web Engineering* são publicados on-line em http://fistserv.macarthur.uws.edu.au/san/icse99-WebE/ICSE99-WebE-Proc/ default.htm).

[FOW01] Fowler M. e Highsmith, J., "The Agile Manifesto", *Software Development Magazine*, ago. de 2001, http://www.sdmagazine.com/documents/s=844/ sdm0108a/0108a.htm.

[MCD01] McDonald, A. e Welland, R. *Agile Web Engineering (AWE) Process*, Department of Computer Science, University of Glascow, Technical Report TR-2001-98, 2001, disponível em http://www.dcs.gla.ac.uk/~andrew/TR-2001-98.pdf.

[MUR99] Murugesan, S., *WebE Home Page*, http://fistserv.macarthur.uws.edu.au/ san/ WebEHome, jul. 1999.

[NOR99] Norton, K., "Applying Cross Functional Evolutionary Methodologies to Web Development", *Proc. First ICSE Workshop on Web Engineering*, ACM, Los Angeles, maio de 1999.

[POW98] Powell, T. A., *Web Site Engineering*, Prentice-Hall, 1998.

[PRE98] Pressman, R. S. (moderador), "Can Internet-Based Applications Be Engineered?", *IEEE Software*, set. de 1998, p. 104-110.

[QUI01] Quibeldey-Cirkel, K., "Checklist for Web Site Quality Assurance", *Quality Week Europe*, 2001, disponível em www.fbi.fh-darmstadt.de/~quibeldey/ Projekte/QWE2001/ Paper_Quibeldey_Cirkel.pdf.

[WEI02] Weinschenk, S., "Psychology and the Web: Designing for People", 2002, http://www.weinschenk.com/learn/facts.asp.

Document shared on www.docsity.com

Problemas e Pontos a Considerar

16.1. Há outros atributos genéricos que diferenciam as WebApps de aplicações de software mais convencional? Tente citar dois.

16.2. Como você julga a "qualidade" de um *site* Web? Faça uma lista ordenada de 10 atributos de qualidade que você acredita ser os mais importantes.

16.3. Faça uma breve pesquisa e escreva um trabalho de duas ou três páginas que resuma uma das três tecnologias mencionadas na Seção 16.2.3.

16.4. Usando um site real como exemplo, ilustre as diferentes manifestações do "conteúdo" da WebApp.

16.5. Revise os processos de engenharia de software descritos nos Capítulos 3 e 4. Existe(m) outro(s) processo(s) — diferente(s) do modelo de processo ágil — que poderia(m) ser aplicável(veis) à engenharia Web? Em caso afirmativo, qual(quais) é(são) o(s) processo(s) e por quê?

16.6. Revise a discussão do "Manifesto para Desenvolvimento Ágil de Software" apresentado no Capítulo 4. Quais dos 12 princípios deveriam funcionar bem em um projeto de 2 anos (envolvendo dezenas de pessoas) que construirá um grande sistema de e-commerce para uma empresa automobilística? Quais dos 12 princípios deveriam funcionar bem em um projeto de 2 meses que construirá um site de informação para uma pequena empresa corretora de imóveis?

16.7. Faça uma lista de "riscos" que poderiam provavelmente ocorrer durante o desenvolvimento de uma nova aplicação de e-commerce que é projetada para vender telefones móveis e serviços diretamente pela Web.

Leituras e Fontes de Informação Adicionais

Centenas de livros que discutem um ou mais tópicos de engenharia da Web foram publicados nos últimos anos, apesar de relativamente poucos tratarem de todos os tópicos de WebE. Sarukkai *(Foundations of Web Technology,* Kluwar Academic Publishers, 2002) apresenta uma compilação de tecnologias que são necessárias para WebE que vale a pena. Murugusan e Deshpande *(Web Engineering: Managing Diversity and Complexity of Web Development,* Springer-Verlag, 2001) editaram uma coleção de artigos úteis sobre WebE. Minutas de conferências internacionais sobre Engenharia da Web e Engenharia de Sistemas de Informação são publicadas anualmente pelo IEEE Computer Society Press.

Flor *(Web Business Engineering,* Addison-Wesley, 2000) discute análise de negócios e tópicos relacionados que auxiliam o engenheiro da Web a melhor entender as necessidades do cliente. Bean *(Engineering Global E-Commerce Sites,* Morgan Kaufmann, 2003) apresenta diretrizes para desenvolvimento de WebApps globais. Lowe e Hall *(Hypermedia and the Web: An Engineering Approach,* Wiley, 1999) e Powell [POW98] fornecem cobertura razoavelmente completa. Umar *(Application Re-engineering: Building Web-Based Applications and Dealing with Legacy Systems,* Prentice-Hall, 1997) trata de um dos tópicos mais difíceis em WebE — a reengenharia de sistemas legados para torná-los compatíveis com sistemas baseados na Web. IEEE Std. 2001-1999 define práticas básicas de WebE.

Uma ampla variedade de fontes de informação sobre engenharia da Web está disponível na Internet. Uma lista atualizada de referências da World Wide Web que são relevantes pode ser encontrada no site deste livro: **http://www. mhhe. com/pressman.**

Formulação e Planejamento para Engenharia da Web

CAPÍTULO 17

Conceitos-Chave

comunicação 393
e-projetos 395
equipes 396
questões de formulação .. 390
gestão de projeto 398
metas e objetivos 391
métricas 404
coleta de requisitos 392
piores práticas 406
planejamento 398
terceirização 398
tipos de análise 395
WebE interna 401

Durante a efervescente década de 1990, o "boom" da Internet fez mais estardalhaço do que qualquer outro evento na história dos computadores. Desenvolvedores de WebApps em centenas de jovens empresas ponto.com alegaram que um novo paradigma para desenvolvimento de software havia surgido, que as velhas regras não mais se aplicavam, que o prazo de colocação no mercado atropelava todas as outras preocupações. Eles riram da noção de que formulação e planejamento cuidadosos deveriam ocorrer antes do começo da construção. E quem poderia discutir? Havia dinheiro por toda a parte, jovens de 24 anos tornaram-se multimilionários (pelo menos no papel) — talvez as coisas realmente tenham mudado. E depois a base ruiu.

Tornou-se penosamente aparente quando o vigésimo primeiro século começou que uma filosofia "construa que eles virão" simplesmente não funciona, que a formulação do problema é essencial para garantir que uma WebApp é realmente necessária e que planejamento vale o esforço, mesmo quando os cronogramas de desenvolvimento são apertados. Constantine e Lockwood [CON02] observam essa situação quando escrevem:

> Apesar das declarações sem fôlego de que a Web representa um novo paradigma definido por novas regras, os desenvolvedores profissionais estão percebendo que as lições aprendidas nos dias do desenvolvimento de software antes da Internet ainda se aplicam. Páginas Web são interfaces com o usuário. Programação HTML é programação, e aplicações implantadas por navegador são sistemas de software que podem se beneficiar dos princípios básicos de engenharia de software.

Entre os princípios mais fundamentais da engenharia de software destaca-se: *Entenda o problema antes de começar a resolvê-lo, e certifique-se de que a solução que você venha a conceber seja o que as pessoas realmente desejam*. Essa é a base da formulação, a primeira atividade importante da engenharia da Web. Outro princípio fundamental de engenharia de software é: *Planeje o trabalho antes de começar a realizá-lo*. Essa é a filosofia subjacente ao planejamento de projeto.

Panorama

O que é? Iniciar é sempre difícil. De um lado, há uma tendência de adiar, esperar até que todo *t* esteja cruzado e todo *i* pingado antes que o trabalho comece. De outro, há um desejo de ir direto, iniciar a construção antes mesmo que você realmente saiba o que precisa ser feito. Ambas as abordagens são inadequadas, e é por isso que as primeiras duas atividades de arcabouço de engenharia Web enfatizam a formulação e o planejamento. Formulação avalia a necessidade subjacente da WebApp, as características globais e as funções que o usuário deseja, e o escopo do esforço de desenvolvimento. Planejamento trata das coisas que devem ser definidas para estabelecer um fluxo de trabalho e um cronograma, e para monitorar o trabalho à medida que o projeto prossegue.

Quem faz? Engenheiros da Web, seus gerentes e interessados não técnicos, todos participam da formulação e do planejamento.

Por que é importante? É difícil viajar para um lugar que você nunca visitou sem direções ou mapas. Você eventualmente pode chegar (ou você pode não chegar), mas a viagem é com certeza frustrante e desnecessariamente longa. Formulação e planejamento fornecem um mapa para a equipe de engenharia da Web.

Quais são os passos? A formulação começa com a comunicação com os clientes (interessados) a qual trata das razões da WebApp — qual é a necessidade do negócio; quais usuários finais se deseja atingir; que características e

Document shared on www.docsity.com

funções são desejadas; a quais sistemas existentes e bancos de dados se deve ter acesso; o conceito é exeqüível? como o sucesso será medido? Planejamento estabelece um plano de trabalho, desenvolve estimativas para avaliar a possibilidade das datas de entrega desejadas, considera os riscos, define um cronograma e estabelece mecanismos de monitoramento e controle.

Qual é o produto do trabalho? Como o trabalho de engenharia da Web freqüentemente adota uma filosofia ágil, os produtos de trabalho para formulação e planejamento são usualmente simples — mas existem, e devem ser registrados por escrito. A informação obtida durante a formulação é registrada em um documento escrito que serve como base para o planejamento e a modelagem de análise. O plano de projeto estabelece o cronograma do projeto e apresenta toda outra informação que seja necessário comunicar aos membros da equipe de engenharia da Web e aos membros externos.

Como tenho certeza de que fiz corretamente? Desenvolver detalhes suficientes para estabelecer um roteiro sólido, mas não em excesso fazendo-o ficar amarrado. Informação de formulação e planejamento deve ser revisada com os interessados para garantir que inconsistências e omissões sejam identificadas logo.

17.1 Formulação de Sistemas Baseados na Web

A *formulação* de sistemas e aplicações baseados na Web representa uma seqüência de ações de engenharia da Web que começa com a identificação das necessidades do negócio, avança para uma descrição dos objetivos da WebApp, define as principais características e funções e realiza a coleta de requisitos que levam ao desenvolvimento de um modelo de análise. A formulação permite aos interessados e à equipe de engenharia da Web estabelecerem um conjunto comum de metas e objetivos para a construção da WebApp. Também identifica o escopo do esforço de desenvolvimento e fornece meios para determinar um resultado bem-sucedido. A análise — uma atividade técnica que é a continuação da formulação — identifica os requisitos de dados funcionais e comportamentais da WebApp.

PONTO CHAVE

A formulação enfoca o "quadro geral" sobre as necessidades e objetivos do negócio e informações relacionadas.

Antes de considerarmos a formulação em mais detalhes, é razoável perguntar onde a formulação pára e a análise de requisitos começa. Não há resposta fácil para essa questão. A formulação focaliza um "quadro geral" — das necessidades e objetivos de negócio e informação relacionada. No entanto, é virtualmente impossível manter esse nível de abstração. Os interessados e engenheiros da Web querem definir o conteúdo desejado, discutir a funcionalidade específica, enumerar as características específicas e identificar o modo pelo qual usuários finais vão interagir com a WebApp. Isso é a formulação ou coleta de requisitos? A resposta é: as duas coisas.

17.1.1 Questões de Formulação

Powell [POW98] sugere um conjunto de questões que devem ser formuladas e respondidas no início do passo de formulação:

- Qual é a principal motivação (necessidade de negócio) da WebApp?
- Quais são os objetivos que a WebApp deve preencher?
- Quem vai usar a WebApp?

A resposta a cada uma dessas simples questões deve ser dada tão sucintamente quanto possível. Por exemplo, considere que o fabricante do *CasaSegura*[1] tenha decidido estabelecer um site de comércio eletrônico na Web para vender seus produtos diretamente aos consumidores. Uma declaração descrevendo a motivação da WebApp poderia ser:

> CasaSeguraGarantida.com permitirá aos consumidores configurar e adquirir todos os componentes necessários para instalar um sistema de gestão residencial/comercial.

É importante notar que nessa declaração nenhum detalhe é fornecido. O objetivo é limitar a finalidade global da WebApp e colocá-la em um contexto comercial legítimo.

1 O produto *CasaSegura* tem sido usado como exemplo ao longo das Partes 1 e 2 deste livro.

AVISO

À medida que você começa a formular o problema, tente descrever a WebApp que pretende construir em uma única sentença. Se não puder, você não entendeu as metas globais do trabalho.

Depois de discussões com vários interessados, é formulada a resposta à segunda questão:

> CasaSeguraGarantida.com nos permitirá vender diretamente aos consumidores, eliminando conseqüentemente os custos de intermediários e aumentando nossas margens de lucro. Permitirá também aumentar as vendas em 25%, projetados com base nas atuais vendas anuais, e permitirá a penetração em regiões geográficas nas quais atualmente não temos pontos-de-venda.

Finalmente, a empresa define a demografia da WebApp: "Os usuários previstos de CasaSegura Garantida.com são proprietários de residências e donos de pequenos negócios".

As respostas fixadas anteriormente implicam metas específicas para o site CasaSeguraGarantida.com. Em geral, duas categorias de metas [GNA99] são identificadas:

- *Metas informacionais* — indicam a intenção de fornecer conteúdo específico e/ou informação ao usuário final.
- *Metas aplicativas* — indicam a capacidade de realizar alguma tarefa dentro da WebApp.

No conteúdo da WebApp CasaSeguraGarantida.com, uma meta informacional poderia ser:

> O site fornecerá especificações detalhadas do produto aos usuários, incluindo descrições técnicas, instruções de instalação e informações de preço.

O exame das respostas às perguntas propostas poderia levar ao enunciado da seguinte meta aplicativa:

> CasaSeguraGarantida.com consultará o usuário sobre o que (por exemplo, casa, escritório, espaço de varejo) deve ser protegido e fazer recomendações sob medida a respeito do produto e da configuração a ser usada.

Uma vez identificadas as metas informacional e aplicativa, é desenvolvido um perfil de usuário. O perfil de usuário capta "características relevantes relacionadas aos usuários em potencial, incluindo sua experiência (*background*), conhecimento, preferências e até mais" [GNA99]. No caso do CasaSeguraGarantida.com, um perfil de usuário identificaria as características de um comprador típico de sistemas de segurança (essa informação seria suprida pelo departamento de marketing).

> "Se você estiver fazendo seriamente [WebApps], sua filosofia é provavelmente 'preparar, fogo, mirar'. Se você estiver seriamente decidido a fazê-las funcionar, deveria ser 'preparar, mirar, fogo'."
>
> **Autor desconhecido**

Uma vez desenvolvidas as metas e perfis de usuários, a atividade de formulação focaliza uma declaração de escopo para a WebApp. Em muitos casos, as metas já desenvolvidas são integradas na declaração de escopo. Além disso, no entanto, é útil indicar o grau de integração esperado para a WebApp. Isto é, freqüentemente é necessário integrar sistemas de informação existentes (por exemplo, uma aplicação existente de banco de dados) com uma fachada baseada na Web. Tópicos de conectividade são considerados nesse estágio.

17.1.2 Coleta de Requisitos para WebApps

Métodos de coleta de requisitos foram discutidos no Capítulo 7. Embora a atividade de coleta de requisitos para engenharia da Web possa ser abreviada, os objetivos da coleta global de requisitos propostos pela engenharia de software permanecem inalterados. Adaptados para WebApp esses objetivos tornam-se:

- Identificar os requisitos de conteúdo.
- Identificar os requisitos funcionais.
- Definir os cenários de interação para as diferentes classes de usuários.

Document shared on www.docsity.com

? Que passos de coleta de requisitos são usados para WebApp?

Os seguintes passos para obtenção de requisitos são conduzidos para atingir esses objetivos:

1. Peça aos interessados para definir as categorias de usuários e desenvolver descrições de cada categoria.
2. Comunique-se com os interessados para definir os requisitos básicos da WebApp.
3. Analise a informação coletada e use a informação para conferir com os interessados.
4. Defina os casos de uso (Capítulo 8) que descrevem os cenários de interação para cada classe do usuário.

AVISO

Entender a experiência, motivação e objetivos do usuário é crítico em todo trabalho de engenharia de software. Se você construir uma WebApp sem conhecer essas coisas, seu trabalho estará em risco.

Definição de categorias de usuário. Pode-se argumentar que a complexidade da WebApp é diretamente proporcional ao número de categorias de usuário do sistema. Para definir uma categoria de usuário um conjunto de questões fundamentais deve ser tratado.

- *Qual é o objetivo global do usuário quando usa a WebApp?* Por exemplo, um usuário do site de e-commerce CasaSeguraGarantida.com pode estar interessado em obter informação sobre os produtos de gestão de residência. Outro usuário pode desejar fazer uma comparação de preço. Um terceiro usuário deseja comprar o produto *CasaSegura*. Cada um representa uma diferente classe ou categoria de usuário; cada um tem diferentes necessidades e navegará pela WebApp diferentemente. Um quarto usuário já possui o *CasaSegura* e está procurando suporte técnico ou deseja comprar sensores ou acessórios adicionais.
- *Qual o conhecimento e sofisticação do usuário em relação ao conteúdo e à funcionalidade da WebApp?* Se um usuário tem um conhecimento técnico e sofisticação significativa, conteúdo ou funcionalidade elementar fornecerá pouco benefício. Alternativamente, um novato requer conteúdo e funcionalidade elementar e ficaria confuso se isso fosse omitido.
- *Como o usuário chegará à WebApp?* A chegada ocorrerá por meio de uma referência de outro site da Web (com conteúdo e funcionalidade parecidos com os da WebApp), ou de um modo mais controlado?
- *Que características genéricas da WebApp o usuário gosta/não gosta?* Diferentes tipos de usuários podem ter gostos distintos e previsíveis. É válido tentar determinar se eles gostam ou não. Em muitas situações, a resposta a essa questão pode ser determinada perguntando quais as suas WebApps favoritas e as menos favoritas.

CASASEGURA

Coleta de Requisitos para as WebApps

A cena: Escritório de Doug Miller.

Os personagens: Doug Miller, gerente do grupo de engenharia de software; Vinod Raman, membro da equipe de engenharia de software do *CasaSegura* e posteriormente três pessoas de marketing.

A conversa:

Doug: A gerência decidiu que iremos construir um site de e-commerce para vender o *CasaSegura*.

Vinod: Pode parar, Doug! Não temos tempo para fazer isso... estamos atolados com o trabalho do produto de software.

Doug: Eu sei, eu sei... vamos terceirizar o desenvolvimento para uma empresa que é especialista em construção de sites de e-commerce. Eles nos contaram que vão colocar o sistema em pé e rodando em menos de um mês... montes de componentes reusáveis.

Vinod: Hmmm. Está certo... então por que estou aqui?

Doug: Para apressar as coisas — eles precisam de nós para fazer a coleta de requisitos para o site. Eu gostaria que você se encontrasse com os vários interessados para obter algumas informações sobre os requisitos básicos.

Vinod (exasperado): Doug... você não está me ouvindo... nós estamos perdidos em termos de tempo e isso...

Doug (interrompendo): Dê-lhe exatamente apenas um dia do seu tempo, Vinod. Encontre-se com as pessoas de marketing e faça com que eles especifiquem o conteúdo básico, função, você sabe, o usual.

Vinod (resignado): Está certo, eu darei um telefonema para eles e agendarei alguma coisa para amanhã, mas você não está tornando a minha vida mais fácil.

Doug (rindo): É por isso que você ganha tanto dinheiro.

Vinod: Certo.

(Vinod encontra-se com as três pessoas de marketing no dia seguinte.)

Vinod: Você estava me falando sobre os objetivos e conhecimentos do usuário.

Pessoa nº 1 de Marketing: Como eu disse, queremos que o usuário seja capaz de personalizar todo o sistema *CasaSegura*, você sabe, escolher os sensores, painéis de controle, características e funções, depois obter uma "lista de materiais" gerada automaticamente, obter preço e depois comprar o sistema pelo site.

Pessoa nº 2 de Marketing: Consideramos que o usuário é um proprietário — não técnico —, assim precisamos guiá-lo ou guiá-la pelo processo passo a passo.

Pessoa nº 3 de Marketing: Eu não sou técnico, mas estou preocupado com o material especializado que precisamos fazer, além das coisas básicas de e-commerce.

Vinod (dirigindo-se ao nº 3): Isso significa?

Pessoa nº 3 de Marketing: A parte difícil vai ser guiar o usuário pelo "processo de personalização", de um modo que seja simples e completo. O material atual de e-commerce é bastante direto.

Pessoa nº 1 de Marketing: Temos de fornecer um número 0800 para o pessoal que não quer fazer a personalização por si próprio.

Pessoa nº 3 de Marketing: Eu concordo.

Vinod: Certo, vamos ter de conversar sobre exatamente como você gostaria de fazer a personalização do produto como uma atividade pré-venda, mas vamos aguardar um instante sobre isso. Eu tenho algumas outras questões fundamentais.

Vinod (olhando para a Pessoa nº 2 de Marketing): Você disse que queria guiar os usuários durante o processo. Alguma abordagem especial?

Pessoa nº 2 de Marketing: Eu gostaria de ver um processo passo a passo, com respostas às questões básicas de requisitos em formulários do tipo preencher-os-espaços, menu *pull down*, essa espécie de coisa. Cada passo é uma janela, e os dados de cada janela são validados antes de ir para o passo seguinte.

Vinod: Você verificou isso com usuários representativos?

Pessoa nº 2 de Marketing: Não, mas eu farei isso.

Vinod: Mais uma coisa... como um usuário encontra o nosso site?

Pessoa nº 1 de Marketing: Estamos trabalhando em uma campanha de publicidade que vai colocar www.CasaSeguraGarantida.com em anúncios de revista, mala direta dirigida, anúncios sensíveis ao contexto que aparecem em motores de busca, e talvez até alguns anúncios de rádio e TV.

Vinod: O que eu quero dizer é... eles vão sempre entrar pela página principal?

Pessoa nº 3 de Marketing: Isso é o que gostaríamos.

Vinod: Certo, agora nós temos de trabalhar. Vamos explorar os detalhes de como você quer personalizar o sistema on-line.

Usando as respostas a essas questões, o menor conjunto razoável de classes de usuários deve ser definido. À medida que a coleta de requisitos prossegue, cada classe de usuário definida precisa ser consultada para fornecer informação.

? Que mecanismos de comunicação podem ser usados em trabalhos da WebE?

Comunicação com os interessados e usuários finais. A maioria das WebApps tem uma vasta população de usuários finais. Embora a criação das categorias ou classes de usuário faça uma avaliação dos requisitos do usuário mais gerenciável, não é aconselhável usar a informação coletada de uma única ou de duas pessoas como a base para formulação ou análise. Mais pessoas (e mais opiniões/pontos de vista) devem ser consideradas.

Comunicação pode ser conseguida usando um ou mais dos seguintes mecanismos [FUC02a]:

- *Grupos focais tradicionais* — um moderador treinado se reúne com um pequeno grupo (usualmente menos que 10 pessoas) de representantes dos usuários finais (ou interessados internos desempenhando o papel de usuários finais). O objetivo é discutir a WebApp a ser desenvolvida e, fora da discussão, melhorar o entendimento dos requisitos do sistema.
- *Grupos focais eletrônicos* — um moderador de discussão eletrônica conduz um grupo de representantes de usuários finais e interessados. O número de pessoas que participa pode ser maior. Como todos os usuários podem participar ao mesmo tempo, mais informação pode ser coletada em período de tempo mais curto. Como toda a discussão é baseada em texto, um registro contemporâneo é automático.

Document shared on www.docsity.com

- *Levantamentos iterativos* — uma série de levantamentos resumidos, voltada para representantes de usuários e solicitando respostas a questões específicas sobre a WebApp, é conduzida por um site ou por e-mail. Respostas são analisadas e usadas para refinar o próximo levantamento.
- *Levantamentos exploratórios* — um levantamento baseado na Web que está ligado a uma ou mais WebApps que tem usuários similares àqueles que vão usar a WebApp a ser desenvolvida. Usuários se conectam ao levantamento e respondem a uma série de questões (usualmente recebendo alguma recompensa pela participação).
- *Construção de cenário* — usuários selecionados são solicitados a criar casos de uso informais que descrevem interações específicas com a WebApp.

AVISO

Uma evolução de objetos de conteúdo e operações pode ser adiada até que a modelagem de análise comece. Nesse ponto é mais importante coletar a informação do que avaliá-la.

Análise da informação coletada. À medida que a informação é coletada, ela é categorizada por classe de usuários e tipo de transação, e depois avaliada quanto à relevância. O objetivo é desenvolver listas de objetos de conteúdo, operações que são aplicadas aos objetos de conteúdo dentro de uma transação de usuário específica, funções (por exemplo, informacional, computacional, lógica e orientada a ajuda) que a WebApp fornece para os usuários finais e outros requisitos não funcionais que são observados durante as atividades de comunicação.

Fuccella e Pizzolato [FUC02b] sugerem um método simples (baixa tecnologia) para entender como o conteúdo e a funcionalidade devem ser organizados. Uma pilha de "cartões" é criada para objetos de conteúdo, operações aplicadas aos objetos de conteúdo, funções da WebApp e outros requisitos não funcionais. Os cartões são embaralhados em ordem aleatória e depois distribuídos aos representantes de cada categoria de usuário. Os usuários são solicitados a arrumar os cartões em agrupamentos que reflitam como eles gostariam que o conteúdo e a funcionalidade fossem organizados na WebApp. Os usuários são então solicitados a descrever cada agrupamento e as razões pelas quais ele é importante para eles. Após cada usuário realizar esse exercício, a equipe de engenharia da Web procura agrupamentos comuns entre diferentes classes de usuário e outros agrupamentos que são únicos para uma classe específica de usuário.

A equipe de WebE desenvolve uma lista de rótulos que pode ser usada para apontar para informação dentro de cada um dos agrupamentos derivados usando as pilhas de cartões. Diferentes pilhas de cartões são entregues aos representantes de usuários e eles são solicitados a alocar conteúdo e funcionalidade a cada um dos rótulos. O objetivo aqui é determinar quando os rótulos (referências na WepApp real) implicam adequadamente acesso ao conteúdo e às funções que os usuários esperam encontrar por trás do rótulo. Esse passo é aplicado iterativamente até que um consenso seja atingido.

AVISO

Os casos de uso foram discutidos em detalhes na Parte 2 deste livro. Embora muitos defendam o desenvolvimento de casos de uso extensos, mesmo uma narrativa informal traz algum benefício. Convença os usuários a escrever casos de uso.

> "Se você não pode descrever o que está fazendo como um processo, você não sabe o que está fazendo."
>
> W. E. Deming

Desenvolvimento de casos de uso. Casos de uso[2] descrevem como uma categoria de usuário específica (chamada de *ator*) vai interagir com a WebApp para realizar uma ação específica. A ação pode ser tão simples quanto adquirir um conteúdo definido, ou tão complexa quanto conduzir a análise detalhada, dirigida pelo usuário, de registros selecionados mantidos em um banco de dados *on-line*. Os casos de uso descrevem a interação do ponto de vista do usuário.

Embora o seu desenvolvimento e análise levem tempo, os casos de uso (1) ajudam o desenvolvedor a entender como os usuários percebem sua interação com a WebApp; (2) fornecem o detalhe necessário para criar um modelo de análise efetivo; (3) ajudam a compartimentalizar o trabalho de WebE; e (4) fornecem diretrizes importantes para aqueles que precisam testar a WebApp.

2 Técnicas de desenvolvimento de casos de uso foram apresentadas em detalhes nos Capítulos 7 e 8.

CONJUNTO DE TAREFAS

Comunicação com o Cliente (Análise/Formulação)

1. *Identifique os interessados no negócio.* Exatamente, quem é o "cliente" da WebApp? Que pessoas de negócio podem servir como especialistas e representantes dos usuários finais? Quem vai servir como membro ativo da equipe?
2. *Formule o contexto do negócio.* Como a WebApp se encaixa em uma estratégia de negócio mais ampla?
3. *Defina as metas e os objetivos-chave de negócio para a WebApp.* Como é medido o sucesso da WebApp tanto em termos qualitativos quanto quantitativos?
4. *Defina as metas informacionais e aplicativas.* Que classes de conteúdo devem ser fornecidas aos usuários finais? Que funções/tarefas devem ser realizadas quando se usa a WebApp?
5. *Identifique o problema.* Que problema específico a WebApp resolve?
6. *Colete requisitos.* Que tarefas de usuário serão realizadas usando a WebApp? Que conteúdo deve ser desenvolvido? Que metáfora de interação será usada? Que funções computacionais serão fornecidas pela WebApp? Como a WebApp será configurada para a utilização em rede? Que esquema navegacional é desejado?

AVISO

Para pequenos projetos, um simples "banco de dados de requisitos" pode ser mantido (usando uma planilha) no lugar de modelos UML. Isso permite que todos os membros da equipe de WebE rastreiem os requisitos até o conteúdo e função entregues e controlem melhor a inevitável cadeia de modificações que ocorrerá.

17.1.3 A Ponte para Modelagem de Análise

Como observamos anteriormente neste capítulo, as atividades conduzidas por uma equipe de engenharia da Web, da formulação à modelagem de análise, representam uma continuidade. Em essência, o nível de abstração considerado durante os estágios iniciais da formulação é estratégico em termos de negócio. No entanto, à medida que a formulação prossegue, detalhes táticos são discutidos e requisitos específicos da Web são tratados. Finalmente, esses requisitos são modelados (usando casos de uso e notação UML).

Os conceitos e princípios discutidos para análise de requisitos de software (Capítulos 7 e 8) aplicam-se sem revisão à atividade de análise da WebE. Durante a análise, o escopo definido na atividade de formulação é refinado para criar um modelo completo de análise para a WebApp. Quatro diferentes tipos de análise são conduzidos durante a WebE: análise de conteúdo, análise de interação, análise de função e análise de configuração. Cada uma dessas tarefas de análise e as técnicas de modelagem a elas associadas são discutidas no Capítulo 18.

> "Deixando de preparar, você está se preparando para falhar."
>
> Benjamin Franklin

17.2 PLANEJAMENTO DE PROJETOS DE ENGENHARIA WEB

Veja na Web

Ferramentas que apóiam gestão de e-projeto podem ser encontradas em www.eproject.com

Dada a natureza imediatista das WebApps, é razoável perguntar: precisamos, realmente, gastar tempo planejando e gerenciando um esforço de WebApp? Não deveríamos deixar a WebApp evoluir naturalmente, com pouco ou nenhum gerenciamento explícito? Mais do que uns poucos desenvolvedores Web optariam por pouco ou nenhum gerenciamento, mas isso não faz com que estejam corretos!

A Figura 17.1 apresenta uma tabela adaptada de Kulik e Samuelsen [KUL00] que indica como "e-projetos" (*e-projects*, em inglês, é o termo usado por esses autores para projetos de WebApp) se comparam com projetos de software tradicional. Com referência à figura, projetos de software tradicional e e-projetos importantes têm substanciais similaridades. Como a gestão de projeto é indicada para projetos tradicionais, seria razoável argumentar que também seria indicada para e-projetos importantes. E-projetos pequenos têm características especiais que os tornam diferentes de projetos tradicionais. No entanto, mesmo no caso de pequenos e-projetos, planejamento deve ocorrer, riscos precisam ser considerados, um cronograma deve ser estabelecido e controles precisam ser definidos de modo que sejam evitadas confusão, frustração e falhas.

Document shared on www.docsity.com

FIGURA 17.1 Diferenças entre os projetos tradicionais e e-projetos (adaptado de KUL00)

	Projetos Tradicionais	Pequenos e-projetos	e-projetos Importantes
Coleta de requisitos	Rigorosa	Limitada	Rigorosa
Especificação técnica	Robusta: modelos, especificação	Visão geral descritiva	Robusta: modelos UML, especificação
Duração da fase de projeto	Medida em meses ou anos	Medida em dias, semanas ou meses	Medida em meses ou anos
Teste e QA	Focada na obtenção de qualidade do produto	Focada no controle de riscos	SQA como descrito no Capítulo 26
Gestão de riscos	Explícita	Inerente	Explícita
Meia-vida dos produtos entregues	18 meses ou mais	3 a 6 meses ou menos	6 a 12 meses ou menos
Processo de entrega	Rigoroso	Expedito	Rigoroso
Realimentação do cliente após a entrega	Requer esforço proativo	Automaticamente obtido da interação com o usuário	Obtido tanto automaticamente quanto por realimentação solicitada

17.3 A Equipe de Engenharia da Web

Uma equipe de engenharia da Web bem-sucedida mistura uma grande variedade de talentos que precisam trabalhar como uma equipe em um ambiente de projeto de alta pressão. Os prazos são curtos, as modificações são ininterruptas e a tecnologia mantém-se em evolução. Criar uma equipe que se aglutina (veja Capítulo 21) não é coisa simples.

> "No mundo de hoje, centrado em rede e habilitado na Web, uma pessoa precisa saber muito sobre muita coisa."
>
> Scott Tilley e Shihong Huang

17.3.1 Os Personagens

A criação de uma aplicação bem-sucedida na Web exige uma ampla gama de habilidades. Tilley e Huang [TIL99] tratam desse tópico quando declaram: "Há tantos aspectos diferentes relativos a uma aplicação de software [da Web] que o (re)aparecimento de um especialista, alguém que domina várias áreas, se torna necessário... ". Enquanto os autores forem absolutamente corretos, os *experts* são relativamente desnecessários; e dada a demanda associada com os principais projetos de desenvolvimento de WebApps, o conjunto das diversas habilidades exigidas pode ser mais bem distribuído por uma equipe de engenharia da Web.

As equipes de engenharia da Web podem ser organizadas de modo bastante semelhante às equipes de software discutidas no Capítulo 21. No entanto, os personagens e seus papéis são freqüentemente bem diferentes. Entre as várias habilidades que precisam ser distribuídas pelos membros de uma equipe de WebE estão a engenharia de software baseada em componentes, o projeto de rede, arquitetural e navegacional, as normas/linguagens da Internet, o projeto de interface homem/máquina, o projeto gráfico, o leiaute de conteúdo e o teste de WebApp.

Os seguintes papéis[3] devem ser distribuídos entre os membros da equipe de WebE:

3 Esses papéis foram adaptados de Hansen e seus colegas [HAN99].

? Que papéis as pessoas desempenham em uma equipe de WebE?

Desenvolvedores/provedores de conteúdo. Como as WebApps são inerentemente guiadas por conteúdo, o papel de um membro da equipe de WebE deve se voltar para a geração ou coleta de conteúdo. Lembrando que o conteúdo abrange uma ampla gama de objetos de dados, os desenvolvedores/provedores de conteúdo precisam ser oriundos de diversas áreas (não de software).

Editor da Web. O conteúdo diversificado, gerado pelos desenvolvedores e provedores de conteúdo, precisa ser organizado para inclusão na WebApp. Além disso, alguém precisa agir como elemento de ligação entre a equipe técnica que constrói a WebApp e os desenvolvedores/provedores de conteúdo não-técnico. Esse papel é preenchido pelo *editor* da Web, que precisa entender tanto de conteúdo quanto de tecnologia da WebApp.

Engenheiro da Web. Um engenheiro da Web se vê envolvido com uma grande variedade de atividades durante o desenvolvimento de uma WebApp, inclusive a elicitação de requisitos, modelagem de análise, projeto arquitetural, navegacional e de interface; implementação da WebApp e teste. O engenheiro da Web deve também ter um sólido conhecimento de tecnologias de componentes, arquiteturas cliente/servidor, HTML/XML e tecnologias de bancos de dados, bem como conhecimento funcional de conceitos de multimídia, plataformas de hardware/software, segurança de rede e tópicos de suporte a sites Web.

Especialistas no domínio do negócio. Um especialista no domínio do negócio deve ser capaz de responder a todas as questões relativas às metas, aos objetivos e aos requisitos de negócio associados a WebApp.

Especialista de suporte. Esse papel é atribuído a uma pessoa (pessoas) responsável por dar continuidade ao suporte da WebApp. Como as WebApps evoluem continuamente, o especialista de suporte é responsável por correções, adaptações e aperfeiçoamentos do site, inclusive atualizações de conteúdo, implementação de novos procedimentos e formulários, e modificações no padrão de navegação.

Administrador. Freqüentemente chamado de "Web Master", essa pessoa é responsável pela operação do dia-a-dia da WebApp, inclusive: desenvolvimento e implementação de políticas de operação da WebApp, estabelecimento de procedimentos de suporte e realimentação, implementação de segurança e direitos de acesso, medição e análise do tráfego no site, coordenação dos procedimentos de controle de modificações (Capítulo 27) e coordenação com especialistas de suporte. O administrador pode se envolver também com as atividades técnicas realizadas por engenheiros da Web e especialistas de suporte.

17.3.2 Construção da Equipe

No Capítulo 21, diretrizes para construir com sucesso equipes de engenharia de software são discutidas com algum detalhe. Mas, essas diretrizes aplicam-se ao mundo cheio de pressões dos projetos de WebApp? A resposta é sim.

No seu livro, que figurou entre os mais vendidos sobre a antiga indústria de computador, Tracy Kidder [KID00] conta a história de uma heróica tentativa de uma empresa de computador de construir um computador para vencer o desafio de um novo produto construído por uma grande concorrente[4].

A história é uma metáfora para equipe de trabalho, liderança e a pressão crescente que todos os tecnologistas encontram quando projetos críticos não evoluem tão suavemente quanto o planejado.

Um resumo do livro de Kidder dificilmente lhe faz justiça, mas esses pontos-chave [PIC01] têm particular relevância quando uma organização constrói uma equipe de engenharia da Web:

Deve ser estabelecido um conjunto de diretrizes para a equipe. Essas diretrizes englobam o que é esperado de cada pessoa, como os problemas devem ser tratados e que mecanismos existem para melhorar a efetividade da equipe à medida que o projeto prossegue.

Forte liderança é uma necessidade. O líder da equipe precisa conduzir pelo exemplo e pelo contato. Precisa exibir um nível de entusiasmo que contagie os outros membros da equipe para "associá-los" psicologicamente ao trabalho que eles enfrentam.

AVISO

Essas características são típicas de equipes colaborativas, auto-organizadas, que têm adotado uma filosofia ágil (Capítulo 4). Quanto melhor a sua equipe, melhor o produto de software que você produz.

4 *The Soul of a New Machine* de Kidder, originalmente publicado em 1981, é leitura altamente recomendada para quem pretende fazer carreira em computação e para todos que já tenham feito.

Document shared on www.docsity.com

Respeitar os talentos individuais é crítico. Nem todos são bons em tudo. As melhores equipes fazem uso das vantagens individuais. Os melhores líderes de equipe dão liberdade individual para ir adiante com uma boa idéia.

Todos os membros da equipe devem estar comprometidos. O protagonista principal no livro de Kidder chama isso de "assinar".

É fácil dar o início, mas é muito difícil manter o ritmo. As melhores equipes nunca deixam um problema "insuperável" detê-los. Os membros da equipe desenvolvem uma solução "suficientemente boa" e prosseguem, esperando que o ritmo de progresso adiante possa levar a uma solução ainda melhor a longo prazo.

17.4 Tópicos de Gestão de Projeto para Engenharia Web

Depois que a formulação ocorrer e os requisitos básicos da WebApp tiverem sido identificados, é preciso escolher uma das duas opções da engenharia da Web (1) a WebApp é *terceirizada* — engenharia da Web é realizada por uma empresa terceirizada que tem a experiência, o talento e os recursos que faltam dentro do negócio; ou (2) a WebApp é desenvolvida *internamente* usando engenheiros da Web que são empregados no negócio. Uma terceira alternativa é fazer algum trabalho de engenharia da Web internamente e terceirizar outros trabalhos.

> "Como Thomas Hobbs observou no décimo sétimo século, a vida sob o governo da multidão é solitária, pobre, desagradável, estúpida e curta. A vida em um projeto de software maldirigido é solitária, pobre, desagradável, estúpida e dificilmente curta o suficiente."
>
> **Steve McConnell**

O trabalho a ser realizado permanece o mesmo independentemente se uma WebApp for terceirizada, desenvolvida internamente, ou distribuída entre um fornecedor externo e a equipe da casa. Mas, os requisitos de comunicação, a distribuição das atividades técnicas, o grau de interação entre os interessados e desenvolvedores, e uma grande quantidade de outros importantes tópicos críticos fazem a diferença.

A Figura 17.2 ilustra a diferença organizacional entre o desenvolvimento terceirizado e interno de WebApps. O desenvolvimento interno (Figura 17.2a) integra (o círculo sombreado demonstra a integração) todos os membros da equipe de engenharia Web diretamente. Comunicação ocorre usando os caminhos organizacionais normais. Para a terceirização (Figura 17.2b), é tanto não prático quanto desaconselhável ter cada componente interno (por exemplo, desenvolvedores de conteúdo, interessados, engenheiros internos da Web) se comunicando diretamente com o fornecedor terceirizado sem algum elemento de ligação com o fornecedor para coordenar e controlar a comunicação. Nas próximas seções, examinaremos o planejamento para o desenvolvimento terceirizado e interno em mais detalhes.

AVISO

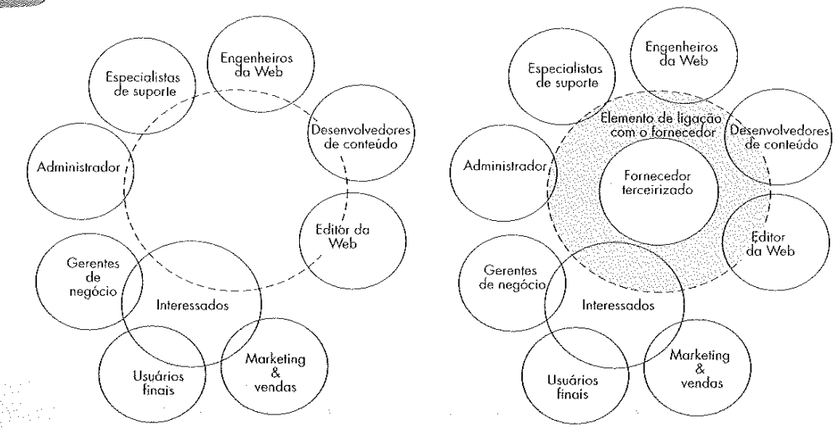
Não imagine que porque você terceirizou uma WebApp, suas responsabilidades são menores. De fato, é provável que mais supervisão e gerenciamento, e não menos, sejam necessários.

17.4.1 Planejamento de Terceirização de WebApp

Uma porcentagem substancial de WebApps é terceirizada para fornecedores que (deliberdamente) se especializam no desenvolvimento de sistemas e aplicações[5] baseados na Web. Em tais casos, uma empresa (o cliente) pede uma cotação para o desenvolvimento da WebApp a dois ou mais fornecedores, avalia as cotações concorrentes e depois seleciona um fornecedor para fazer o trabalho. Mas o que a organização contratante está procurando? Como é aferida a competência do fornecedor de WebApp? Como saber se um preço cotado é razoável? Que grau de planejamento, cronogramação e avaliação de risco pode ser esperado à medida que uma organização (e seu terceirizado) se lança em um importante esforço de desenvolvimento de WebApp?

5 Apesar da dificuldade de encontrar dados confiáveis na indústria, pode-se dizer, com segurança, que essa porcentagem é consideravelmente maior do que a encontrada no trabalho de software convencional. Discussão adicional sobre terceirização pode ser encontrada no Capítulo 23.

FIGURA 17.2 A diferença organizacional entre o desenvolvimento terceirizado e o interno



(a) desenvolvimento interno (b) desenvolvimento terceirizado

> "Muitas empresas da lista das 500 maiores da revista *Fortune* descobriram o software como modelo de serviço [terceirização] e estão empregando modelos similares internamente ou externamente."
>
> **Nick Evans**

Essas questões não são sempre fáceis de responder, mas vale a pena considerar algumas diretrizes.

Inicie o projeto. Se a terceirização é a estratégia escolhida para o desenvolvimento da WebApp, a empresa precisa realizar algumas tarefas antes de procurar um fornecedor terceirizado para fazer o trabalho:

1. *Muitas das atividades de análise discutidas na Seção 17.1.3 (e no Capítulo 18) devem ser realizadas internamente.* A audiência da WebApp é identificada; interessados internos na WebApp são listados; os objetivos globais da WebApp são definidos e revistos; a informação e os serviços a serem fornecidos pela WebApp são especificados; sites concorrentes são registrados; "medidas" qualitativas e quantitativas de uma WebApp bem-sucedida são definidas. Essa informação deve ser documentada em uma especificação de produto que é fornecida para o fornecedor terceirizado.
2. *Deve ser desenvolvido internamente um esboço de projeto da WebApp.* Obviamente, um desenvolvedor experiente da Web criará um projeto completo, mas tempo e custo podem ser poupados se a aparência geral da WebApp for identificada para o fornecedor terceirizado (isso sempre pode ser modificado durante os estágios preliminares do projeto). O projeto deve incluir uma indicação do tipo e do volume de conteúdo a ser apresentado pela WebApp, e dos tipos de processamento interativo (por exemplo, formulários, entrada de pedidos) a ser realizado. Essa informação deve ser adicionada à especificação do produto.
3. *Um esboço de cronograma de projeto, que inclua não apenas as datas finais de entrega, mas também marcos intermediários de prazo, deve ser desenvolvido.* Os marcos intermediários de prazo devem estar ligados às versões (incrementos) da WebApp a serem entregues, à medida que a WebApp evolui.
4. *Uma lista de responsabilidades da organização interna e do fornecedor terceirizado é criada.* Em essência, essa tarefa trata de que informação, contatos e outros recursos são necessários por parte de ambas as organizações.

AVISO

Algumas pessoas alegam que o "esboço de projeto" é desnecessário. Considere isso uma "primeira idéia" que o fornecedor terceirizado pode modificar e aperfeiçoar. Pelo menos, você está comunicando suas idéias sobre que aparência o resultado final deve ter.

Document shared on www.docsity.com

5. *Deve ser identificado o grau de supervisão e interação do contratante com o fornecedor.* Isso deve incluir a designação de um elemento de contato com o fornecedor e a identificação das responsabilidades e da autoridade desse contato, a definição dos pontos de revisão de qualidade, à medida que o desenvolvimento prossegue, e as responsabilidades do fornecedor, no que diz respeito à comunicação interorganizacional.

? Que diretrizes devem ser usadas quando são considerados vários fornecedores terceirizados?

Toda a informação desenvolvida durante esses passos deve ser organizada em um pedido de cotação que é transmitido aos candidatos a fornecedor[6].

Selecione candidatos a fornecedor terceirizado. Nos últimos anos, milhares de empresas de "desenvolvimento Web" surgiram para ajudar os negociantes a estabelecerem presença na Web e/ou se engajarem no comércio eletrônico. Muitos tornaram-se adeptos do processo de WebE, mas muitos outros não passam de "curiosos". A fim de selecionar candidatos a desenvolvedor na Web, o contratante deve realizar uma diligência cuidadosa: (1) entrevistar clientes anteriores, para determinar o profissionalismo do fornecedor da Web, sua habilidade de cumprir compromissos de cronograma e custo e sua capacidade efetiva de comunicação; (2) determinar o nome do(s) principal(is) engenheiro(s) da Web do fornecedor quanto a projetos anteriores bem-sucedidos (e depois certificar-se de que essa(s) pessoa(s) é(são) obrigada(s) contratualmente a se envolver(em) no seu projeto); e (3) examinar cuidadosamente amostras do trabalho do fornecedor que sejam semelhantes em aparência (e em área de negócios) à WebApp a ser contratada. Mesmo antes de um pedido de cotação ser efetuado, um encontro pessoal deve proporcionar um conhecimento aprofundado e substancial quanto à "harmonia" entre contratante e fornecedor.

> "Você paga amendoins, você recebe macacos."
>
> George Peppard desempenhando o papel de Col. John "Hannibal" Smith no *The A-Team* (um programa de TV na década de 1980)

Avalie a validade das cotações de preço e a confiabilidade das estimativas. Como existem relativamente poucos dados históricos e o escopo das WebApps é notoriamente duvidoso, a estimativa é um risco inerente. Por essa razão, alguns fornecedores embutem margens de segurança substanciais no custo cotado para um projeto. Isso tanto é compreensível quanto adequado. A questão *não* é se conseguimos o melhor resultado para o nosso dinheiro. Em vez disso, as questões deveriam ser:

- O custo cotado para a WebApp fornece um retorno de investimento direto ou indireto que justifique o projeto?
- O fornecedor que deu a cotação tem o profissionalismo e a experiência de que precisamos?

Se as respostas a essas perguntas forem "sim", a cotação de preço é justa.

Entenda o grau de gestão de projeto que você pode esperar/realizar. A formalidade associada às tarefas de gestão de projeto (realizadas tanto pelo fornecedor quanto pela empresa contratante) é diretamente proporcional ao tamanho, ao custo e à complexidade da WebApp. Para projetos complexos de grande porte, um cronograma detalhado de projeto, que define tarefas de trabalho, pontos de verificação de SQA, produtos de trabalho da engenharia, pontos de revisão do cliente e prazos principais deve ser desenvolvido. O fornecedor e o contratante devem avaliar conjuntamente o risco e desenvolver planos para mitigar, monitorar e gerir aqueles que forem considerados importantes. Mecanismos de garantia de qualidade e controle de modificação devem ser explicitamente definidos por escrito. Métodos para comunicação efetiva entre o contratante e o fornecedor devem ser estabelecidos.

Avalie o cronograma de desenvolvimento. Como os cronogramas de desenvolvimento da WebApp se estendem por um período relativamente curto (freqüentemente menor do que um ou dois meses

6 Nada muda se o trabalho de desenvolvimento da WebApp for conduzido por um grupo interno! O projeto é iniciado do mesmo modo.

PONTO CHAVE

Para gerir o escopo, o trabalho deve ser realizado quando um incremento está congelado. Modificações são adiadas até o próximo incremento da WebApp.

por incremento entregue), o cronograma de desenvolvimento deve ter uma granularidade fina — ou seja, as tarefas de trabalho e os marcos intermediários de prazo devem ser programados em base diária. Essa granularidade fina permite que tanto o contratante quanto o fornecedor reconheçam atrasos de cronograma, antes que ameacem a data final de conclusão.

Gerencie escopo. Como é muito provável que o escopo se modifique à medida que o projeto da WebApp avança, o modelo de processo de WebE deve ser adaptável e incremental. Isso permite que a equipe de desenvolvimento do fornecedor "congele" o escopo de um incremento até que uma versão operacional da WebApp possa ser criada. O incremento seguinte pode tratar das modificações de escopo sugeridas pela revisão do incremento anterior, mas uma vez iniciado o segundo incremento, o escopo é novamente "congelado" temporariamente. Essa abordagem permite à equipe da WebApp trabalhar sem ter de acomodar uma torrente contínua de modificações, mas sim reconhecer a característica de evolução contínua da maioria das WebApps.

As diretrizes sugeridas não pretendem ser um livro de receitas infalível para a produção de WebApp de baixo custo e sem atraso. No entanto, vão ajudar tanto o contratante quanto o fornecedor a iniciarem o trabalho suavemente com um mínimo de mal-entendidos.

CASASEGURA

Preliminares da Terceirização

A cena: Escritório de Doug Miller.

Os personagens: Doug Miller (gerente da equipe de engenharia de software do *CasaSegura*) e Sharon Woods — funcionário da SistemasDeE-Commerce, o fornecedor terceirizado para sites de e-commerce para o *CasaSegura* e gerente da equipe de engenharia da Web que vai fazer o trabalho.

A conversa:

Doug: Bom, finalmente encontramos você, Sharon. Certamente, temos algum trabalho para fazer no próximo mês ou por aí.

Sharon (rindo): Nós temos, mas vocês parecem ter sua parte pronta. Vinod já nos deu um rascunho de especificação do site e também definiu a maioria dos objetos de conteúdo importantes e a funcionalidade do site.

Doug: Bom. Do que mais você precisa?

Sharon: A funcionalidade do e-commerce é fácil. A coisa que me preocupa é a fachada... o trabalho necessário para fazer o usuário personalizar o produto antes da compra.

Doug: Vinod deu a você o procedimento básico, não deu?

Sharon: Sim, mas eu gostaria de validá-lo com alguns usuários reais. Também vamos precisar contatar os seus desenvolvedores de conteúdo para obter as descrições adequadas de cada sensor, quadros, estabelecimento de preço, informação de interface/interconexão, coisas desse tipo.

Doug: Vinod teve tempo de fazer um esboço do roteiro do processo de personalização para você?

Sharon: Ele está trabalhando nisso, enquanto falamos. Disse que tinha que apagar um incêndio no lado do produto. Ele sabe que é crítico... disse que ia me mandar um e-mail amanhã cedo.

Doug: Certo... veja, eu gostaria de estar no ciclo desse projeto. Podemos estabelecer algumas regras básicas para supervisão do seu lado. Eu não quero ficar no seu caminho, mas...

Sharon: Não tem problema, gostamos de manter nossos clientes envolvidos.

Doug: Eu vou servir como elemento de ligação para esse projeto. Toda comunicação virá por meu intermédio ou alguém como Vinod que eu designe. Como estamos com um cronograma apertado, gostaria de estabelecer um cronograma de granularidade diária e conversar pessoalmente ou por e-mail com você diariamente sobre realizações, problemas etc. Eu sei que é muito, mas isso é o que eu acho adequado.

Sharon: Está certo.

Doug (pegando algumas páginas de papel sobre seu computador e entregando-as a Sharon): Eu escrevi um esboço de cronograma com datas dos prazos intermediários... o que você acha?

Sharon (depois de estudar o cronograma): Hmmm. Eu não estou certo de que isso vai funcionar para nós. Deixe-me trabalhar em uma alternativa que vou enviar por e-mail mais tarde, ainda hoje...

Doug: Certo.

17.4.2 Planejamento da WebApp — Engenharia da Web Interna

À medida que as WebApps se tornam mais difundidas e estratégicas ao negócio, muitas empresas têm optado por controlar o desenvolvimento internamente. Não surpreendentemente, WebE interna é gerida um pouco diferentemente do que um esforço terceirizado.

Document shared on www.docsity.com

> "O que você faz quando precisa ter um site Web funcionando ontem?"
>
> James Lewin

A gestão de projetos WebE pequenos e de tamanho moderado (isto é, menos do que 3-5 meses de duração) requer uma abordagem ágil que não enfatize a gestão do projeto, mas *não* elimine a necessidade de planejar. Princípios básicos de gestão de projeto (Capítulo 21) ainda se aplicam, mas a abordagem global é mais simples e menos formal. No entanto, à medida que o tamanho do projeto da WebApp cresce, gestão de projeto de engenharia da Web torna-se mais e mais semelhante a gestão de projeto de engenharia de software (Parte 4 deste livro). Os passos seguintes são recomendados para projetos WebE pequenos e de tamanho moderado.

AVISO

É importante reconhecer que os passos discutidos nesta seção podem ser realizados rapidamente. Em nenhum caso o planejamento de projetos de WebE desse tamanho deve gastar mais do que 5% do esforço global do projeto.

Entenda o escopo, as dimensões da modificação e as restrições de projeto. Nenhum projeto — independentemente de quão apertadas sejam as restrições de tempo — pode começar até que a equipe de projeto entenda o que deve ser construído. Coleta de requisitos e comunicação com o cliente são antecedentes essenciais para o planejamento efetivo da WebApp.

Defina uma estratégia de projeto incremental. Já mencionamos que as WebApps evoluem com o tempo. Se a evolução é descontrolada e caótica, a probabilidade de um resultado de sucesso é pequena. No entanto, se a equipe estabelece uma estratégia de projeto que define incrementos da WebApp (versões) que fornecem conteúdo e funcionalidade úteis para os usuários finais, o esforço de engenharia pode ser mais efetivamente focalizado.

Realize análise de risco. Uma discussão detalhada de análise de risco para projetos de engenharia de software tradicional é apresentada no Capítulo 25.[7] Todas as tarefas de gestão de risco são realizadas para projetos WebE, mas a abordagem para elas é abreviada.

Risco de cronograma e risco de tecnologia dominam a preocupação da maioria das equipes de engenharia da Web. Entre as muitas questões relacionadas a risco que a equipe deve formular e responder estão: Os incrementos planejados da WebApp podem ser entregues dentro do cronograma definido? Esses incrementos vão fornecer valor útil para os usuários finais enquanto os incrementos adicionais estiverem sendo construídos? Que impacto as solicitações de modificação terão nos cronogramas de entrega? A equipe entende a necessidade dos métodos, das tecnologias e das ferramentas de engenharia da Web? A tecnologia disponível é adequada para o trabalho? Modificações vão provavelmente requerer a introdução de novas tecnologias?

Desenvolva uma rápida estimativa. O enfoque da estimativa para a maioria dos projetos de engenharia da Web é de tópicos macroscópicos, em vez de microscópicos. A equipe de WebE avalia se os incrementos planejados da WebApp podem ser desenvolvidos com os recursos disponíveis de acordo com as restrições de cronograma definidas. Isso é obtido considerando o conteúdo e a função de cada incremento como um todo. A funcionalidade "microscópica" ou de subdivisão do trabalho do incremento, seguida do cálculo de estimativas de pontos multidados (ver Capítulo 23) não é normalmente conduzida.

Selecione um conjunto de tarefas (descrição do processo). Usando um arcabouço de processo (Capítulo 16), selecione um conjunto de tarefas de engenharia da Web que seja adequado para as características do problema, do produto, do projeto e do pessoal da equipe da engenharia da Web. Reconheça que o conjunto de tarefas pode ser adaptado para encaixar cada um dos incrementos de desenvolvimento.

Estabeleça um cronograma. Um cronograma do projeto de WebE tem granularidade relativamente fina para as tarefas a serem realizadas a curto prazo e depois granularidade muito mais grossa durante períodos posteriores (quando incrementos adicionais tiverem que ser entregues). Isto é, as tarefas de engenharia da Web são distribuídas ao longo do prazo do projeto para o incremento a ser desenvolvido. A distribuição de tarefas dos incrementos subseqüentes da WebApp é adiada até a entrega desse incremento.

7 Os leitores que não estão familiarizados com a terminologia e práticas básicas de gestão de riscos devem examinar o Capítulo 25 agora.

PONTO CHAVE

Independente do tamanho do projeto, é importante estabelecer marcos intermediários para que o progresso seja avaliado.

Defina mecanismos para acompanhar o projeto. Em um ambiente de desenvolvimento ágil, a entrega de um incremento operacional de software é freqüentemente a medida principal de progresso. Mas, muito antes de um incremento de software estar disponível, o engenheiro da Web vai inevitavelmente encontrar a questão "Onde estamos"? Em trabalho de engenharia de software convencional, o progresso é medido por determinação de marcos intermediários de prazo (por exemplo, uma revisão bem-sucedida de um produto de trabalho) que tiverem sido alcançados. Para projetos de engenharia da Web de tamanho pequeno ou moderado, marcos intermediários de prazo podem ser menos definidos e atividades formais de garantia de qualidade podem ser não enfatizadas. Assim, uma resposta pode ser derivada da consulta à equipe de engenharia da Web para determinar que atividades de arcabouço foram completadas. No entanto, essa abordagem pode não ser confiável. Outra abordagem é determinar quantos casos de uso foram implementados e quantos dos casos de uso (para um dado incremento) ainda restam ser implementados. Isso fornece uma indicação grosseira do relativo grau de "completeza" do incremento de projeto.

> "Progresso é feito corrigindo os erros resultantes do progresso feito."
>
> Claude Gibb

Estabeleça uma abordagem de modificação. Gestão de modificação é facilitada pela estratégia de desenvolvimento incremental que tem sido recomendada para WebApps. Como o prazo de desenvolvimento para um incremento é pequeno, é freqüentemente possível adiar a introdução de uma modificação até o próximo incremento, reduzindo, assim, os efeitos de atraso associados a modificações que precisam ser implementadas "voando". Uma discussão de gestão de configuração e de conteúdo para WebApps é apresentada no Capítulo 27.

FERRAMENTAS DE SOFTWARE

Gestão de Projetos WebE

Objetivo: Apoiar uma equipe de engenharia da Web no planejamento, gestão, controle e rastreamento de projetos de engenharia da Web.

Mecânica: Ferramentas de gestão de projeto possibilitam a uma equipe de WebE estabelecer um conjunto de tarefas de trabalho, atribuir esforço e responsabilidade específica para cada tarefa, estabelecer as dependências de tarefas, definir um cronograma, e acompanhar e controlar as atividade no projeto. Muitas ferramentas dessa categoria são baseadas na Web.

Ferramentas Representativas[8]

Business Engine, desenvolvida por Business Engine (www.businessengine.com), é um conjunto de ferramentas baseadas na Web que fornece plenas facilidades de gestão de projeto para projetos de WebE e de software convencional.

Iteamwork, desenvolvida por iTeamwork.com (www.iteamwork.com), "é uma aplicação livre, on-line, baseada na Web, de gestão de equipe de projeto que você usa com o seu navegador Web."

OurProject, desenvolvida por Our Project (www.ourproject.com), é um conjunto de ferramentas de gestão de projetos que é aplicado a projetos da WebE e de software convencional.

Proj-Net, desenvolvida por Rational Concepts, Inc. (www.rationalconcepts.com), implementa um "escritório virtual de projeto (virtual project office, VPO) para colaboração e comunicação".

StartWright (www.startwright.com/project1.htm) desenvolveu um dos recursos mais abrangentes da Web tanto para WebE quanto para software convencional de ferramentas e informação de gestão de projetos.

Deve-se notar que muitas ferramentas de gestão de projeto convencional (Parte 4 deste livro) podem também ser usadas efetivamente para projetos de WebE.

8 As ferramentas mencionadas não representam uma recomendação, mas em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

Document shared on www.docsity.com

17.5 Métricas para Engenharia da Web e WebApps

Engenheiros Web desenvolvem sistemas complexos, e como outros tecnologistas que realizam essa tarefa, devem usar métricas para aperfeiçoar os processos e produtos da engenharia da Web. No Capítulo 15, discutimos os usos estratégicos e táticos de métricas de software no contexto de engenharia de software. Esses usos também se aplicam a engenharia da Web.

AVISO

Em geral, o número de métricas de WebE que você deveria coletar e sua complexidade global deve ser diretamente proporcional ao tamanho da WebApp que tem de ser construída.

Para resumir, métricas de software fornecem uma base para aperfeiçoar o processo de software, aumentar a precisão das estimativas de projeto, melhorar o acompanhamento de projeto e aperfeiçoar a qualidade do software. Métricas de engenharia da Web poderiam, se adequadamente caracterizadas, alcançar todos esses benefícios e também aperfeiçoar a usabilidade, o desempenho da WebApp e a satisfação do usuário.

No contexto da engenharia da Web, as métricas têm três objetivos principais: (1) fornecer uma indicação da qualidade da WebApp de um ponto de vista técnico; (2) fornecer uma base para estimativa de esforço; e (3) fornecer uma indicação da WebApp do ponto de vista do negócio.

Nesta seção, resumimos um conjunto de métricas de esforço e complexidade comuns[9] para WebApps. Elas podem ser usadas para desenvolver um banco de dados histórico para estimativa de esforço. Além disso, métricas de complexidade podem em última análise levar a uma capacidade de avaliar quantitativamente um ou mais dos atributos técnicos das WebApps discutidos no Capítulo 16.

17.5.1 Métricas para Esforço da Engenharia da Web

Engenheiros Web gastam esforço humano realizando várias tarefas de trabalho, à medida que a WebApp evolui. Mendes e seus colegas [MEN01] sugerem um certo número de possíveis medidas de esforço para WebApps. Algumas ou todas elas poderiam ser registradas por uma equipe de engenharia da Web e posteriormente usadas para construir um banco de dados histórico para estimativa (Capítulo 23).

Tarefas de Autoria e Projeto da Aplicação

Medidas sugeridas	Descrição
esforço de estruturação	tempo para estruturar a WebApp e/ou derivar a arquitetura
esforço de interligação	tempo para interligar páginas para construir a estrutura das WebApps
planejamento de interface	tempo gasto para planejar a interface das WebApps
construção da interface	tempo gasto para implementar a interface das WebApps
esforço de teste dos links	tempo gasto para testar todos os links das WebApp
esforço de teste de mída	tempo gasto para testar toda a mídia das WebApp
esforço total	esforço de estruturação + esforço de interligação + planejamento de interface + construção da interface + esforço de teste dos links + esforço de teste de mídia

Autoria de Página

Medidas sugeridas	Descrição
esforço de texto	tempo gasto com autoria ou reuso de texto na página
esforço de links na página	tempo gasto com links de autoria nas páginas
esforço de estruturação de página	tempo gasto para estruturar página
esforço total de páginas	esforço de texto + esforço de links na página + esforço de estruturação de página

9 É importante notar que métricas de WebE estão ainda na infância.

Autoria de Mídia

Medidas sugeridas	Descrição
esforço de mídia	tempo gasto com autoria ou reuso de arquivos de mídia
esforço de digitação de mídia	tempo gasto para digitar mídia
esforço total de mídia	esforço de mídia + esforço de digitação de mídia

Autoria de Programa

Medidas sugeridas	Descrição
esforço de programação	tempo gasto com autoria de HTML, Java ou implementações com linguagens relacionadas
esforço de reuso	tempo para reusar/modificar programação existente

Veja na Web

Uma excelente referência sobre muitos assuntos relacionados a negócios na Internet é **www.internet.com**.

17.5.2 Métricas para Avaliar Valor de Negócio

É interessante notar que o pessoal de negócio tem ultrapassado consideravelmente os engenheiros da Web no desenvolvimento, coleta e uso de métricas para WebApps (por exemplo, [STE02], [NOB01]). Compreendendo a demografia dos usuários finais e seus padrões de uso, uma empresa ou organização pode desenvolver entrada imediata para conteúdo de WebApp mais significativa, vendas e esforços de marketing mais efetivos e mais lucrabilidade no negócio.

Os mecanismos necessários para coletar dados de valor de negócio são freqüentemente implementados pela equipe de engenharia da Web, mas a avaliação dos dados e as ações resultantes são realizadas por outros participantes. Por exemplo, considere que o número de visões de página pode ser determinado para cada visitante especial. Baseados nas métricas coletadas, visitantes que chegam do motor de busca X têm em média nove visões de página, enquanto visitantes que vêm do portal Y têm apenas duas visões de página. Essas médias podem ser usadas pelo departamento de marketing para distribuir os orçamentos de propaganda em flâmulas (propaganda no motor de busca X fornece mais exposição, com base nas métricas coletadas, do que propaganda no portal Y).

FERRAMENTAS DE SOFTWARE

Métricas da Web

Objetivo: Avaliar o modo pelo qual uma WebApp está sendo usada, as categorias de usuários e a usabilidade da WebApp.

Mecânica: A grande maioria das ferramentas para métricas da Web capta informação de uso tão logo a WebApp entra em serviço. Essas ferramentas fornecem uma ampla gama de dados que podem ser usados para avaliar quais elementos da WebApp são mais usados, como eles são usados e quem os usa.

Ferramentas Representativas[10]

Clicktracks, desenvolvida por clicktracks.com (www.clicktracks.com), é uma ferramenta de análise de registro de arquivo que exibe comportamento de visitante de site da Web diretamente em páginas do site.

Marketforce, desenvolvida por Coremetrics (www.Coremetrics.com), é representativa de muitas ferramentas que coletam dados que podem ser usados para avaliar o sucesso de WebApps de e-commerce.

Web Metrics Testbed, desenvolvida por NIST (zing.ncsl.nist.gov/WebTools/), é um conjunto de ferramentas baseadas na Web que avaliam a usabilidade de uma WebApp.

WebTrends, desenvolvida por netIQ (www.NetIQ.com), coleta uma ampla variedade de dados de uso para WebApps de todos os tipos.

10 As ferramentas mencionadas não representam uma recomendação, mas em vez disso uma exemplificação de ferramentas dessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

Document shared on www.docsity.com

Uma discussão completa sobre coleta e uso de métricas de valor de negócio (inclusive o debate sobre privacidade pessoal) está além do escopo deste livro. O leitor interessado deve examinar [INA02], [EIS02], [PAT02] ou [RIG01].

17.6 "Piores Práticas" de Projetos de WebApp

Algumas vezes o melhor modo de aprender como fazer algo corretamente é examinar como não fazê-lo! Durante a década passada, mais do que umas poucas WebApps falharam por conta de (1) desobediência aos princípios de gestão de projeto e de modificações (ainda que informal) resultou em uma equipe de engenharia da Web que "despencava das paredes"; (2) uma abordagem *ad hoc* para o desenvolvimento de WebApp falhou em produzir um sistema em funcionamento; (3) uma abordagem arrogante de coleta e análise de requisitos falhou em produzir um sistema que satisfizesse as necessidades dos usuários; (4) uma abordagem incompetente para projeto falhou em produzir o desenvolvimento de uma WebApp que fosse usável, funcional, extensível (manutenível) e testável; (5) uma abordagem de teste desfocada falhou em produzir um sistema que funcionasse antes da sua introdução.

Com essas realidades em mente, poderia valer a pena considerar um conjunto de "piores práticas" da engenharia da Web, adaptada de um artigo de Tom Bragg [BRA00]. Se seu e-projeto exibe algum deles, é imediatamente necessária uma ação que os remedie.

Pior prática 1: *Temos uma grande idéia, vamos começar a construir a WebApp — agora.* Não se importe em considerar se a WebApp é justificável em termos de negócio, se usuários vão realmente querer usá-la e se você entende os requisitos do negócio. O tempo é curto, temos que começar.

Realidade: Gaste algumas horas/dias e faça um caso de negócio para a WebApp. Certifique-se de que a idéia é endossada por aqueles que vão patrociná-la e por aqueles que vão usá-la.

Pior prática 2: *As coisas vão mudar constantemente, assim não tem sentido tentar entender os requisitos da WebApp.* Nunca escreva nada (perde tempo). Confie apenas na palavra falada.

Realidade: É verdade que os requisitos da WebApp evoluem à medida que as atividades de engenharia da Web prosseguem. É também rápido e simples transmitir informação verbalmente. No entanto, uma abordagem arrogante à coleta e análise de requisitos é um catalisador para ainda mais modificações (desnecessárias).

Pior prática 3: *Desenvolvedores cuja experiência dominante tenha sido em desenvolvimento de software tradicional podem desenvolver WebApps imediatamente. Nenhum treinamento novo é necessário.* Afinal de contas, software é software, não é?

Realidade: WebApps são diferentes. Uma grande variedade de métodos, tecnologias e ferramentas precisam ser aplicados habilmente. Treinamento e experiência com eles é essencial[11].

Pior prática 4: *Seja burocrático.* Insista em modelos de processo pesados, planilhas, muitas reuniões de "progresso" desnecessárias e líderes de projeto que nunca geriram um projeto de WebApp.

Realidade: Encoraje um processo ágil que enfatize a competência e a criatividade de uma equipe experiente de engenharia da Web. Depois saia da frente e deixe que eles façam o trabalho. Se dados relacionados precisam ser coletados (por razões legais ou para o cálculo de métricas), a entrada/coleta de dados deve ser tão simples e não obstrutiva quanto possível.

Pior prática 5: *Testar? Por que se incomodar?* Vamos dá-lo para alguns usuários finais e deixar que eles nos digam o que funciona e o que não funciona.

Realidade: Ao longo do tempo, usuários finais realizam "testes" rigorosos, mas eles ficam tão desgostosos com o desempenho fraco e não confiável, que eles desaparecem (para nunca mais voltar) muito antes dos problemas serem corrigidos.

Nos capítulos seguintes, consideramos métodos de engenharia da Web que vão ajudá-los a evitar esses erros.

11 Muitos projetos grandes de WebE exigem integração com aplicações e bancos de dados convencionais. Em tais instâncias, indivíduos com apenas experiência convencional podem e devem ser envolvidos.

17.7 Resumo

Formulação é uma atividade de comunicação com o cliente que define o problema que uma WebApp tem de resolver. Necessidade do negócio, metas e objetivos do projeto, categorias de usuários finais, principais funções e características e o grau de interoperabilidade com outras operações são todos identificados. À medida que informação mais detalhada e técnica é adquirida, a formulação torna-se análise de requisitos.

A equipe de WebE é composta de um grupo de membros técnicos e não técnicos que são organizados de um modo que lhes dá considerável autonomia e flexibilidade. Gestão de projeto é necessária durante a engenharia da Web, mas as tarefas de gestão de projeto são abreviadas e consideravelmente menos formais do que aquelas aplicadas a projetos convencionais de engenharia de software. Muitos projetos de WebApp são terceirizados, mas existe uma tendência crescente de desenvolvimento interno de WebApp. A gestão de projeto para cada abordagem difere tanto em estratégia quanto em tática.

Métricas de engenharia da Web estão na infância, mas têm potencial para fornecer uma indicação da qualidade da WebApp, fornecer base para esforço de estimativa e fornecer uma indicação do sucesso da WebApp do ponto de vista do negócio.

Referências Bibliográficas

[BRA00] Bragg, T., "Worst Practices for e-Business Projects: We Have Met the Enemy and He Is Us!", *Cutter IT Journal*, vol. 13, n. 4, abril de 2000, p. 35–39.

[CON02] Constantine, L. e L. Lockwood, "User-Centered Engineering for Web Applications", *IEEE Software*, vol. 19, n. 2, março/abril de 2002, p. 42–50.

[EIS02] Eisenberg, B., "How to Interpret Web Metrics", *ClickZ Today*, março de 2002, disponível em http://www.clickz.com/sales/traffic/article.php/992351.

[FUC02a] Fuccella, J. Pizzolato, J. e Franks, J., "Finding Out What Users Want from your Web Site," IBM developerWorks, 2002, http://www-106.ibm.com/ developerworks/library/moderator-guide/requirements.html.

[FUC02b] Fuccella, J. e Pizzolato, J., "Giving People What They Want: How to Involve Users in Site Design", IBM developerWorks, 2002, http://www-106.ibm.com/developerworks/ library/design-by-feedback/expectations.html.

[GNA99] Gnado, C. e Larcher, F., "A User-Centered Methodology for Complex and Customizable Web Applications Engineering", *Proc. First ICSE Workshop in Web Engineering*, ACM, Los Angeles, maio de 1999.

[HAN99] Hansen, S., Deshpande, Y. e Murugesan, S., "A Skills Hierarchy for Web Information System Development", *Proc. First ICSE Workshop on Web Engineering*, ACM, Los Angeles, maio de 1999.

[INA02] Inan, H. e Kean, M., *Measuring the Success of Your Web Site*, Longman Publishing, 2002.

[KID00] Kidder, T., *The Soul of a New Machine*, Back Bay Books (reprint edition), 2000.

[KUL00] Kulik, P. e Samuelsen, R., "e-Project Management for a New e-Reality", Project Management Institute, dez. de 2000, http://www.seeprojects.com/e-Projects/ e-projects.html.

[LOW98] Lowe, D. e Hall, W., Eds., *Hypertext and the Web — An Engineering Approach*, Wiley, 1998.

[MEN01] Mendes, E., Mosley, N., e Counsell, S., "Estimating Design and Authoring Effort", *IEEE Multimedia*, jan-março de 2001, p. 50–57.

[NOB01] Nobles, R. e Grady, K., *Web Site Analysis and Reporting*, Premier Press, 2001.

[PAT02] Patton, S., "Web Metrics That Matter", *CIO*, 15-11-2002. disponível em http://www.computerworld.com/developmenttopics/websitemgmt/story/ 0,10801,76002,00.html.

[PIC01] Pickering, C., "Building an Effective E-Project Team", *E-Project Management Advisory Service*, Cutter Consortium, vol. 2, n. 1, 2001, http://www.cutter.com/ consortium.

[POW98] Powell, T. A., *Web Site Engineering*, Prentice-Hall, 1998.

[RIG01] Riggins, F. e Mitra, S., "A Framework for Developing E-Business Metrics through Functionality Interaction", jan. de 2001, disponível em http:// digitalenterprise. org/metrics/metrics.html.

[STE02] Sterne, J., *Web Metrics: Proven Methods for Measuring Web Site Success*, Wiley, 2002.

[TIL99] Tilley, S. e Huang, S., "On the Emergence of the Renaissance Software Engineer", *Proc. 1st ICSE Workshop on Web Engineering*, ACM, Los Angeles, maio de 1999.

Problemas e Pontos a Considerar

17.1. Como formulação difere de coleta de requisitos? Como formulação difere de análise de requisitos e de modelagem de análise?

Document shared on www.docsity.com

17.2. Três questões fundamentais sobre formulação são propostas na Seção 17.1.1. Existem outras questões que você acha que podem ser formuladas nesse ponto? Em caso afirmativo, quais são elas e por que você as formularia?

17.3. No contexto de coleta de requisitos, o que é uma "categoria de usuário"? Dê exemplos de três categorias de usuários para um vendedor de livros *on-line*.

17.4. Considerando o site de e-commerce do *CasaSegura* discutido neste capítulo, que mecanismo de comunicação com o usuário você usaria para elicitar requisitos do sistema e por quê?

17.5. Com as próprias palavras, discuta como a informação coletada durante a comunicação com o cliente é "analisada" e qual é a saída dessa atividade.

17.6. Quais benefícios podem ser derivados da exigência do desenvolvimento de casos de uso como parte da atividade de coleta de requisitos?

17.7. Revise a tabela apresentada na Figura 17.1. Adicione três linhas a mais que diferenciem melhor projetos tradicionais de e-projetos.

17.8. Descreva o papel do editor Web com suas próprias palavras.

17.9. Revise as características das equipes de desenvolvimento ágil discutidas no Capítulo 4. Você acha que uma organização de equipe ágil é adequada para WebE? Você faria alguma mudança na organização para o desenvolvimento de WebApps?

17.10. Descreva cinco riscos associados com a terceirização do desenvolvimento de WebApps.

17.11. Descreva cinco riscos associados com o desenvolvimento interno de WebApps.

17.12. Considere as métricas para esforço de engenharia da Web discutidas na Seção 17.5.1. Tente desenvolver cinco ou mais métricas adicionais para uma ou mais categorias.

17.13. A facilidade de navegação em um site Web é um indicador importante da qualidade da WebApp. Desenvolva duas ou três métricas que poderiam ser usadas para indicar a facilidade de navegação.

17.14. Usando uma das referências citadas na Seção 17.5.2, discuta como métricas de valor de negócio podem ser usadas para apoiar a tomada de decisão pragmática de negócio.

Leituras e Fontes de Informação Adicionais

Métodos para formulação e coleta de requisitos de WebApps podem ser adaptados a partir da discussão de métodos similares para aplicação convencional de software. Leituras adicionais recomendadas nos Capítulos 7 e 8 contêm muita informação útil para o engenheiro da Web.

Flor *(Web Business Engineering,* Addison-Wesley, 2000) discute análise de negócio e tópicos relacionados que possibilitam ao engenheiro da Web melhor entender as necessidades do cliente. Usabilidade de WebApp é um conceito que enfatiza muito da informação definida como parte da formulação e coleta de requisitos. Krug and Black *(Don't Make Me Think: A Common Sense Approach to Web Usability,* Que Publishing, 2000) oferecem muitas diretrizes e exemplos que podem ajudar o engenheiro da Web a traduzir os requisitos do usuário para uma WebApp efetiva.

Gestão de projeto para projetos de WebE se apóia em muitos dos mesmos princípios e conceitos que são aplicados em projetos de software convencional. No entanto, agilidade é palavra de alerta. Wallace *(Extreme Programming for Web Projects,* Addison-Wesley, 2003) descreve como desenvolvimento ágil pode ser usado para WebE e oferece discussões úteis sobre tópicos de gestão de projeto. Shelford e Remillard *(Real Web Project Management,* Addison-Wesley, 2003), O'Connell *(How to Run Successful Projects in Web Time,* Artech House, 2000), Freidlein *(Web Project Management,* Morgan Kaufman, 2000), e Gilbert *(90 Days to Launch: Internet Projects on Time and on Budget,* Wiley, 2000) discutem uma ampla variedade de tópicos de gestão de projeto para WebE. Whitehead *(Leading a Software Development Team,* Addison-Wesley, 2001) apresenta muitas diretrizes úteis que podem ser adaptadas para equipes de engenharia da Web.

Técnicas de uso de métricas Web em tomada de decisão de negócio são apresentadas nos livros de Sterne [STE02], Inan [INA02], Nobles [NOB01] e Menasce e Almeida *(Capacity Planning for Web Services: Metrics, Models and Methods,* Prentice-Hall, 2001). "Piores práticas" são consideradas por Ferry e Ferry *(77 Sure-Fire Ways to Kill a Software Project,* iUniverse.com, 2000).

Uma ampla variedade de fontes de informação sobre formulação e planejamento de engenharia da Web está disponível na Internet. Uma lista atualizada de referências que são relevantes para formulação e planjemento pode ser encontrada no site deste livro: **http://www.mhhe.com/pressman.**

Modelagem de Análise para Aplicações da Web

CAPÍTULO 18

Conceitos-Chave

análise de relacionamento . 421
análise navegacional 422
árvores de dados..... 415
casos de uso 411
hierarquia de usuário 410
modelagem de análise.... 410
modelo de configuração... 420
modelo de conteúdo 414
modelo de interação 417
modelo funcional..... 419
relacionamentos de conteúdo 415
RNA 421

À primeira vista, há uma aparente contradição quando consideramos modelagem de análise no contexto de engenharia da Web. Afinal de contas, temos observado (Capítulo 16) que WebApps têm um imediatismo e uma volatilidade que vão contra a modelagem detalhada tanto no nível de análise quanto de projeto. E, se fizermos qualquer modelagem, a filosofia dinâmica (um modelo de processo adequado para muitos projetos de engenharia da Web) sugere que a modelagem de análise seja menos enfatizada em favor de modelagem de projeto limitada. Franklin [FRA02] observa essa situação quando escreve:

> Sites Web são tipicamente complexos e altamente dinâmicos. Eles requerem fases curtas de desenvolvimento a fim de obter o produto pronto e rodando rapidamente. Com freqüência, desenvolvedores vão direto para a fase de codificação sem realmente entender o que eles estão tentando construir ou como querem construir. Codificação do lado do servidor é freqüentemente feita *ad hoc*; tabelas nos bancos de dados são adicionadas quando necessário e a arquitetura evolui de um modo um tanto não intencional. Mas alguma modelagem e engenharia de software disciplinada podem tornar o processo de desenvolvimento de software muito mais suave e garantir que o sistema Web seja mais manutenível no futuro.

É possível conciliar os dois modos? Podemos fazer "alguma modelagem e engenharia de software disciplinada" e ainda trabalhar efetivamente em um mundo em que imediatismo e volatilidade reinam? A resposta é um sim qualificado.

Panorama

O que é? A análise de uma potencial aplicação Web focaliza três questões importantes: (1) que informação ou conteúdo deve ser apresentado ou manipulado; (2) que funções devem ser realizadas pelo usuário final; e (3) que comportamentos devem ser exibitos pela WebApp quando ela apresenta conteúdo e realiza funções? As respostas a essas questões são representadas como parte de um modelo de análise que engloba uma variedade de representações UML.

Quem faz? Engenheiros da Web, desenvolvedores não técnicos de conteúdo e interessados que participam da criação do modelo de análise.

Por que é importante? Ao longo deste livro enfatizamos a necessidade de entender o problema antes de começar a resolvê-lo. Modelagem de análise é importante não porque habilita uma equipe de engenharia da Web a desenvolver um modelo concreto de requisitos da WebApp (as coisas modificam-se muito freqüentemente para que isso seja uma expectativa realista), mas, em vez disso, modelagem de análise habilita um engenheiro da Web a definir tópicos fundamentais do problema — coisas improváveis de se modificar (no curto prazo). Quando o conteúdo, função e comportamento fundamentais são entendidos, projeto e construção são facilitados.

Quais são os passos? Modelagem de análise enfoca quatro tópicos fundamentais do problema — conteúdo, interação, função e configuração. Análise de conteúdo identifica as classes de conteúdo e suas colaborações. Análise de interação descreve elementos básicos da interação com o usuário, navegação e os comportamentos do sistema que ocorrem como conseqüência. Análise de função define as funções da WebApp realizadas para o usuário e a seqüência de processamento que ocorre como conseqüência. Análise de configuração identifica o(s) ambiente(s) operacional(is) no(s) qual(is) a WebApp reside.

Qual é o produto do trabalho? O modelo de análise é constituído de um conjunto de diagramas UML e texto que descreve o conteúdo, interação, função e configuração.

Como tenho certeza de que fiz corretamente? Os produtos de trabalho da modelagem de análise precisam ser revisados quanto à correção, completeza e consistência.

Document shared on www.docsity.com

Uma equipe de engenharia da Web deve adotar modelagem de análise quando a maioria ou todas as condições seguintes forem encontradas:

- A WebApp a ser construída é grande e/ou complexa.
- O número de interessados é grande.
- O número de engenheiros da Web e outros colaboradores é grande.
- As metas e os objetivos (determinados durante a formulação) da WebApp vão afetar os fundamentos básicos do negócio.
- O sucesso da WebApp terá um forte efeito no sucesso do negócio.

Se essas condições não são apresentadas, é possível desviar a ênfase da modelagem de análise, usando informação obtida durante a formulação e a coleta de requisitos (Capítulo 17) como base para criar um modelo de projeto para a WebApp. Em tais circunstâncias, modelagem de análise limitada pode ocorrer, mas vai estar com o projeto.

18.1 Análise de Requisitos para WebApps

Análise de requisitos de WebApps engloba três tarefas principais: formulação, coleta de requisitos[1] e modelagem de análise. Durante a formulação, a motivação básica (metas) e objetivos para a WebApp são identificadas e as categorias de usuários definidas. À medida que a coleta de requisitos começa, a comunicação entre a equipe de engenharia da Web e os interessados na WebApp (por exemplo, clientes e usuários finais) se intensifica. Os requisitos de conteúdo e funcionais são listados e cenários de interação (casos de uso) escritos sob o ponto de vista do usuário final são desenvolvidos. O objetivo é estabelecer um entendimento básico do porquê a WebApp deve ser construída, quem vai usá-la e qual(is) problema(s) vai(ão) resolver para seus usuários.

> "Os princípios de engenharia, planejar antes de projetar e projetar antes de construir, se mantiveram ao longo de todas as transições de tecnologia anteriores; eles vão sobreviver igualmente a essa transição."
>
> **Watts Humphrey**

18.1.1 A Hierarquia de Usuário

AVISO

É uma boa idéia construir uma hierarquia de usuários. Ela lhe fornece um instantâneo da população de usuários e uma verificação cruzada para ajudar a garantir que as necessidades de cada usuário tenham sido observadas.

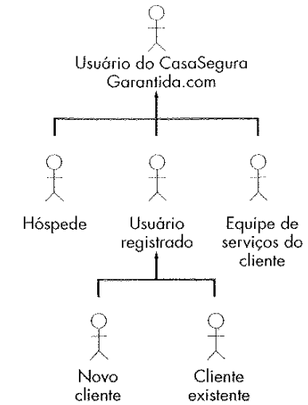
As categorias de usuários finais que vão interagir com a WebApp são identificadas como parte das tarefas de formulação e coleta de requisitos. Na maioria dos casos, categorias de usuários são relativamente limitadas e uma representação UML delas é desnecessária. No entanto, quando o número de categorias de usuário aumenta, é algumas vezes prudente desenvolver uma *hierarquia de usuário* como mostra a Figura 18.1. A figura mostra os usuários para o site de e-comércio do *CasaSeguraGarantida.com* discutido nos Capítulos 16 e 17.

As categorias de usuário (freqüentemente chamadas de *atores*) apresentadas na Figura 18.1 fornecem uma indicação da funcionalidade a ser oferecida pela WebApp e indicam a necessidade de casos de uso a serem desenvolvidos para cada usuário final (ator) observado na hierarquia. Referindo-se à figura, o **usuário do CasaSeguraGarantida.com** no topo da hierarquia representa a classe mais geral de usuário (categoria) e é refinada em níveis mais baixos. Um **hóspede** é um usuário que visita o site, mas não se registra. Tais usuários estão freqüentemente procurando informação geral, comparação de compras e/ou estão interessados no conteúdo ou na funcionalidade "livre". Um **usuário registrado** gasta tempo para fornecer informação de contato (com outros dados demográficos solicitados por formulários de entrada). Subcategorias de **usuário registrado** incluem:

1 Formulação e coleta de requisitos são discutidas em detalhe no Capítulo 17.

FIGURA 18.1

Hierarquia de usuários para CasaSegura Garantida.com



- **novo cliente** — um usuário registrado que deseja personalizar e depois comprar componentes do *CasaSegura* (e, portanto, precisa interagir com a funcionalidade de e-comércio da WebApp);
- **cliente existente** — um usuário que já possui componentes do *CasaSegura* e está usando a WebApp para (1) comprar componentes adicionais; (2) para adquirir informação de suporte técnico; ou (3) contatar o suporte ao cliente.

Membros da **equipe de serviços de cliente** são usuários especiais que podem também interagir com o conteúdo e a funcionalidade do *CasaSeguraGarantida.com* à medida que eles ajudam os clientes que contataram o suporte ao cliente do *CasaSegura*.

18.1.2 Desenvolvimento de Casos de Uso

Franklin [FRA01] refere-se a casos de uso como "feixes de funcionalidade". Essa descrição capta a essência dessa importante técnica de modelagem de análise.[2] Casos de uso são desenvolvidos para cada categoria de usuário descrita na hierarquia de usuários. No contexto de engenharia da Web, o caso de uso em si é relativamente informal — um parágrafo narrativo que descreve uma interação específica entre um usuário e a WebApp.[3]

A Figura 18.2 representa um diagrama de caso de uso UML para a categoria de usuário **novo cliente** (Figura 18.1). Cada elipse no diagrama representa um caso de uso que descreve uma interação específica entre **novo cliente** e a WebApp. Por exemplo, a primeira interação é descrita pelo caso de uso *entrar no CasaSeguraGarantida.com*. Não mais do que um parágrafo simples seria necessário para descrever essa interação comum.

A principal funcionalidade da WebApp (e os casos de uso relevantes a ela) é observada dentro das caixas tracejadas na Figura 18.2. Estas são referidas como "pacotes" em UML e representam funcionalidade específica. Dois pacotes são observados: *personalização* e *e-commerce*.

Como exemplo, consideramos o pacote de casos de uso de *personalização*. Um novo cliente precisa descrever o ambiente do prédio no qual o *CasaSegura* vai ser instalado. Para realizar isso, os casos de uso *descrever leiaute do prédio, selecionar componentes do CasaSegura* e *salvar configuração* são iniciados pelo **novo cliente**. Considere esses casos de uso preliminares escritos sob o ponto de vista de um **novo cliente:**

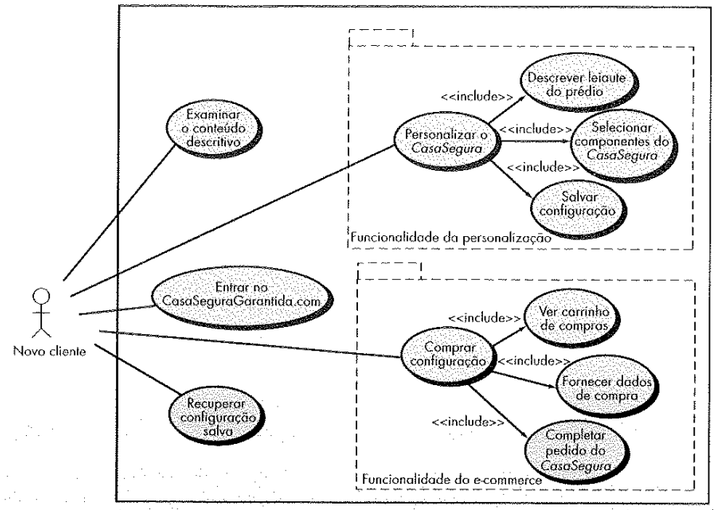
2 Técnicas para desenvolvimento de casos de uso foram discutidas em detalhe anteriormente neste livro (veja os Capítulos 7 e 8).

3 Embora seja possível desenvolver descrições de casos de uso mais formais, a necessidade de agilidade para WebE freqüentemente descarta essa abordagem.

Document shared on www.docsity.com

FIGURA 18.2

Diagrama de casos de uso para novo cliente



AVISO

À medida que o tamanho de uma WebApp aumenta e a modelagem de análise torna-se mais rigorosa, os casos de uso preliminares apresentados aqui teriam que ser expandidos para se aproximarem do formato sugerido na Seção 8.5 do Capítulo 8.

Caso de uso: *descrever leiaute do prédio*

A WebApp formulará algumas questões gerais sobre o ambiente no qual planejo instalar o *CasaSegura* — o número, tamanho e tipo de cômodos, número de pavimentos, número de portas e janelas externas. A WebApp vai me habilitar a construir um esboço de planta baixa montando contornos dos cômodos de cada pavimento. Estarei habilitado a dar à planta baixa um nome e salvá-la para referência futura (veja o caso de uso: *salvar configuração*).

Caso de uso: *selecionar componentes do CasaSegura*

A WebApp vai então recomendar componentes de produto (por exemplo, painéis de controle, sensores, câmeras) e outras características (por exemplo, funcionalidade baseada em PC implementada em software) para cada cômodo e entrada externa. Se solicito alternativas, a WebApp vai fornecê-las, se existirem. Poderei obter informação descritiva e de preço de cada componente de produto. A WebApp vai criar e mostrar uma lista de materiais à medida que seleciono os vários componentes. Poderei dar à lista de materiais um nome e salvá-la para referência futura (veja o caso de uso: *salvar configuração*).

Caso de uso: *salvar configuração*

A WebApp vai me permitir salvar dados de personalização de modo que eu possa retornar a eles mais tarde. Posso salvar o leiaute do prédio e a lista de materiais do *CasaSegura* que eu escolher para o leiaute. Para conseguir isso, forneço um identificador único para o leiaute do prédio e para a lista de materiais. Também forneço uma senha especial de configuração que deve ser validada antes que eu possa ter acesso à informação salva.

Embora consideravelmente mais detalhe possa ser fornecido para cada um desses casos de uso, a descrição de texto informal fornece informação útil. Descrições similares podem ser desenvolvidas para cada elipse da Figura 18.2

CASASEGURA

Refinamento de Casos de Uso para WebApps

A cena: Escritório de Doug Miller.

Os personagens: Doug Miller (gerente do grupo de engenharia de software do *CasaSegura*), Sharon Woods, gerente da equipe do fornecedor terceirizado para o site de e-comércio do *CasaSegura* e Sam Chen, gerente da organização de suporte ao cliente do *CasaSeguraGarantida*.com.

A conversa:

Doug: Estou feliz por ouvir que as coisas estão progredindo bem, Sharon. A modelagem de análise já está quase completa?

Sharon (sorrindo): Estamos progredindo. O único conjunto de casos de uso ainda não desenvolvido da hierarquia de usuário [Figura 18.1] é o da categoria *equipe de serviço ao cliente*.

Doug (olhando para Sam): E você tem eles aqui, Sam?

Sam: Tenho. Eu os enviei por e-mail para você, Sharon, e com cópia para você, Doug. Aqui está uma versão em papel.(Ele entrega algumas folhas a Doug e a Sharon.)

Sam: Do modo que estamos considerando isso, desejamos usar o site do *CasaSeguraGarantida*.com como ferramenta de apoio quando um cliente faz um pedido por telefone. Nossos representantes no telefone vão completar todos os formulários necessários etc. e processar o pedido do cliente.

Doug: Por que não apenas encaminhar o cliente para o site?

Sam (sorrindo): Vocês técnicos pensam que todo mundo se sente confortável na Web. Eles não se sentem! Muitas pessoas ainda gostam do telefone, então temos que lhes dar essa opção. Mas não queremos construir um sistema de processamento de pedido separado quando a maioria das partes já está pronta na Web.

Sharon: Faz sentido.

(Todas as partes lêem os casos de uso [segue-se um exemplo]:)

Caso de uso: *descrever leiaute do prédio* [observe que difere do caso de uso de mesmo nome para a categoria *novo cliente*].

Vou pedir ao cliente (via telefone) para descrever cada cômodo do prédio e entrarei com as dimensões e outras características do cômodo em um formulário grande projetado especificamente para o pessoal de apoio ao cliente. Uma vez introduzidos os dados do prédio, posso salvá-los sob o nome ou número de telefone do cliente.

Sharon: Sam, você tem sido um tanto conciso nas suas descrições preliminares de caso de uso. Acho que vamos precisar recheá-las um pouco.

Doug (assentindo com a cabeça): Eu concordo.

Sam (carrancudo): Como assim?

Sharon: Bem... sua menção a "um formulário grande projetado especificamente para o pessoal de apoio ao cliente". Vamos precisar de mais detalhes.

Sam: O que eu quero dizer é que não precisamos fazer os nossos representantes passar pelo processo pelo qual um cliente on-line passa. Um formulário grande deve resolver o problema.

Sharon: Vamos esboçar que aparência o formulário deve ter.

As partes trabalham para fornecer detalhe suficiente para permitir que a equipe de Sharon faça uso efetivo do caso de uso.

18.1.3 Refinamento do Modelo de Caso de Uso

À medida que os diagramas de caso de uso são criados para cada categoria de usuário, uma visão de alto nível dos requisitos da WebApp externamente observáveis é desenvolvida. Casos de uso são organizados em pacotes funcionais e cada pacote é avaliado [CON00] para garantir que ele seja:

? Como avaliamos pacotes de casos de uso que foram agrupados por função de usuário?

- *Compreensível* — todos os interessados entendem o objetivo do pacote.
- *Coesivo* — o pacote trata de funções que estão diretamente relacionadas entre si.
- *Fracamente acoplado* — funções ou classes dentro do pacote colaboram entre si, mas a colaboração fora do pacote é restrita a um mínimo.
- *Hierarquicamente raso* — hierarquias funcionais profundas são difíceis de navegar e difíceis para o usuário final entender; assim, o número de níveis dentro da hierarquia de casos de uso deve ser minimizado sempre que possível.

Como análise e modelagem de requisitos são atividades interativas, é provável que novos casos de uso venham a ser adicionados aos pacotes definidos, que casos de uso existentes venham a ser refinados e que casos de uso específicos possam ser realocados para diferentes pacotes.

Document shared on www.docsity.com

18.2 Modelo de Análise para WebApps

Um *modelo de análise* de WebApp é guiado por informação contida nos casos de uso desenvolvidos para a aplicação. Descrições de caso de uso são analisadas gramaticalmente para identificar classes potenciais de análise e operações e atributos associados com cada classe. O conteúdo a ser apresentado pela WebApp é identificado e funções a ser realizadas são extraídas das descrições de casos de uso. Finalmente, os requisitos específicos da implementação devem ser desenvolvidos de modo que o ambiente e a infra-estrutura que apóiam a WebApp possam ser construídos.

São quatro as atividades de análise — cada uma contribuiundo para a criação de um modelo de análise completo:

? Que tipos de atividade de análise ocorrem durante a modelagem de uma WebApp?

- *Análise de conteúdo* identifica todo o espectro de conteúdo a ser fornecido pela WebApp. O conteúdo inclui texto, gráficos, imagens e dados de vídeo e de áudio.
- *Análise de interação* descreve o modo pelo qual o usuário interage com a WebApp.
- *Análise funcional* define as operações que serão aplicadas ao conteúdo da WebApp e descreve outras funções de processamento independentes do conteúdo, mas necessárias para o usuário final.
- *Análise de configuração* descreve o ambiente e a infra-estrutura nos quais a WebApp reside.

A informação coletada durante essas quatro tarefas de análise deve ser revisada, modificada quando necessário e então organizada em um modelo que pode ser passado aos projetistas da WebApp.

O modelo em si contém elementos estruturais e dinâmicos. *Elementos estruturais* identificam as classes de análise e objetos de conteúdo necessários para criar uma WebApp que atenda às necessidades dos interessados. Os *elementos dinâmicos* do modelo de análise descrevem como os elementos estruturais interagem entre si e com os usuários finais.

> "WebApps bem-sucedidas permitem aos clientes satisfazer às suas necessidades por si mesmos, melhor, de forma mais rápida e econômica, em vez de trabalhar por intermédio de usuários finais empregados da empresa."
>
> Mark McDonald

18.3 O Modelo de Conteúdo

O *modelo de conteúdo* contém elementos estruturais que fornecem uma visão importante dos requisitos de conteúdo para uma WebApp. Esses elementos estruturais englobam objetos de conteúdo (por exemplo, texto, imagens gráficas, fotografias, imagens de vídeo, áudio) apresentados como partes da WebApp. Além disso, o modelo de conteúdo inclui todas as classes de análise — entidades visíveis ao usuário criadas ou manipuladas à medida que um usuário interage com a WebApp. Uma classe de análise engloba atributos que a descrevem, operações que realizam o comportamento necessário para a classe e colaborações que permitem que a classe se comunique com outras classes.

Como outros elementos do modelo de análise, o modelo de conteúdo é derivado de um exame cuidadoso dos casos de uso desenvolvidos para a WebApp. Casos de uso são analisados sintaticamente para extrair os objetos de conteúdo e as classes de análise.

18.3.1 Definição dos Objetos de Conteúdo

Aplicações Web apresentam informação preexistente — chamada de *conteúdo* — para um usuário final. O tipo e a forma do conteúdo abrangem um amplo espectro de sofisticação e complexidade. Conteúdo pode ser desenvolvido antes da implementação da WebApp, enquanto a WebApp está sendo construída, ou muito depois de a WebApp estar operacional. Em qualquer caso, é incorporado via referência navegacional na estrutura global da WebApp. Um *objeto de conteúdo* pode

PONTO CHAVE

Um objeto de conteúdo é qualquer item de informação coesiva que deve ser apresentado a um usuário final. Tipicamente, objetos de conteúdo são extraídos de casos de uso.

ser uma descrição textual de um produto, um artigo descrevendo um evento que é notícia, uma ação fotografada para um evento esportivo, uma representação animada de um logotipo de uma empresa, um vídeo curto de um discurso ou uma cobertura de áudio para uma coleção de slides do PowerPoint.

Objetos de conteúdo são extraídos de casos de uso pelo exame da descrição de cenário à procura de referências diretas ou indiretas ao conteúdo. Por exemplo, no caso de uso *selecionar componentes do CasaSegura*, encontramos a sentença:

> Eu poderei obter informação descritiva e de preço de cada componente de produto.

Embora não haja referência direta ao conteúdo, ela está implícita. O engenheiro da Web pode encontrar-se com o autor do caso de uso e obter um entendimento mais detalhado do que "informação descritiva e de preço" significa. Nesse caso, o autor do caso de uso pode indicar que "informação descritiva" inclui (1) uma descrição geral de um parágrafo do componente; (2) uma fotografia do componente; (3) uma descrição técnica de vários parágrafos do componente; (4) um diagrama esquemático do componente mostrando como ele se encaixa em um sistema típico do *CasaSegura*; e (5) um vídeo instantâneo que mostra como instalar o componente em um ambiente predial típico.

É importante observar que cada um desses objetos de conteúdo precisa ser desenvolvido (freqüentemente por desenvolvedores de conteúdo que *não* são engenheiros da Web) ou adquirido para a integração na arquitetura da WebApp (discutida no Capítulo 19.)

> "A Web — tanto conteúdo, tão pouco tempo."
>
> Autor desconhecido

PONTO CHAVE

Uma árvore de dados representa uma hierarquia de objetos de conteúdo.

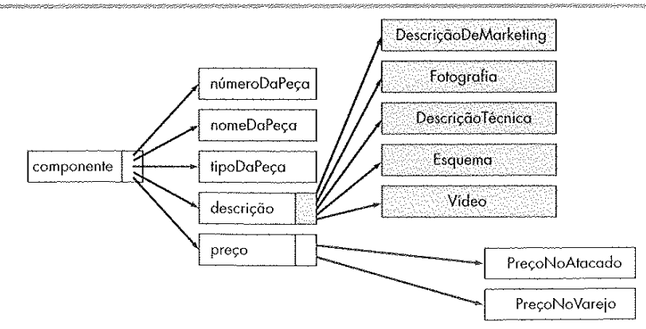
18.3.2 Relacionamentos e Hierarquia de Conteúdo

Em muitas instâncias, uma lista simples de objetos de conteúdo, acoplados com descrição breve de cada objeto, é suficiente para definir os requisitos de conteúdo que devem ser projetados e implementados. No entanto, em alguns casos, o modelo de conteúdo pode conter diagramas entidade-relacionamento (Capítulo 8) ou *árvores de dados* [SRI01] que mostram os relacionamentos entre os objetos de conteúdo e/ou a hierarquia de conteúdo mantida por uma WebApp.

Considere a árvore de dados criada para um componente do *CasaSegura* mostrada na Figura 18.3. A árvore representa uma hierarquia de informação usada para descrever o componente (mais tarde veremos que um componente do *CasaSegura* é, na realidade, uma classe de análise dessa aplicação). Itens de dados simples ou compostos (um ou mais valores de dados) são representados por retângulos não sombreados. Objetos de conteúdo são representados por retângulos sombreados. Na figura, **descrição** é definida por cinco objetos de conteúdo (os retângulos sombreados). Em alguns casos, um ou mais desses objetos podem ser mais refinados à medida que a árvore de dados se expande.

FIGURA 18.3

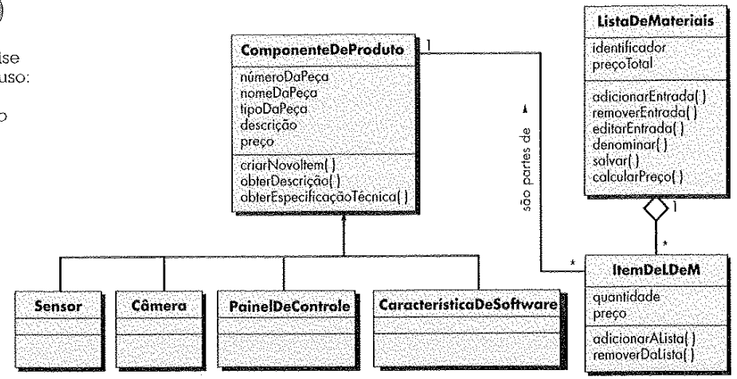
Árvore de dados para um componente do *CasaSegura*



Document shared on www.docsity.com

FIGURA 18.4

Classes de análise para o caso de uso: *selecionar componentes do CasaSegura*



18.3.3 Classes de Análise[4] de WebApps

Como já mencionamos, classes de análise são derivadas pelo exame de cada caso de uso. Por exemplo, considere o caso de uso preliminar: *selecionar componentes do CasaSegura* apresentado na Seção 18.1.2.

> **Caso de uso:** *selecionar componentes do CasaSegura*
>
> A WebApp vai então recomendar componentes de produto (por exemplo, painéis de controle, sensores, câmeras) e outras características (por exemplo, funcionalidade baseada em PC, implementada em software) para cada cômodo e entrada externa. Se solicito alternativas, a WebApp vai fornecê-las, se existirem. Poderei obter informação descritiva e de preço de cada componente de produto. A WebApp vai criar e mostrar uma lista de materiais à medida que seleciono os vários componentes. Poderei dar à lista de materiais um nome e salvá-la para referência futura (veja o caso de uso: *salvar configuração*).

Uma rápida análise gramatical do caso de uso identifica duas classes candidatas (sublinhadas): **ComponentesDeProduto** e **ListaDeMateriais**. Uma primeira descrição de cada classe é mostrada na Figura 18.4.

A classe **ComponenteDeProduto** engloba todos os componentes do *CasaSegura* que podem ser comprados para personalizar o produto para uma particular instalação. É uma representação generalizada de **Sensor**, **Câmera**, **PainelDeControle** e **CaracteristicaDeSoftware**. Cada objeto **ComponenteDeProduto** contém informação correspondente à árvore de dados mostrada na Figura 18.3 para a classe. Alguns desses atributos da classe são itens de dados simples ou compostos e outros são objetos de conteúdo (veja a Figura 18.3). Operações relevantes à classe são também apresentadas.

A classe **ListaDeMateriais** engloba uma lista de componentes que um **novo cliente** selecionou. **ListaDeMateriais** é, na verdade, uma agregação de **ItemDeLDeM** (várias instâncias de **ItemDeLDeM** formam uma **ListaDeMateriais**) — uma classe que constrói uma lista composta de cada componente a ser comprado e atributos específicos do componente, como mostra a Figura 18.4.

Cada caso de uso identificado para *CasaSeguraGarantida.com* é analisado para procura de objetos de análise. Modelos de classe similares ao descrito nesta seção são desenvolvidos para cada caso de uso.

4 Uma discussão detalhada da mecânica para identificar e representar classes de análise foi apresentada no Capítulo 8. Se você ainda não o fez, revise o Capítulo 8 agora.

18.4 O Modelo de Interação

A vasta maioria de WebApps possibilita uma "conversa" entre um usuário final e a funcionalidade, o conteúdo e o comportamento da aplicação. Esse *modelo de interação* é composto de quatro elementos: (1) casos de uso, (2) diagramas de seqüência, (3) diagramas de estado,[5] e (4) um protótipo de interface com o usuário. Em adição a essas representações, a interação também é representada no contexto do modelo navegacional (Seção 18.7).

AVISO

As técnicas associadas com a análise de tarefas (Capítulo 12) podem ser usadas para ajudar a definir os modos de interação do usuário.

Casos de uso. Casos de uso são o elemento dominante do modelo de interação para WebApps. Não é incomum descrever 100 ou mais casos de uso quando WebApps grandes e complexas são analisadas, projetadas e construídas. No entanto, uma porcentagem relativamente pequena desses casos de uso descreve as principais interações entre as categorias de usuário final (atores) e o sistema. Outros casos de uso refinam as interações, fornecendo os detalhes de análise necessários para conduzir o projeto e a construção.

Diagramas de seqüência. Diagramas de seqüência UML fornecem uma representação abreviada da maneira pela qual ações de usuário (os elementos dinâmicos de um sistema definido por casos de uso) colaboram com classes de análise (os elementos estruturais de um sistema definido por diagramas de classe). Como as classes de análise são extraídas das descrições de casos de uso, há necessidade de garantir que exista rastreabilidade entre as classes que foram definidas e os casos de uso que descrevem as interações do sistema.

Nos capítulos anteriores mencionamos que os diagramas de seqüência fornecem uma ligação entre as ações descritas no caso de uso e as classes de análise (entidades estruturais). Conallen [CON00] observa isso quando escreve: "A intercalação de elementos dinâmicos e estruturais do modelo [de análise] é a ligação-chave na rastreabilidade do modelo e deve ser levada muito a sério".

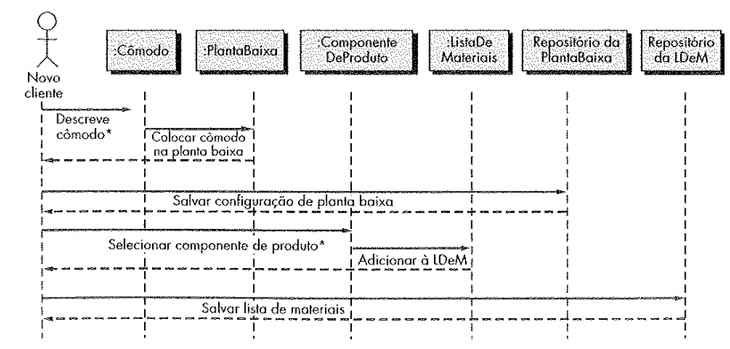
AVISO

Em alguns casos, extratos do texto real do caso de uso podem ser reproduzidos na coluna do lado esquerdo (abaixo do usuário) de modo que a rastreabilidade direta possa ser representada.

Um diagrama de seqüência para o caso de uso *Selecionar componentes do CasaSegura* é mostrado na Figura 18.5. Os eixos verticais do diagrama representam ações definidas no caso de uso. Os eixos horizontais identificam as classes de análise usadas à medida que o caso de uso prossegue. Por exemplo, um novo cliente deve primeiro descrever cada cômodo do prédio (o asterisco que se segue a "descrever cômodo" indica que a ação é iterativa). Para conseguir isso, o novo cliente responde a questões sobre o tamanho, as portas e as janelas etc. do cômodo. Uma vez definido um cômodo, ele é colocado na planta baixa do prédio. O novo cliente então descreve o cômodo seguinte ou prosssegue para a ação seguinte (que é salvar a configuração da planta baixa). O movimento para a direita e para baixo do diagrama de seqüência amarra cada classe de análise a ações do caso

FIGURA 18.5

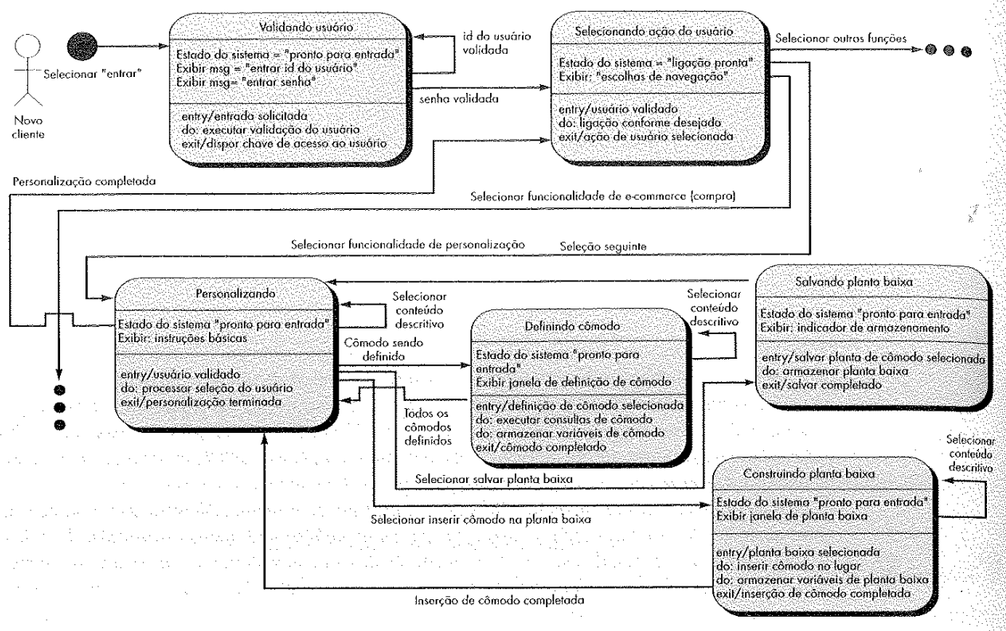
Diagrama de seqüência para o caso de uso: *selecionar componentes do CasaSegura*



5 Cada um desses elementos é uma importante notação UML e foi descrita no Capítulo 8.

Document shared on www.docsity.com

FIGURA 18.6 Diagrama parcial de estado para a interação de novo cliente



de uso. Se uma ação do caso de uso é omitida do diagrama, o engenheiro da Web deve reavaliar a descrição das classes de análise para determinar se uma ou mais classes foram omitidas. Diagramas de seqüência podem ser criados para cada caso de uso quando as classes de análise forem definidas para o caso de uso.

Diagramas de estado. O diagrama de estado UML (Capítulo 8) fornece uma outra representação do comportamento dinâmico da WebApp à medida que uma interação ocorre. Como a maioria das representações de modelagem usadas na engenharia da Web (ou na engenharia de software), o digrama de estado pode ser representado em diferentes níveis de abstração. A Figura 18.6 mostra um diagrama parcial de estado, de nível mais alto (alto nível de abstração) para a interação entre um novo cliente e a WebApp *CasaSeguraGarantida.com*.

No diagrama de estado mostrado, seis estados externamente observáveis são identificados: *validando usuário, selecionando ação do usuário, personalizando, definindo cômodo, construindo planta baixa e salvando planta baixa*. O diagrama de estado indica os eventos necessários para mover o novo cliente de um estado para outro, a informação exibida quando se entra no estado, o processamento que ocorre no estado e a condição de saída que causa uma transição de um estado para outro.

Como casos de uso, diagramas de seqüência e diagramas de estado apresentam informação relacionada, é razoável perguntar por que todos os três são necessários. Em alguns casos eles não são. Casos de uso podem ser suficientes em algumas situações, no entanto, estes fornecem uma visão um tanto unidimensional da interação. Diagramas de seqüência apresentam uma segunda dimensão que é mais de natureza procedimental (dinâmica). Diagramas de estado fornecem uma terceira dimensão que é mais comportamental, e contém informações sobre os caminhos potenciais de navegação que não são fornecidos pelos casos de uso ou pelos diagramas de seqüência.

Quando todas as três dimensões são usadas, omissões e inconsistências que poderiam escapar de serem detectadas em uma dimensão tornam-se óbvias quando uma segunda (ou terceira) dimensão é examinada. Essa é a razão pela qual WebApps grandes e complexas podem se beneficiar de um modelo de interação que engloba todas as três representações.

AVISO

Alguns engenheiros da Web preferem um esboço com lápis e papel das principais páginas (telas) da WebApp. Embora tais esboços possam ser desenvolvidos muito rapidamente, o fluxo de navegação é menos óbvio do que com um protótipo operacional.

Protótipo de interface com o usuário. O leiaute da interface com o usuário, o conteúdo que apresenta, os mecanismos de interação que implementa e a estética global das conexões usuário-WebApp têm muito a ver com a satisfação do usuário e a aceitação global da WebApp. Embora possa ser argumentado que a criação de um protótipo de interface com o usuário seja uma atividade de projeto, é uma boa idéia realizá-la durante a criação do modelo de análise. Tão logo uma representação física da interface com o usuário possa ser revisada, é alta a probabilidade de que os usuários finais obtenham o que eles querem. Análise e projeto de interface com o usuário são discutidos em detalhe no Capítulo 12.

Como as ferramentas de desenvolvimento de WebApps são abundantes, relativamente baratas e funcionalmente poderosas, é melhor criar um protótipo de interface usando tais ferramentas. O protótipo deve implementar as principais ligações navegacionais e representar um leiaute de tela global do mesmo modo que será construído.

18.5 O MODELO FUNCIONAL

O *modelo funcional* atende a dois elementos de processamento da WebApp, cada um representando um diferente nível de abstração procedimental: (1) funcionalidade observável pelo usuário que é entregue pela WebApp aos usuários finais, e (2) as operações contidas nas classes de análise que implementam comportamentos associados com a classe.

A funcionalidade observável pelo usuário engloba quaisquer funções de processamento que são iniciadas diretamente pelo usuário. Por exemplo, um site financeiro da Web poderia implementar uma variedade de funções financeiras (por exemplo, um calculador de economia nos pagamentos de universidade ou um calculador de economias de aposentadoria). Essas funções podem ser realmente implementadas usando operações dentro de classes de análise, mas, do ponto de vista do usuário final, a função (mais corretamente, os dados fornecidos pela função) é a saída visível.

AVISO

Como uma alternativa, você pode também escrever uma simples narrativa de processamento ou representação em linguagem de projeto de programa (Capítulo 11). No entanto, muitas pessoas preferem uma representação gráfica.

Em um nível mais baixo de abstração procedimental, o modelo de análise descreve o processamento a ser realizado pelas operações da classe de análise. Essas operações manipulam atributos de classe e são envolvidas à medida que as classes colaboram umas com as outras para conseguir algum comportamento necessário.

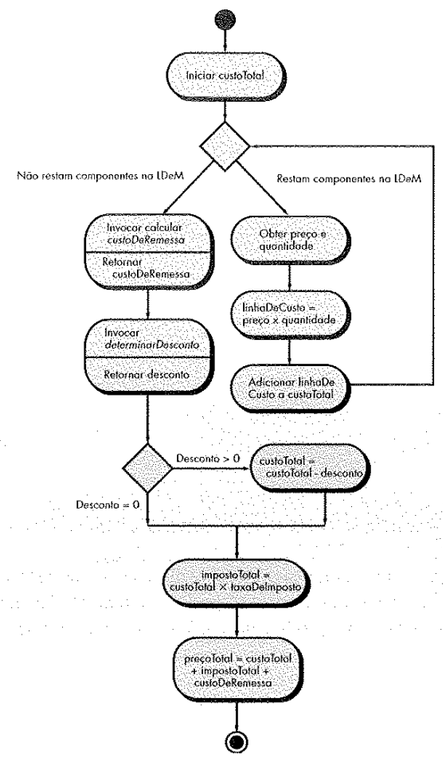
Independentemente do nível de abstração procedimental, o diagrama de atividade UML pode ser usado para representar detalhes de processamento. A Figura 18.7 mostra um diagrama de atividade para a operação *calcularPreço()* que é parte da classe de análise[6] **ListaDeMateriais**. Como mencionamos no Capítulo 8, o diagrama de atividade é similar ao fluxograma, ilustrando o fluxo de processamento e as decisões lógicas com o fluxo. Deve ser observado que duas operações adicionais são invocadas no fluxo procedimental: *calcularCustoDeRemessa()*, que calcula o custo de remessa dependendo do método de remessa escolhido pelo cliente, e *determinarDesconto()*, que determina qualquer desconto especial para os componentes do *CasaSegura* que foram selecionados para compra. Os detalhes de construção indicando como essas operações são invocadas e os detalhes de interface para cada operação não são considerados até que o projeto da WebApp comece.

6 Uma revisão da classe de análise **ListaDeMateriais** poderia determinar que no interesse da coesão, a operação *calcularPreço()* pode ser melhor colocada dentro de uma classe **Fatura**. Essa sugestão tem mérito. No entanto, ela permanece na classe de análise **ListaDeMateriais** para o propósito desse exemplo.

Document shared on www.docsity.com

FIGURA 18.7

Diagrama de atividade para a operação *calcularPreço()*



18.6 O Modelo de Configuração

AVISO

Apesar de ser muito importante considerar todas as configurações em que é provável que sejam usadas, lembre-se de que uma WebApp precisa ser construída para servir a seus usuários finais, não às idiossincrasias de um navegador particular.

WebApps devem ser projetadas e implementadas de modo que acomodem uma variedade de ambientes tanto do lado do servidor quanto do lado do cliente[7]. A WebApp pode residir em um servidor que fornece acesso via Internet, Intranet ou Extranet. O hardware do servidor e o ambiente do sistema operacional devem ser especificados. Além disso, considerações de interoperabilidade do lado do servidor devem ser levadas em conta. Se a WebApp precisar ter acesso a um grande banco de dados ou interoperar com aplicações da empresa que existem no lado do servidor, interfaces apropriadas, protocolos de comunicação e informação colaborativa relacionada devem ser especificadas.

Software do lado do cliente fornece a infra-estrutura que possibilita acesso à WebApp a partir da localização do usuário. Em geral, software de navegação é usado para entregar o conteúdo e a funcionalidade da WebApp que é baixada do servidor. Embora existam normas, cada navegador tem suas próprias peculiaridades. Por essa razão, a WebApp deve ser rigorosamente testada com cada configuração de navegador que é especificada como parte do *modelo de configuração*.

7 O *lado do servidor* hospeda a WebApp e todas as características relacionadas do sistema que possibilitam a vários usuários terem acesso à WebApp via rede de computador. O *lado do cliente* fornece um ambiente de software (por exemplo, navegadores) que possibilita aos usuários finais interagir com a WebApp por meio do computador do usuário.

Em alguns casos, o modelo de configuração não é nada mais do que uma lista de atributos do lado do servidor e do lado do cliente. No entanto, para WebApps mais complexas, uma variedade de detalhes de configuração (por exemplo, distribuição de carga entre vários servidores, arquiteturas de cache, bancos de dados remotos, vários servidores servindo a vários objetos da mesma página Web) pode ter impacto sobre a análise e o projeto. O diagrama de implantação UML (Capítulo 10) pode ser usado em situações nas quais arquiteturas de configuração complexas devem ser consideradas.

18.7 Análise de Relacionamento-Navegação

Os elementos do modelo de análise descritos nas seções anteriores identificam elementos de conteúdo e funcionais, além do modo pelo qual eles são usados, para implementar a interação do usuário. À medida que a análise evolui para projeto, esses elementos tornam-se parte da arquitetura da WebApp. No contexto de aplicações Web, cada elemento arquitetural tem o potencial de ser ligado a todos os outros elementos arquiteturais. No entanto, como o número de ligações aumenta, a complexidade navegacional através da WebApp também aumenta. A questão, então, é como estabelecer as ligações adequadas entre objetos de conteúdo e as funções que fornecem as habilidades requeridas pelo usuário.

> "[Navegação] não é somente a ação de pular de página para página, mas a idéia de mover-se em um espaço de informação."
>
> A. Reina e J. Torres

Análise relacionamento-navegação (Relationship-navigation analysis-RNA) fornece uma série de passos de análise que buscam identificar relacionamentos entre os elementos descobertos como parte da criação do modelo de análise.[8] Yoo and Bieber [YOO00] descrevem RNA da seguinte maneira:

> RNA fornece aos analistas de sistemas uma técnica sistemática para determinação da estrutura de relacionamento de uma aplicação, ajudando-os a descobrir todos os relacionamentos potencialmente úteis nos domínios de aplicação. Esses, posteriormente, podem ser implementados como links. RNA também ajuda a determinar estruturas navegacionais adequadas sobre esses links. RNA melhora o entendimento dos desenvolvedores de sistema sobre domínios de aplicação, ampliando e aprofundando seu modelo conceitual do domínio. Desenvolvedores podem então melhorar sua implementação pela inclusão de links, metainformação e navegação adicionais.

A abordagem RNA está organizada em cinco passos:

- *Análise de interessados* — identifica as várias categorias de usuários (como descrito na Seção 18.1) e estabelece uma hierarquia adequada de interessados.
- *Análise de elementos* — identifica os objetos de conteúdo e elementos funcionais que são de interesses para os usuários finais (como descrito nas Seções 18.3 e 18.5).
- *Análise de relacionamentos* — descreve os relacionamentos que existem entre os elementos da WebApp.
- *Análise da navegação* — examina como os usuários podem ter acesso a elementos individuais ou grupos de elementos.
- *Análise de avaliação* — considera tópicos pragmáticos (por exemplo, custo/benefício) associado com a implementação dos relacionamentos definidos anteriormente.

Os dois primeiros passos da abordagem RNA foram discutidos anteriormente neste capítulo. Nas seções seguintes, consideraremos métodos para estabelecer os relacionamentos que existem entre objetos de conteúdo e funções.

8 Deve-se notar que RNA pode ser aplicada a qualquer sistema de informação e foi originalmente desenvolvida para sistemas de hipermídia em geral. Pode, no entanto, ser facilmente adaptada para engenharia da Web.

Document shared on www.docsity.com

18.7.1 Análise de Relacionamentos — Questões-chave

Yoo and Bieder [YOO00] sugerem uma lista de questões a que um engenheiro da Web ou analista de sistemas deveria responder sobre cada elemento (objeto de conteúdo ou função) que tenha sido identificado no modelo de análise. A seguinte lista, adaptada para WebApps, é representativa [YOO00]:

? Como fazemos avaliação dos elementos do modelo de análise para entender o relacionamento entre eles?

- O elemento é membro de uma categoria mais ampla de elementos?
- Quais atributos ou parâmetros foram identificados para o elemento?
- A informação descritiva sobre o elemento já existe? Em caso afirmativo, onde está?
- O elemento aparece em diferentes localizações dentro da WebApp? Em caso afirmativo, onde?
- O elemento é composto de outros elementos menores? Em caso afirmativo, quais são eles?
- O elemento é membro de uma coleção maior de elementos? Em caso afirmativo, qual é ela e qual é a sua estrutura?
- O elemento é descrito pela classe de análise?
- Outros elementos são similares ao elemento que está sendo considerado? Em caso afirmativo, é possível que eles fossem combinados em um elemento?
- O elemento é usado em uma ordenação específica de outros elementos? Sua aparência depende de outros elementos?
- Um outro elemento sempre segue a aparência do elemento sob consideração?
- Quais pré e pós-condições devem ser satisfeitas para que o elemento seja usado?
- Categorias específicas de usuário usam o elemento? Diferentes categorias de usuário usam o elemento de forma diferente? Em caso afirmativo, como?
- Pode o elemento ser associado com uma meta ou objetivo específico de formulação? Com um requisito específico da WebApp?
- Esse elemento sempre aparece na mesma hora em que outros elementos aparecem? Em caso afirmativo, quais são os outros elementos?
- Esse elemento sempre aparece no mesmo lugar (por exemplo, mesma localização de tela ou página) que outros elementos aparecem? Em caso afirmativo, quais são os outros elementos?

As respostas a essas e outras questões ajudam o engenheiro da Web a posicionar o elemento em questão dentro da WebApp e estabelecer relacionamentos entre os elementos.

É possível desenvolver uma taxonomia de relacionamento e categorizar cada relacionamento identificado à medida que um resultado das questões for notado. O leitor interessado deve ler [YOO00] para mais detalhe.

18.7.2 Análise de Navegação

Uma vez que as relações foram desenvolvidas entre os elementos definidos dentro do modelo de análise, o engenheiro da Web deve considerar os requisitos que ditam como cada categoria de usuário vai navegar de um elemento (por exemplo, objeto de conteúdo) para um outro. Os mecanismos de navegação são definidos como parte do projeto. Nesse estágio, desenvolvedores deveriam considerar requisitos globais de navegação. As seguintes questões devem ser formuladas e respondidas:

- Certos elementos deveriam ser mais fáceis de atingir (requerem menos passos de navegação) do que outros? Qual é a prioridade para a apresentação?

? Que questões deveriam ser formuladas para melhor entender requisitos de navegação?

- Certos elementos deveriam ser realçados para forçar os usuários a navegar na sua direção?
- Como os erros de navegação deveriam ser manipulados?
- Navegação para grupos relacionados de elementos deve ter prioridade sobre navegação para um elemento específico?
- Navegação deveria ser conseguida por meio de vínculos (links), acesso baseado em busca, ou algum outro meio?
- Certos elementos deveriam ser apresentados aos usuários com base no contexto das ações de navegação anteriores?
- Um registro de navegação deveria ser mantido para os usuários?
- Um mapa ou menu completo da navegação (em oposição a um único link de "retorno" ou ponteiro direcionado) deveria estar disponível em cada ponto de interação com um usuário?
- O projeto de navegação deveria ser guiado pelos comportamentos de usuário mais comumente esperados ou pela importância dos elementos definidos da WebApp?
- Um usuário pode "armazenar" sua navegação anterior por meio da WebApp para acelerar uso futuro?
- Para que categoria de usuários deve ser projetada a navegação ótima?
- Como links externos à WebApp deveriam ser manipulados? Sobrepostos à janela de navegação existente? Em uma nova janela de navegação? Em um quadro (*frame*) separado?

AVISO

Quando você analisar os requisitos navegacionais, lembre-se de que o usuário deve sempre saber onde está e para onde pode ir. Para fazer isso, o usuário precisa de um "mapa".

Essas e muitas outras questões deveriam ser formuladas e respondidas como parte da análise de navegação.

A equipe de engenharia da Web e seus interessados devem também determinar os requisitos globais para navegação. Por exemplo, um "mapa do site" será fornecido para dar aos usuários uma visão geral da estrutura de toda WebApp? Um usuário pode fazer um "passeio guiado" que realce os elementos (objetos de conteúdo e funções) mais importantes que estão disponíveis? Um usuário será capaz de ter acesso a objetos de conteúdo ou funções com base em atributos definidos para aqueles elementos (por exemplo, um usuário pode desejar ter acesso a todas as fotografias de um prédio específico ou a todas as funções que permitam cálculo de peso).

18.8 Resumo

Formulação, coleta de requisitos e modelagem de análise são realizadas como parte da análise de requisitos das WebApps. O objetivo dessas atividades é (1) descrever a motivação básica (metas) e os objetivos da WebApp; (2) definir as categorias de usuários; (3) observar o conteúdo e requisitos funcionais da WebApp; e (4) estabelecer um entendimento básico do porquê a WebApp deve ser construída, quem vai usá-la e que problema(s) ela vai resolver para seus usuários.

Casos de uso são o catalisador de todas as atividades de análise e modelagem de requisitos. Casos de uso podem ser organizados em pacotes funcionais e cada pacote é avaliado para garantir que seja abrangente, coesivo, fracamente acoplado e hierarquicamente raso.

Quatro atividades de análise contribuem para a criação de um modelo completo de análise: análise de conteúdo identifica o espectro completo de conteúdo a ser fornecido para a WebApp; análise de interação descreve o modo pelo qual o usuário interage com a WebApp; análise funcional define as operações que serão aplicadas ao conteúdo da WebApp e descreve outras funções de processamento independentes do conteúdo, mas necessárias para o usuário final, e análise de configuração descreve o ambiente e a infra-estrutura nos quais a WebApp reside.

Document shared on www.docsity.com

O modelo de conteúdo descreve o espectro de objetos de conteúdo que devem ser incorporados à WebApp. Esses objetos de conteúdo devem ser desenvolvidos ou adquiridos para integração na arquitetura da WebApp. Uma árvore de dados pode ser usada para representar uma hierarquia de objetos de conteúdo. Classes de análise (derivadas de casos de uso) fornecem outro meio de representar objetos-chave que a WebApp vai manipular.

O modelo de interação é construído com casos de uso, diagramas de seqüência UML e diagramas de estado UML para descrever a "conversação" entre o usuário e a WebApp. Além disso, um protótipo de interface pode ser construído para apoiar o desenvolvimento de leiaute e requisitos de navegação.

O modelo funcional descreve funções observáveis pelo usuário e operações de classe usando o diagrama de atividade UML. O modelo de configuração descreve o ambiente do sistema que a WebApp vai necessitar tanto do lado do servidor quanto do lado do cliente.

Análise de relacionamento-navegação identifica relacionamentos entre o conteúdo e elementos funcionais definidos no modelo de análise e estabelece requisitos para definir vínculos (links) de navegação adequadas ao longo do sistema. Uma série de questões ajuda a estabelecer relacionamentos e a identificar características que terão influência no projeto de navegação.

Referências Bibliográficas

[CON00] Conallen, J., *Building Web Applications with UML*, Addison-Wesley, 2000.

[FRA01] Franklin, S., "Planning Your Web Site with UML", *webReview*, disponível em http://www.webreview.com/2001/05_18/developers/index01.shtml.

[SRI01] Sridhar, M. e Mandyam, N., "Effective Use of Data Models in Building Web Applications", 2001, disponível em http://www2002.org/CDROM/alternate/698/.

[YOO99] Yoo, J. e Bieber, M., "A Systematic Relationship Analysis for Modeling Information Domains", 1999, disponível em http://citeseer.nj.nec.com/ 312025. html.

[YOO00] _____, "Toward a Relationship Navigation Analysis", *Proc. 33rd Hawaii Conf. On System Sciences*, v. 6., IEEE, jan. 2000, disponível em www.cs.njit.edu/_bieber/pub/hicss00/INWEB02.pdf.

Problemas e Pontos a Considerar

18.1. Usando a enorme gama de recursos de desenvolvimento ágil de software disponível na Web, faça uma pequena pesquisa e elabore uma argumentação contra a modelagem de análise de WebApps. Você acredita que sua argumentação se aplica a todos os casos?

18.2. Se você fosse forçado a fazer "um pouco de modelagem de análise" — isto é, modelagem de análise mínima —, que representações, diagramas e informação poderia definir durante essa atividade de engenharia da Web?

18.3. Usando um diagrama similar ao mostrado na Figura 18.1, estabeleça uma hierarquia de usuário para (a) site Web para serviços financeiros ou (b) site Web para um vendedor de livros.

18.4. O que um pacote de caso de uso representa?

18.5. Casos de uso ou pacotes de caso de uso são avaliados para garantir que sejam *compreensíveis, coesivos, fracamente acoplados e hierarquicamente raso*. Descreva o que esses termos significam com as suas próprias palavras.

Selecione uma WebApp que você visite regularmente com base em uma das seguintes categorias: (a) notícias ou esportes, (b) entretenimento, (c) e-commerce, (d) jogo, (e) relacionada a computador, (f) uma WebApp recomendada pelo seu professor. Execute as atividades mencionadas nos Problemas 18.6 a 18.12.

18.6. Desenvolva um ou mais casos de uso que descrevam o comportamento específico de usuário da WebApp.

18.7. Represente uma hierarquia de conteúdo parcial e defina no mínimo três classes de análise da WebApp.

18.8. Desenvolva um diagrama de seqüência UML e um diagrama de estado UML que descreva uma interação específica na WebApp.

18.9. Considere a interface de uma WebApp. Elabore um protótipo de uma modificação para a interface que você acredita que vá melhorá-la.

18.10. Selecione um usuário de uma função observável da WebApp e modele-o usando um diagrama de atividade UML.

18.11. Selecione um objeto de conteúdo ou função que seja parte da arquitetura da WebApp e responda às questões de relacionamento-navegação listadas na Seção 18.7.1.

18.12. Considere a WebApp existente, reponda às questões de relacionamento-navegação listadas na Seção 18.7.2.

Leituras e Fontes de Informação Adicionais

Muitos livros dedicados à modelagem de análise de software convencional — com particular ênfase em casos de uso e notação UML — contêm muita informação útil que pode ser prontamente adaptada por engenheiros da Web. Casos de uso formam a base para modelagem de análise de WebApps. Livros de Kulak e seus colegas *(Use Cases: Requirements in Context*, segunda edição, Addison-Wesley, 2004), Bittner e Spence *(Use Case Modeling*, Addison-Wesley, 2002), Cockburn *(Writing Effective Use Cases*, Addison-Wesley, 2001), Armour e Miller *(Advanced Use-Case Modeling: Software Systems*, Addison-Wesley, 2000), Rosenberg e Scott *(Use Case Driven Object Modeling with UML: A Practical Approach*, Addison-Wesley, 1999) e Schneider, Winters, e Jacobson *(Applying Use Cases: A Practical Guide*, Addison-Wesley, 1998) fornecem diretrizes que valem a pena para a criação e o uso desse importante mecanismo de representação de requisitos. Discussão que vale a pena da UML foi escrita por Arlow e Neustadt *(UML and the Unified Process*, Addison-Wesley, 2002), Schmuller *(Teach Yourself UML*, Sams Publishing, 2002), Booch e seus colegas *(The UML User Guide*, Addison-Wesley, 1998), Rumbaugh e seus colegas *(The Unified Modeling Language Reference Manual*, Addison-Wesley, 1998).

Livros dedicatos a projeto de sites da Web freqüentemente contêm um ou mais capítulos que discutem tópicos de análise (embora esses sejam freqüentemente superficiais). Os seguintes livros contêm um ou mais tópicos de análise no contexto de engenharia da Web: Van Duyne e seus colegas *(The Design of Sites*, Addison-Wesley, 2002), Rosenfeld e Morville *(Information Architecture for the World Wide Web*, O'Reilly & Associates, 2002), Wodtke *(Information Architecture*, New Riders Publishing, 2002), Garrett *(The Elements of User Experience: User Centered Design for the Web*, New Riders Publishing, 2002), Niederst *(Web Design in a Nutshell*, O'Reilly & Associates, 2001), Lowe e Hall *(Hypertext and the Web: An Engineering Approach*, Wiley, 1999) e Powell *(Web Site Engineering*, Prentice-Hall, 1998) fornecem cobertura razoavelmente completa. Norris, West e Watson *(Media Engineering: A Guide to Developing Information Products*, Wiley, 1997), Navarro e Khan *(Effective Web Design: Master the Essentials*, Sybex, 1998) e Fleming e Koman *(Web Navigation: Designing the User Experience*, O'Reilly & Associates, 1998) fornecem diretrizes adicionais para análise e projeto.

Uma ampla variedade de fontes de informação sobre modelagem de análise de engenharia da Web está disponível na Internet. Uma lista atualizada de referências da World Wide Web pode ser encontrada sobre "recursos de engenharia de software" no site deste livro: **http://www.mhhe.com/pressman.**

Document shared on www.docsity.com

CAPÍTULO 19

MODELAGEM DE PROJETO PARA APLICAÇÕES WEB

CONCEITOS-CHAVE

arquitetura de conteúdo .. 441
arquitetura MVC 443
atributos de qualidade 427
métricas 450
OOHDM 447
padrões 446
projeto de arquitetura 440
projeto de conteúdo 439
projeto estético 437
projeto de interface 431
projeto de navegação 444
projeto no nível de componente 446

Em seu livro referência sobre projeto na Web, Jakob Nielsen [NIE00] declara: "Há essencialmente duas abordagens básicas de projeto: o ideal artístico de auto-expressão e o ideal de engenharia para resolver um problema para o cliente". Durante a primeira década de desenvolvimento na Web, a idéia artística foi a abordagem que muitos desenvolvedores escolheram. O projeto ocorreu de modo *ad hoc* e foi usualmente conduzido à medida que HTML era gerado, e evoluiu com base em uma visão artística que, por sua vez, evoluiu à medida que construções WebApp ocorreram.

Mesmo hoje, os proponentes mais "extremados" de desenvolvimento ágil de software (Capítulo 4) usam aplicações Web como um modelo de propaganda para "projeto limitado". Eles afirmam que o imediatismo e a volatilidade da WebApp depõem contra projeto formal, que o projeto evolui à medida que uma aplicação é construída (codificada), e que relativamente pouco tempo deveria ser gasto na criação de um modelo de projeto detalhado. Esse argumento tem mérito, mas somente para WebApps relativamente simples. Quando conteúdo e função são complexos, quando o tamanho da WebApp engloba centenas de objetos de conteúdo, funções e classes de análise, quando o sucesso da WebApp terá um impacto direto no sucesso do negócio, o projeto não pode e não deve ser feito levianamente.

Essa realidade leva-nos à segunda abordagem de Nielsen — "o ideal da engenharia é resolver um problema para um cliente". A engenharia da Web adota essa filosofia, e uma abordagem mais rigorosa para projeto de WebApp capacita o desenvolvedor a ser bem-sucedido.

PANORAMA

O que é? Projeto de WebApps engloba atividades técnicas e não técnicas. A aparência do conteúdo é desenvolvida como parte do projeto gráfico, o leiaute de estética da interface com o usuário é criado como parte do projeto de interface, e a estrutura técnica da WebApp é modelada como parte do projeto arquitetural e navegacional. Em cada instância, um modelo de projeto deve ser criado antes de a construção começar, mas um bom engenheiro da Web reconhece que o projeto vai evoluir muito à medida que mais é aprendido sobre os requisitos dos interessados conforme a WebApp é construída.

Quem faz? Engenheiros da Web, projetistas gráficos, desenvolvedores de conteúdo e outros interessados, todos participam da criação de um modelo de projeto para engenharia da Web.

Por que é importante? O projeto permite ao engenheiro da Web criar um modelo que pode ser avaliado quanto à qualidade e melhorado antes que o conteúdo e o código sejam gerados, testes sejam conduzidos e grande número de usuários finais se envolvam. O projeto é o lugar em que a qualidade da WebApp é estabelecida.

Quais são os passos? Projeto da WebApp engloba seis passos principais, guiados pela informação obtida durante a modelagem de análise. Projeto de conteúdo usa a informação contida no modelo de análise como base para estabelecer o projeto dos objetos de conteúdo e seus relacionamentos. Projeto de estética (também chamado de projeto gráfico) estabelece a aparência que o usuário final vê. Projeto arquitetural focaliza a estrutura global de hipermídia de todos os objetos de conteúdo e funções. Projeto de interface estabelece o leiaute global e mecanismos de interação que definem a interface com o usuário. Projeto de navegação define como o usuário final navega pela estrutura de hipermídia, e projeto de componentes representa a estrutura interna detalhada dos elementos funcionais da WebApp.

Qual é o produto de trabalho? Um modelo de projeto que engloba conteúdo, estética, arquitetura, interface, navegação e tópicos de projeto no nível de componente é o produto de trabalho principal do projeto de engenharia da Web.

Como tenho certeza de que fiz corretamente? Cada elemento do modelo de projeto é revisado pela equipe de engenharia da Web (e interessados selecionados) em um esforço para descobrir erros, inconsistências ou omissões. Além disso, soluções alternativas são consideradas e o grau em que o atual modelo de projeto vai levar a uma implementação efetiva é também avaliado.

19.1 TÓPICOS DE PROJETO PARA ENGENHARIA DA WEB

Quando projeto é aplicado no contexto de engenharia da Web, tanto tópicos genéricos quanto específicos devem ser considerados. Do ponto de vista genérico, ele resulta em um modelo que orienta a construção da WebApp. O modelo de projeto, independentemente de sua forma, deve conter informação suficiente para refletir como os requisitos do interessado (definidos no modelo de análise) devem ser traduzidos em conteúdo e código executável. Mas, o projeto deve também ser específico; deve atender a atributos-chave de uma WebApp de modo que habilite um engenheiro da Web a construir e testar efetivamente.

19.1.1 Projeto e Qualidade da WebApp

Em capítulos anteriores, mencionamos que projeto é uma atividade de engenharia que leva a um produto de alta qualidade. Isso nos traz a uma questão recorrente encontrada em todas as disciplinas de engenharia: o que é qualidade? Nesta seção examinamos a resposta no contexto de engenharia da Web.

Toda pessoa que tenha navegado na Web ou usado uma Intranet empresarial tem uma opinião sobre o que torna uma WebApp "boa". Pontos de vista individuais variam amplamente. Alguns usuários apreciam gráficos espalhafatosos, outros querem texto simples. Alguns exigem informação abundante, outros querem uma apresentação resumida. Alguns gostam de ferramentas analíticas sofisticadas ou acesso a banco de dados, outros gostam de manter as coisas simples. De fato, a percepção do usuário de excelência (e a aceitação ou rejeição resultante da WepApp como conseqüência) pode ser mais importante que qualquer discussão técnica de qualidade de WebApp.

Contudo, como é percebida a qualidade da WebApp? Quais atributos devem ser exibidos para atingir a excelência aos olhos dos usuários finais e ao mesmo tempo exibir características técnicas de qualidade que vão habilitar um engenheiro da Web a corrigir, adaptar, melhorar e manter a aplicação em longo prazo?

Na realidade, todas as características gerais de qualidade de software discutidas nos Capítulos 9, 15 e 26 se aplicam às WebApps. No entanto, as mais relevantes dessas características — usabilidade, funcionalidade, confiabilidade, eficiência e manutenibilidade — fornecem uma base útil para avaliar a qualidade de sistemas baseados na Web.

> "Se produtos são projetados para melhor acomodar as tendências naturais do comportamento humano, então as pessoas vão ficar mais satisfeitas, mais felizes e mais produtivas."
>
> Susan Weinschenk

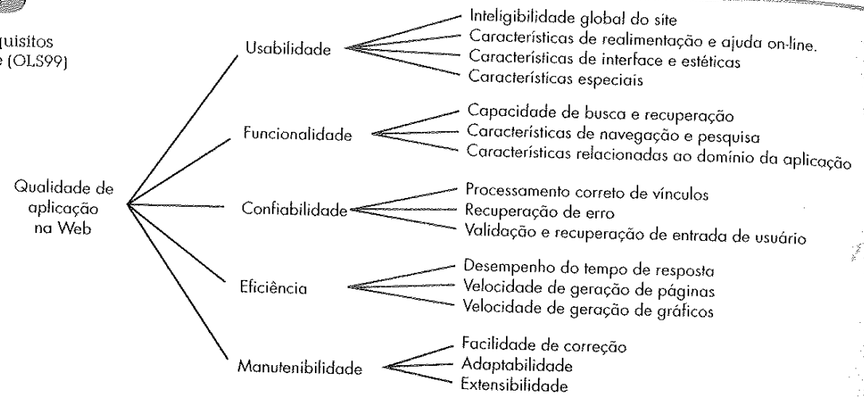
Olsina e seus colegas [OLS99] prepararam uma "árvore de requisitos de qualidade" que identifica um conjunto de atributos técnicos — usabilidade, funcionalidade, confiabilidade, eficiência, manutenibilidade — que levam à alta qualidade de WebApps.[1] A Figura 19.1 resume seu trabalho. Os critérios observados na figura são de particular interesse para engenheiros da Web que precisam projetar, construir e manter WebApps no longo prazo.

1 Esses atributos de qualidade são similares àqueles apresentados nos Capítulos 9, 15 e 26. Conclusão: características de qualidade são universais para todo software.

Document shared on www.docsity.com

FIGURA 19.1

Árvore de requisitos de qualidade (OLS99)



Offutt [OFF02] estende os cinco principais atributos de qualidade observados na Figura 19.1 adicionando os seguintes atributos:

? Quais são os principais atributos de qualidade de WebApps?

Segurança. WebApps tornaram-se fortemente integradas com banco de dados empresariais e governamentais críticos. Aplicações de e-comércio extraem e depois armazenam informação confidencial de clientes. Por essas e muitas outras razões, segurança para a WebApp é vital em muitas situações. A medida-chave de segurança é a capacidade de a WebApp e seu ambiente servidor rechaçarem acesso não autorizado e/ou rebaterem ataques mal-intencionados. Uma discussão detalhada de segurança de WebApp está além do escopo deste livro. O leitor interessado deve consultar [MCC01], [NOR02] ou [KAL03].

Disponibilidade. Mesmo a melhor WebApp não vai satisfazer às necessidades dos usuários se não estiver disponível. No sentido técnico, disponibilidade é a medida da porcentagem de tempo que uma WebApp está disponível para uso. O usuário final típico espera que WebApps estejam disponíveis 24/7/365. Qualquer coisa menor é considerada inaceitável.[2] Mas "no ar" não é o único indicador de disponibilidade. Offutt [OFF02] sugere que "características de uso disponíveis em apenas um navegador ou uma plataforma" tornam a WebApp não disponível para aqueles que estejam com uma configuração de navegador/plataforma diferente. O usuário vai invariavelmente embora.

Escalabilidade. A WebApp e seu ambiente servidor podem ser escalados para atender 100, 1000, 10.000 ou 100.000 usuários? A WebApp e os sistemas com os quais ela estabelece uma interface vão poder atender variação significativa de volume, ou a capacidade de resposta vai cair drasticamente (ou cessar totalmente)? Não é suficiente construir uma WebpApp bem-sucedida. É igualmente importante construir uma WebApp que possa acomodar as responsabilidades do sucesso (significativamente mais usuários finais) e tornar-se ainda mais bem-sucedida.

Prazo de colocação no mercado. Embora o prazo de colocação no mercado não seja um atributo real de qualidade no sentido técnico, é uma medida de qualidade do ponto de vista do negócio. A primeira WebApp colocada no mercado freqüentemente capta um número desproporcional de usuários finais.

Bilhões de páginas da Web estão disponíveis para aqueles que buscam informação na World Wide Web. Mesmo buscas na Web bem direcionadas resultam em uma avalanche de conteúdo. Com tantas fontes de informação para escolher, como o usuário avalia a qualidade (por exemplo, veracidade, precisão, completeza, oportunidade) do conteúdo que é apresentado em uma WebApp? Tillman [TIL00] sugere um conjunto útil de critérios para avaliar a qualidade do conteúdo:

? O que deve ser considerado quando se avalia qualidade de conteúdo?

- O escopo e a profundidade do conteúdo podem facilmente ser determinados para garantir que atendam às necessidades do usuário?
- A autoridade e a experiência dos autores do conteúdo podem ser facilmente identificadas?
- É possível determinar a atualidade do conteúdo, a última atualização e o que foi atualizado?
- O conteúdo e sua localização são estáveis (isto é, vão permanecer na URL referida)?

Além dessas questões relacionadas ao conteúdo, as seguintes poderiam ser adicionadas:

- O conteúdo é confiável?
- O conteúdo é exclusivo, isto é, a WebApp fornece algum benefício exclusivo àqueles que a utilizam?
- O conteúdo tem valor para a comunidade de usuários visada?
- O conteúdo é bem organizado? Indexado? De fácil acesso?

As *checklists* mencionadas nesta seção representam apenas um pequeno exemplo dos tópicos que deveriam ser tratados à medida que o projeto de uma WebApp evolui. Uma importante meta da engenharia da Web é desenvolver sistemas nos quais respostas afirmativas são fornecidas para todas as questões relacionadas à qualidade.

> "Só porque você pode não significa que você deve."
>
> Jean Kaiser

19.1.2 Metas de Projeto

Em sua coluna regular sobre projeto na Web, Jean Kaiser [KAI02] sugere as seguintes metas de projeto aplicáveis a virtualmente todas WebApps independentemente do domínio de aplicação, tamanho ou complexidade:

2 Essa expectativa é, sem dúvida, irreal. As principais WebApps precisam programar períodos "fora do ar" para reparos e aperfeiçoamentos.

INFO

Checklist de Qualidade de Projeto de WebApp

A seguinte lista de verificação, ou *checklist*, adaptada da informação apresentada em *Webreference.com*, fornece um conjunto de questões que vão ajudar tanto engenheiros da Web quanto usuários finais a avaliar a qualidade global de uma WebApp:

- As opções de conteúdo e/ou função e/ou navegação podem ser personalizadas para as preferências do usuário?
- O conteúdo e/ou funcionalidade podem ser personalizados para a largura de banda na qual o usuário se comunica?
- Gráficos e outras mídias não textuais foram usados adequadamente? Os tamanhos de arquivos gráficos estão otimizados para eficiência de exibição?
- As tabelas foram organizadas e colocadas em tamanho adequado que permitem que elas sejam entendidas e exibidas eficientemente?
- O HTML foi otimizado para eliminar ineficiências?
- O projeto global da página é fácil de ler e navegar?
- Todos os ponteiros (vínculos) fornecem links a informação que é de interesse dos usuários?
- É provável que a maioria dos links tenha persistência na Web?
- A WebApp está instrumentada com utilitários de gestão de site que incluem ferramentas para rastreamento de uso, teste de vínculo, busca local e segurança?

Document shared on www.docsity.com

Simplicidade. Embora possa parecer antiquado, o provérbio "todas as coisas com moderação" se aplica a WebApps. Há uma tendência entre alguns projetistas de fornecer "em excesso" ao usuário final — conteúdo exaustivo, recursos visuais em demasia, animação inoportuna, páginas enormes, a lista é longa. É melhor buscar moderação e simplicidade.

Consistência. Essa meta de projeto aplica-se a virtualmente todo elemento do modelo de projeto. O conteúdo deve ser construído consistentemente (por exemplo, formatação de texto e estilos de fonte devem ser os mesmos ao longo de todos os documentos de texto, arte gráfica deve ter uma aparência, esquema de cor e estilo consistentes). O projeto gráfico (estética) deve apresentar uma aparência consistente ao longo de todas as partes da WebApp. O projeto arquitetural deve estabelecer gabaritos que levem a uma estrutura de hipermídia consistente. O projeto de interface deve definir modos de interação, navegação e exibição de conteúdo consistentes. Mecanismos de navegação devem ser usados consistentemente ao longo de todos os elementos da WebApp.

Identidade. O projeto de estética, de interface e navegacional de uma WebApp deve ser coerente com o domínio de aplicação para o qual ela está sendo construída. Um site Web para um grupo de *hip-hop* vai indubitavelmente ter um aspecto diferente do que uma WebApp projetada para uma empresa de serviços financeiros. A arquitetura da WebApp será inteiramente diferente, interfaces vão ser construídas para acomodar diferentes categorias de usuários, a navegação será organizada para atingir diferentes objetivos. Um engenheiro da Web (e outros contribuintes do projeto) devem trabalhar para estabelecer uma identidade para a WebApp ao longo do projeto.

Robustez. Com base na identidade estabelecida, uma WebApp freqüentemente faz uma "promessa" implícita para o usuário; ele espera conteúdo e funções robustos que são relevantes para as suas necessidades. Se esses elementos forem omitidos ou insuficientes, é provável que a WebApp falhará.

Navegabilidade. Já observamos que a navegação deve ser simples e consistente. Deve, também, ser projetada de modo que seja intuitiva e previsível. Deve estar claro ao usuário como se movimentar pela WebApp sem ter de procurar links ou instruções de navegação.

Atração visual. De todas as categorias de software, aplicações da Web são inquestionavelmente as mais visuais, as mais dinâmicas e as mais estéticas. Beleza (atração visual) está sem dúvida no olho do cliente, mas muitas características de projeto (por exemplo, o aspecto do conteúdo, leiaute da interface, coordenação de cores, proporção de texto, gráficos e outras mídias, mecanismos de navegação) contribuem efetivamente para a atração visual.

Compatibilidade. Uma WebApp será usada em uma variedade de ambientes (por exemplo, diferentes hardware, tipos de conexão com a Internet, sistemas operacionais, navegadores) e deve ser projetada para ser compatível com cada um.

> "Para alguns, um projeto na Web enfoca o aspecto visual... para outros, projeto na Web é sobre a estruturação da informação e navegação ao longo do espaço do documento. Outros podem até considerar projeto da Web como a tecnologia usada para construir aplicações interativas na Web. Na realidade, projeto inclui todas essas coisas e talvez mais."
>
> Thomas Powell

19.2 A Pirâmide de Projeto da WebE

PONTO CHAVE

WebE abrange seis tipos diferentes de projeto. Cada um contribui para a qualidade global da WebApp.

O que é projeto no contexto de engenharia da Web? Essa simples questão é mais difícil de responder do que se possa imaginar. O projeto leva a um modelo que contém a combinação adequada de estética, conteúdo e tecnologia. A combinação varia dependendo da natureza da WebApp e, como conseqüência, as atividades do projeto que são enfatizadas também variam.

A Figura 19.2 mostra uma pirâmide de projeto para engenharia da Web. Cada nível da pirâmide representa uma das seguintes atividades de projeto:

- *Projeto de interface* — descreve a estrutura e organização da interface com o usuário. Inclui uma representação do leiaute de tela, uma definição dos modos de interação e uma descrição dos mecanismos de navegação.

FIGURA 19.2

A pirâmide de projeto da WebE

- *Projeto estético* — também chamado de *projeto gráfico*, descreve o "aspecto" da WebApp. Inclui esquemas de cor, leiaute geométrico, tamanho, fonte e colocação de texto, uso de gráficos e decisões estéticas relacionadas.
- *Projeto de conteúdo* — define o leiaute, a estrutura e o esboço de todo o conteúdo que é apresentado como parte da WebApp. Estabelece os relacionamentos entre objetos de conteúdo.
- *Projeto de navegação* — representa o fluxo de navegação entre objetos de conteúdo e todas as funções da WebApp.
- *Projeto arquitetural* — identifica a estrutura de hipermídia para a WebApp.
- *Projeto de componente* — desenvolve a lógica de processamento detalhada necessária para implementar os componentes funcionais.

Cada uma dessas atividades de projeto é considerada em mais detalhe nas seções seguintes.

19.3 Projeto de Interface de WebApp[3]

Toda interface de usuário — quer seja projetada para uma WebApp, uma aplicação de software tradicional, um produto de consumo ou um dispositivo industrial — deve exibir as seguintes características: fácil de usar, fácil de aprender, fácil de navegar, intuitiva, consistente, eficiente, livre de erros e funcional. Deve fornecer ao usuário final experiência satisfatória e compensadora. Os conceitos, princípios e métodos de projeto de interface oferecem ao engenheiro da Web as ferramentas necessárias para conseguir essa lista de atributos.

No Capítulo 12, mencionamos que projeto de interface começa não com uma consideração de tecnologia, mas com um exame cuidadoso do usuário final. Durante a modelagem de análise para engenharia da Web (Capítulo 18), uma hierarquia de usuários foi desenvolvida. Cada categoria de usuário pode ter necessidades sutilmente diferentes, querer interagir com a WebApp de diferentes modos, e necessitar funcionalidade e conteúdo exclusivo. Essa informação é derivada durante a análise de requisitos, mas é revisitada como o primeiro passo do projeto de interface.

> "Se um site é perfeitamente utilizável, mas não tem um estilo de projeto elegante e adequado, ele falhará."
>
> Curt Cloninger

3 A maioria, senão todas as diretrizes apresentadas no Capítulo 12, aplica-se igualmente ao projeto de interfaces de WebApp. Se você ainda não o fez, leia agora o Capítulo 12.

Document shared on www.docsity.com

AVISO

Se é provável que usuários possam entrar na sua WebApp em vários pontos e níveis da hierarquia de conteúdo, certifique-se de projetar cada página com características de navegação que levem o usuário a outros pontos de interesse.

Dix [DIX99] argumenta que um engenheiro da Web deve projetar uma interface de modo que responda a três questões principais do usuário final:

Onde estou? A interface deve (1) fornecer uma indicação da WebApp à que se teve acesso[4] e (2) informar o usuário da sua localização na hierarquia de conteúdo.

O que posso fazer agora? A interface deve sempre ajudar o usuário a entender suas opções atuais — que funções estão disponíveis, que ligações estão ativas, que conteúdo é relevante.

Onde estive; para onde vou? A interface deve facilitar a navegação. Assim, deve fornecer um "mapa" (implementado de um modo fácil de entender) de onde o usuário esteve e quais caminhos podem ser tomados para se dirigir a outro lugar dentro da WebApp.

Uma interface de WebApp efetiva deve fornecer respostas para cada uma dessas questões à medida que o usuário navega pelo conteúdo e funcionalidade.

19.3.1 Princípios e Diretrizes de Projeto de Interface

Bruce Tognozzi [TOG01] define um conjunto de características fundamentais que todas as interfaces devem exibir e, ao fazê-lo, estabelece uma filosofia que deve ser seguida por todo projetista de interface da WebApp:

PONTO CHAVE

Uma boa interface de WebApp é inteligível e "generosa", dando ao usuário uma sensação de controle.

> Interfaces efetivas são visualmente inteligentes e "generosas", instilando em seus usuários uma sensação de controle. Os usuários rapidamente vêem a amplitude de suas opções, compreendem como atingir suas metas e fazem seu trabalho.
>
> Interfaces efetivas não preocupam o usuário com o trabalho interno do sistema. O trabalho é cuidadoso e continuamente poupado, com plena opção para o usuário desfazer uma atividade a qualquer momento.
>
> Aplicações e serviços efetivos realizam um máximo de trabalho, enquanto exigem um mínimo de informação dos usuários.

A fim de projetar interfaces que exibam essas características, Tognozzi [TOG01] identifica um conjunto de princípios de projeto prevalentes:[5]

Antecipação — Uma WebApp deve ser projetada de modo que antecipe o próximo movimento do usuário. Por exemplo, considere uma WebApp de apoio ao cliente desenvolvida por um fabricante de impressoras de computador. Um usuário solicitou um objeto de conteúdo que apresenta informação sobre um drive de impressora para um sistema operacional recentemente lançado. O projetista da WebApp deve antecipar que o usuário pode solicitar o *download* do driver e deve fornecer facilidades de navegação para que isso aconteça sem exigir que o usuário procure essa capacidade.

PONTO CHAVE

Uma interface de WebApp deve ser projetada para respeitar o conjunto de princípios aqui mencionado.

Comunicação — A interface deve comunicar o estado de qualquer atividade iniciada pelo usuário. A comunicação pode ser óbvia (por exemplo, uma mensagem de texto) ou sutil (por exemplo, uma folha movendo-se pela impressora para indicar que a impressão está em andamento). A interface deve também comunicar o estado do usuário (por exemplo, a identificação do usuário) e localização dentro da hierarquia de conteúdo da WebApp.

Consistência — O uso de controles de navegação, menus, ícones e estéticas (por exemplo, cor, forma, leiaute) *devem ser consistentes ao longo da WebApp.* Por exemplo, se texto sublinhado em azul significa um link de navegação, o conteúdo nunca deve incorporar texto sublinhado de azul que não implique link. Toda característica da interface deve responder para que seja consistente com as expectativas do usuário.[6]

4 Cada um de nós já marcou uma página na Internet, apenas para revisitá-la depois, e não ter nenhuma indicação do site nem do conteúdo da página (ou nenhum jeito de se dirigir para outra localização dentro do site).

5 Os princípios originais de Tognozzi foram adaptados e estendidos para uso neste livro. Veja [TOG01] para maior discussão desses princípios.

6 Tognozzi [TOG01] menciona que o único modo de certificar-se de que as expectativas do usuário são adequadamente entendidas é por meio de teste de usuário abrangente (Capítulo 20).

Autonomia controlada — A interface deve facilitar o movimento do usuário ao longo da WebApp, mas deve fazê-lo de modo que imponha as convenções de navegação estabelecidas para a aplicação. Por exemplo, navegação para partes restritas da WebApp devem ser controladas pelo ID do usuário e senha, e não deve haver mecanismo de navegação que permita ao usuário contornar esses controles.

Flexibilidade — A interface deve ser suficientemente flexível para permitir que alguns usuários realizem diretamente tarefas e outros explorem a WebApp de modo um tanto aleatório. Em qualquer caso, deve permitir ao usuário entender onde ele está e lhe fornecer funcionalidade que possa desfazer erros e refazer caminhos de navegação mal escolhidos.

Enfoque — A interface da WebApp (e o conteúdo que ela apresenta) deve permanecer concentrada nas tarefa(s) do usuário em mão. Em toda hipermídia há uma tendência de conduzir o usuário a um conteúdo aproximadamente relacionado. Por quê? Porque é mais fácil fazê-lo! O problema é que o usuário pode rapidamente se confundir em muitas camadas de informação de apoio e perder de vista o conteúdo original que ele queria inicialmente.

Lei de Fitt — "O tempo para alcançar um alvo é função da distância e do tamanho do alvo" [TOG01]. Com base em um estudo conduzido na década de 1950 [FIT54], a lei de Fitt "é um método efetivo de modelar movimentos rápidos apontados para onde um apêndice (como uma mão) começa em repouso, em uma posição inicial específica, e se move para estacionar em uma área visada" [ZHA02]. Se uma seqüência de seleções ou entradas padronizadas (com muitas opções diferentes dentro da seqüência) é definida por uma tarefa de usuário, a primeira seleção (por exemplo, clique de mouse) deve ser fisicamente próxima da seleção seguinte. Por exemplo, considere a interface da página principal de uma WebApp em um site de e-commerce que vende produtos eletrônicos ao consumidor.

Cada opção do usuário implica um conjunto de escolhas ou ações subseqüentes do usuário. Por exemplo, uma opção "compra de um produto" requer que o usuário introduza uma categoria de produto seguida pelo nome do produto. A categoria de produto (por exemplo, equipamento de áudio, televisores, aparelhos de DVD) aparece em um menu *pull-down* tão logo a "compra de um produto" é escolhida. Assim, a escolha seguinte é imediatamente óbvia (está próxima), e o tempo para atingi-la é desprezível. Se, por outro lado, a escolha aparecesse em um menu que estivesse localizado do outro lado da tela, o tempo para o usuário atingi-lo (e então fazer a escolha) seria bastante longo.

Veja na Web

Uma busca na Web vai descobrir muitas bibliotecas disponíveis, por exemplo, pacotes JavaAPI, interfaces e classes em **Java.sun.com** ou COM, DCOM e Bibliotecas de Tipo em **msdn.Microsoft.com**.

Objetos de interface humana — Uma extensa biblioteca de objetos de interface humana reutilizáveis foi desenvolvida para WebApps. Use-a. Qualquer objeto de interface que pode ser "visto, ouvido, tocado ou percebido de outro modo" [TOG01] por um usuário final pode ser adquirido de uma entre várias bibliotecas de objetos.

Redução de latência — Em vez de fazer o usuário esperar até que alguma operação interna seja completada (por exemplo, baixar [downloading] uma imagem gráfica complexa), a WebApp deve usar multitarefas para permitir que o usuário prossiga o trabalho como se a operação tivesse sido completada. Além disso, para reduzir latência, atrasos devem ser informados de modo que o usuário entenda o que está acontecendo. Isso inclui (1) fornecer realimentação de áudio (por exemplo, um clique ou campainha) quando uma seleção não resulta em uma ação imediata da WebApp; (2) exibir um relógio animado ou uma barra de progresso para indicar que o processamento está em curso; (3) fornecer algum entretenimento (por exemplo, uma apresentação de animação ou texto) enquanto o processamento demorado ocorre.

> "A melhor viagem é aquela com menos passos. Encurte a distância entre o usuário e seu objetivo."
>
> **Autor desconhecido**

Aprendizado — Uma interface de WebApp deve ser projetada para minimizar o tempo de aprendizado e, uma vez aprendida, para minimizar o reaprendizado necessário quando a WebApp é revisitada. Em geral, a interface deve enfatizar um projeto simples, intuitivo, que organiza conteúdo e funcionalidade em categorias que sejam óbvias para o usuário.

Document shared on www.docsity.com

Metáforas — Uma interface que usa uma metáfora de interação é mais fácil de aprender e de usar, desde que a metáfora seja apropriada para a aplicação e para o usuário. Uma metáfora deve invocar imagens e conceitos da experiência do usuário, mas não precisa ser uma reprodução exata de uma experiência do mundo real. Por exemplo, um site de e-comércio que implementa pagamento de contas automático para uma instituição financeira usa uma metáfora de talão de cheques (não surpreendentemente) para ajudar o usuário na especificação e cronogramação de pagamentos de contas. No entanto, quando um usuário "preenche" um cheque, ele não precisa entrar com o nome completo do pagador, mas pode escolher de uma lista de pagadores ou fazer o sistema selecionar com base nas primeiras letras digitadas. A metáfora permanece intacta, mas o usuário obtém ajuda da WebApp.

AVISO

Metáforas são uma excelente idéia porque refletem experiência do mundo real. Certifique-se, no entanto, de que a metáfora escolhida seja bem conhecida entre os usuários finais.

Manter a integridade do produto de trabalho. Um produto de trabalho (por exemplo, um formulário preenchido pelo usuário, uma lista especificada pelo usuário) deve ser automaticamente salvo para que não seja perdido se um erro ocorrer. Cada um de nós já experimentou a frustração associada com a conclusão do preenchimento de um formulário extenso de WebApp e ter o conteúdo perdido devido a erros (feitos por nós, pela WebApp, ou na transmissão do cliente para o servidor). Para evitar isso, uma WebApp deve ser projetada para auto-salvar todos os dados especificados pelo usuário.

Legibilidade — Toda informação apresentada pela interface deve ser legível a jovens e idosos. O projetista de interface deve enfatizar estilos de tipos legíveis, tamanhos de fonte e escolha de cor de fundo que aumentem o contraste.

Rastrear estado — Quando apropriado, o estado da interação do usuário deve ser rastreado e armazenado para que um usuário possa sair e retornar mais tarde para voltar ao ponto em que estava quando saiu. Em geral, *cookies* podem ser projetados para guardar informação de estado. No entanto, *cookies* são uma tecnologia controversa, e outras soluções de projeto podem ser mais aceitáveis para alguns usuários.

Navegação visível — Uma interface bem projetada de WebApp fornece "a ilusão de que usuários estão no mesmo lugar, com um trabalho trazido até eles" [TOG01]. Quando essa abordagem é usada, navegação não é uma preocupação do usuário. Em vez disso, o usuário recupera objetos de conteúdo e seleciona funções exibidas e executadas ao longo da interface.

CASASEGURA

Revisão do Projeto de Interface

A cena: Escritório de Doug Miller.

Os personagens: Doug Miller (gerente do grupo de engenharia de software do *CasaSegura*) e Vinod Raman, um membro da equipe de engenharia de software do produto *CasaSegura*.

A conversa:

Doug: Vinod, você e a equipe tiveram oportunidade de revisar o protótipo da interface de e-commerce do CasaSeguraGarantida.com?

Vinod: Sim... nós todos passamos por ela de um ponto de vista técnico, e temos um monte de anotações. Eu as passei por e-mail para Sharon [gerente da equipe de engenharia da Web para o fornecedor terceirizado do site de e-comércio da Web do *CasaSegura*] ontem.

Doug: Você e Sharon podem se encontrar e discutir as coisas pequenas... me dê um resumo dos tópicos importantes.

Vinod: Em geral eles fizeram um bom trabalho, porém nada espetacular. É uma típica interface de e-commerce, com estética decente e leiaute razoável; eles cuidaram de todas as funções importantes...

Doug (sorrindo tristemente): Mas?

Vinod: Bem, há algumas coisas...

Doug: Tais como?

Vinod (mostrando a Doug uma seqüência de pranchas do protótipo de interface): Aqui está o menu principal de funções exibido na página principal:

Aprenda sobre o *CasaSegura*

Descreva seu prédio

Obtenha recomendações de componente do *CasaSegura*

Compre um sistema *CasaSegura*

Obtenha suporte técnico

O problema não é com essas funções, elas estão todas certas, mas o nível de abstração não está correto.

Doug: Elas são todas funções principais, não são?

Vinod: São, mas aí que está a coisa... você pode comprar um sistema entrando com uma lista de componentes, não há necessidade real de descrever o prédio, se você não quiser. Eu sugeriria apenas quatro opções de menu na página principal.

Aprenda sobre o *CasaSegura*

Especifique o sistema *CasaSegura* que você quer

Compre um sistema *CasaSegura*

Obtenha suporte técnico

Ao selecionar **Especifique o sistema *CasaSegura* que você quer**, você terá então as seguintes opções:

Selecione componentes do *CasaSegura*

Obtenha recomendações de componentes do *CasaSegura*

Se você for um usuário competente, selecionará componentes de um conjunto de menus *pull-down* categorizado para sensores, câmeras, painéis de controle etc. Se precisar de ajuda, você solicitará uma recomendação e isso exigirá a descrição do prédio. Acho que isso é um pouco mais lógico.

Doug: Eu concordo. Você falou com Sharon sobre isso?

Vinod: Não, quero discutir primeiro com o marketing e depois falarei com ela.

Nielsen e Wagner [NIE96] sugerem algumas diretrizes pragmáticas de projeto de interface (baseadas em experiência dos autores no reprojeto de uma importante WebApp) que fornecem um bom complemento para os princípios sugeridos anteriormente nesta seção:

- A velocidade de leitura em um monitor de computador é aproximadamente 25% menor do que a velocidade de leitura em papel. Assim, não force o usuário a ler quantidades volumosas de texto, particularmente quando o texto explica a operação da WebApp ou apóia a navegação.
- Evite sinais "em construção" — eles levantam expectativas e causam um link desnecessário que vai certamente desapontar.
- Os usuários preferem não usar barra de rolagem. Informação importante deve ser colocada dentro das dimensões de uma típica janela de navegador.
- Menus de navegação e barras de topo devem ser projetados consistentemente e estar disponíveis em todas as páginas que estão à disposição do usuário. O projeto não deve se apoiar nas funções do navegador para ajudar a navegação.
- A estética nunca deve superar a funcionalidade. Por exemplo, um simples botão poderia ser uma melhor opção de navegação do que uma imagem esteticamente agradável mas vaga, ou um ícone cuja intenção não está clara.
- Opções de navegação devem ser óbvias, mesmo para o usuário eventual. O usuário não deveria ter que procurar na tela para determinar como se ligar a outro conteúdo ou serviço.

Uma interface bem projetada melhora a percepção do usuário, do conteúdo ou dos serviços fornecidos pelo site. Não precisa ser espetacular, mas deve ser sempre bem estruturada e ergonomicamente eficiente.

> "As pessoas têm pouquíssima paciência com sites WWW mal projetados."
>
> Jakob Nielsen e Annette Wagner

19.3.2 Mecanismos de Controle de Interface

Os objetivos de uma interface de WebApp são: (1) estabelecer uma janela consistente com o conteúdo e a funcionalidade fornecidos pela interface; (2) guiar o usuário ao longo de uma série de interações com a WebApp; e (3) organizar as opções de navegação e o conteúdo disponíveis ao usuário. Para obter uma interface consistente, o projetista deve primeiro usar projeto estético (Seção 19.4) para estabelecer uma "aparência" coerente para a interface. Isso engloba muitas características, mas deve enfatizar o leiaute e a forma dos mecanismos de navegação. Para orientar quanto

Document shared on www.docsity.com

à interação com o usuário, o projetista de interface pode se apoiar em uma metáfora apropriada,[7] que permite ao usuário obter um entendimento intuitivo da interface. Para implementar opções de navegação, o projetista seleciona com base em um certo número de mecanismos de interação:

? Que mecanismos de interação estão disponíveis para projetistas de WebApp?

- *Menus de navegação* — menus de palavras-chave (organizados vertical ou horizontalmente) que listam o conteúdo-chave e/ou a funcionalidade. Esses menus podem ser implementados de modo que o usuário possa escolher de uma hierarquia de subtópicos exibida quando a opção de menu principal é selecionada.
- *Ícones gráficos* — botões, chaves e imagens gráficas similares que permitem ao usuário selecionar alguma propriedade ou especificar uma decisão.
- *Imagens gráficas* — alguma representação gráfica que é selecionável pelo usuário e implementa uma ligação para um objeto de conteúdo ou funcionalidade da WebApp.

É importante observar que um ou mais desses mecanismos de controle deve ser fornecido em cada nível da hierarquia de conteúdo.

19.3.3 Projeto de Fluxo de Trabalho de Interface

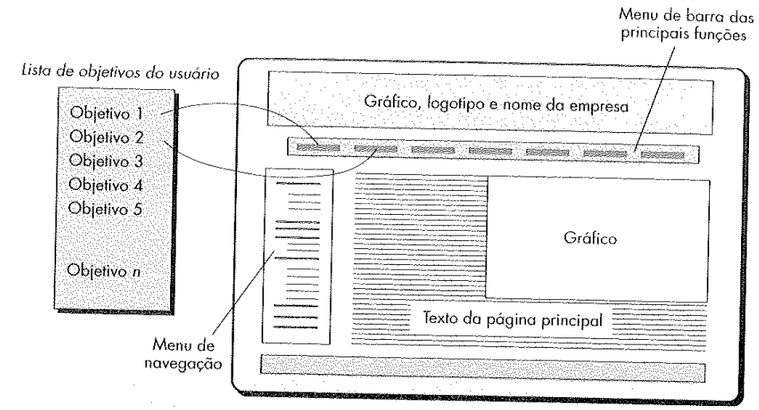
Embora uma discussão profunda de projeto de interface de WebApps seja mais adequada para livros-texto dedicados ao assunto (por exemplo, [GAL02], [RAS00] ou [NIE00]), um breve panorama das tarefas-chave de projeto vale a pena. No Capítulo 12, mencionamos que o projeto de interface com o usuário começa com a identificação dos usuários, da tarefa e dos requisitos ambientais. Uma vez identificadas as tarefas do usuário, cenários de usuário (casos de uso) são criados e analisados para definir um conjunto de objetos e ações de interface. Esse trabalho é representado como parte do modelo de análise da WebApp discutido no Capítulo 18. As seguintes tarefas representam um fluxo de trabalho rudimentar de projeto de interface de WebApp:

1. **Revisar a informação contida no modelo de análise e refiná-lo quando necessário.**
2. **Desenvolver um esboço do leiaute da interface da WebApp.** Um protótipo de interface (incluindo o leiaute) pode ter sido desenvolvido como parte da atividade de modelagem de análise; se deve ser revisado e refinado se necessário; se não foi desenvolvido, a equipe de engenharia da Web deve trabalhar com os interessados para desenvolvê-lo agora. Um primeiro esboço de leiaute esquemático é mostrado na Figura 19.3.
3. **Mapear os objetivos do usuário em ações específicas da interface.** Para a grande maioria de WebApps, o usuário terá um conjunto relativamente pequeno (normalmente entre quatro e sete) de objetivos principais. Estes devem ser mapeados em ações específicas de interface como mostra a Figura 19.3.
4. **Definir um conjunto de tarefas de usuário que estão associadas a cada ação.** Cada ação de interface (por exemplo, "comprar um produto") é associada a um conjunto de tarefas do usuário. Essas tarefas foram identificadas durante a modelagem de análise. Durante o projeto, elas devem ser mapeadas em interações específicas que englobam tópicos de navegação, objetos de conteúdo e funções da WebApp.
5. **Pranchas com imagens de tela para cada ação de interface.** À medida que cada ação é considerada, uma seqüência de pranchas com imagens (imagens de tela) deve ser criada para mostrar como a interface responde à interação do usuário. Objetos de conteúdo precisam ser identificados (mesmo se eles não tiverem ainda sido projetados e desenvolvidos), a funcionalidade da WebApp tem de ser mostrada e links de navegação devem ser indicados.
6. **Refinar o leiaute de interface e pranchas usando entradas do projeto de estética.** O esboço de leiaute e as pranchas são completadas pelos engenheiros da Web, mas o aspecto estético de um site comercial importante é freqüentemente desenvolvido por profissionais da arte, em vez de técnicos.

7 Nesse contexto, a *metáfora* é uma representação (retirada da experiência do usuário no mundo real) que pode ser modelada no contexto da interface. Um exemplo simples poderia ser uma chave deslizante que é usada para controlar o volume de audição de um arquivo .mpg.

FIGURA 19.3

Mapeamento dos objetivos do usuário nas ações de interface



7. **Identificar os objetos de interface com o usuário que são necessários para implementar a interface.** Essa tarefa pode requerer busca em uma biblioteca de objetos existente para encontrar os objetos reusáveis (classes) adequados para a interface da WebApp. Além disso, quaisquer classes de cliente são especificadas nesse momento.
8. **Desenvolver uma representação procedimental da interação do usuário com a interface.** Essa tarefa opcional usa diagramas de seqüência e/ou diagramas de atividade UML (discutidos no Capítulo 18) para tratar do fluxo de atividades (e decisões) que ocorrem à medida que o usuário interage com a WebApp.
9. **Desenvolver uma representação comportamental da interface.** Essa tarefa opcional usa diagramas de estado UML (discutidos no Capítulo 18) para representar transições de estado e os eventos que as causam. Mecanismos de controle (isto é, os objetos e ações disponíveis ao usuário para alterar um estado da WebApp) são definidos.
10. **Descrever o leiaute da interface de cada estado.** Usando informação de projeto desenvolvida nas Tarefas 2 e 5, associar um leiaute específico ou imagem de tela a cada estado da WebApp descrito na Tarefa 9.
11. **Refinar e revisar o modelo de projeto de interface.** A revisão da interface deve enfocar a usabilidade (Capítulo 12).

É importante observar que o conjunto final escolhido pela equipe de engenharia da Web deve ser adaptado para os requisitos especiais da aplicação que está para ser construída.

19.4 Projeto Estético

AVISO

Nem todo engenheiro da Web (ou engenheiro de software) tem talento artístico (estético). Se você está nessa categoria, contrate um projetista gráfico experiente para o projeto estético funcionar.

Projeto estético, também chamado de *projeto gráfico*, é um esforço artístico que complementa os aspectos técnicos da engenharia da Web. Sem ele, uma WebApp pode ser funcional, mas não atraente. Com ele, uma WebApp leva os seus usuários a um mundo que os abraça em um nível físico, bem como intelectual.

Mas o que é estética? Há um velho ditado: "beleza existe no olho do observador". Isso é particularmente apropriado quando o projeto estético de WebApps é considerado. Para realizar um projeto estético efetivo, vamos retornar à hierarquia do usuário desenvolvida como parte do modelo de análise (Capítulo 18) e perguntar: quem são os usuários da WebApp e que "aparência" eles desejam?

Document shared on www.docsity.com

> "Descobrimos que as pessoas avaliam rapidamente um site apenas pelo seu projeto visual."
>
> Diretrizes de Stanford para Credibilidade na Web

19.4.1 Tópicos de Leiaute

Toda página Web tem uma quantidade limitada de "terreno" que pode ser usado para apoiar estética não funcional, características de navegação, conteúdo de informação e funcionalidade dirigida ao usuário. O "desenvolvimento" desse terreno é planejado durante o projeto estético.

Como todos os tópicos de estética, não há regras absolutas quando o leiaute de tela é projetado. No entanto, um certo número de diretrizes gerais de leiaute é digno de consideração:

Não tenha medo de espaço em branco. Não é aconselhável abarrotar cada centímetro de uma página Web com informação. A aglomeração resultante dificulta ao usuário identificar a informação ou as características desejadas e cria um caos visual que não é agradável à vista.

Enfatize o conteúdo. Afinal de contas, essa é a razão pela qual o usuário está lá. Nielsen [NIE00] sugere que a página Web típica deve ter 80% de conteúdo com o espaço restante dedicado à navegação e outras características.

Organize os elementos de leiaute do canto superior esquerdo ao canto inferior direito. A grande maioria dos usuários vai percorrer uma página Web de modo semelhante àquele pelo qual ele percorre a página de um livro — canto superior esquerdo ao canto inferior direito.[8] Se os elementos do leiaute têm prioridades específicas, os elementos de alta prioridade devem ser colocados na porção superior esquerda do espaço da página.

Agrupe navegação, conteúdo e função geograficamente dentro da página. As pessoas procuram padrões virtualmente em tudo. Se não há padrões discerníveis em uma página Web, a frustração do usuário provavelmente vai aumentar (por conta das buscas desnecessárias para informação desejada).

Não estenda seu espaço com a barra de rolagem (scrolling bar). Embora a rolagem seja freqüentemente necessária, a maioria dos estudos indica que usuários prefeririam a não-rolagem. É melhor reduzir o conteúdo da página ou apresentar o conteúdo necessário em várias páginas.

Considere a resolução e o tamanho da janela do navegador quando projetar o leiaute. Em vez de definir tamanhos fixos em um leiaute, o projetista deveria especificar todos os itens de leiaute como uma porcentagem de espaço disponível [NIE00].

19.4.2 Tópicos de Projeto Gráfico

Projeto gráfico considera todos os pontos do aspecto de uma WebApp. O processo de projeto gráfico inicia com o leiaute (Seção 19.4.1) e abrange esquemas de cores, tamanho e estilo de caracteres, tamanhos e estilos gerais, uso de mídia suplementar (por exemplo, áudio, vídeo, animação), e todos os outros elementos estéticos de uma aplicação. O leitor interessado pode obter informações e diretrizes em muitos sites Web dedicados ao assunto (por exemplo, *www.graphic-design.com*, *www.grantasticdesigns.com*, *www.wpdfd.com*) ou de um ou mais recursos impressos (por exemplo, [BAG01], [CLO01], or [HEI02]).

INFO

Sites Web Bem Projetados

Algumas vezes, o melhor modo de entender um bom projeto de WebApp é procurar alguns exemplos. Em seu artigo "The Top Twenty Web Design Tips" [As vinte melhores dicas de projeto na Web], Marcelle Toor (*http://www.graphic-design.com/Web/feature/tips.html*) sugere os seguintes sites como exemplos de bom projeto gráfico:

www.primo.com — empresa de projeto liderada por Primo Angeli.

www.workbook.com — esse site mostra o trabalho de vitrine por ilustradores e projetistas.

www.pbs.org/riverofsong — série da televisão para TV e rádio públicos sobre música americana.

www.RKDINC.com — empresa de projeto com uma seleção de bons trabalhos colocada on-line e boas dicas de projeto.

www.commarts.com/career/index.html — a revista *Communication Arts*, um periódico específico para projetistas gráficos. Uma boa fonte para outros sites bem projetados.

www.btdnyc.com — empresa de projeto liderada por Beth Toudreau.

8 Há exceções baseadas na cultura e na língua, mas essa regra vale para a maioria dos usuários.

19.5 Projeto de Conteúdo

Projeto de conteúdo aborda dois tópicos diferentes de projeto, cada um tratado por indivíduos com conjuntos diferentes de habilidade. Projeto de conteúdo desenvolve uma representação de projeto para objetos de conteúdo e representa os mecanismos necessários para instanciar seus relacionamentos mútuos. Essa atividade de projeto é conduzida por engenheiros da Web.

Além disso, o projeto de conteúdo preocupa-se com a representação da informação em um objeto de conteúdo específico — uma atividade de projeto conduzida por redatores, projetistas gráficos e outros que geram o conteúdo a ser usado em uma WebApp.

> "Bons projetistas podem criar normalidade a partir do caos; eles podem comunicar idéias claramente por meio da organização e manipulação de palavras e figuras."
>
> Jeffery Veen

19.5.1 Objetos de Conteúdo

O relacionamento entre objetos de conteúdo definido como parte do modelo de análise da WebApp (por exemplo, Figura 18.3) e os objetos de projeto que representam conteúdo é análogo ao relacionamento entre as classes de análise e os componentes de projeto descritos no Capítulo 11. No contexto de engenharia da Web, um *objeto de conteúdo* está mais próximo de um objeto de dados de software convencional. Um objeto de conteúdo tem atributos que incluem informação específica de conteúdo (normalmente definida durante a modelagem de análise da WebApp) e atributos específicos de implementação, que são especificados como parte do projeto.

Como exemplo, considere a classe de análise desenvolvida para o sistema de e-commerce do *CasaSegura* denominada **ComponenteDoProduto** desenvolvida no Capítulo 18 e mostrada na Figura 19.4. No Capítulo 18, mencionamos um atributo **descrição** representado aqui como uma classe de projeto denominada **DescriçãoDeComponente** composta de cinco objetos de conteúdo: **DescriçãoDeMarketing**, **Fotografia**, **DescriçãoTécnica**, **Esquema** e **Vídeo**, mostrados como objetos sombreados na figura. A informação contida nos objetos de conteúdo é indicada como atributos. Por exemplo, **Fotografia** (uma imagem .jpg) tem os atributos **dimensão horizontal**, **dimensão vertical** e **estilo de borda**.

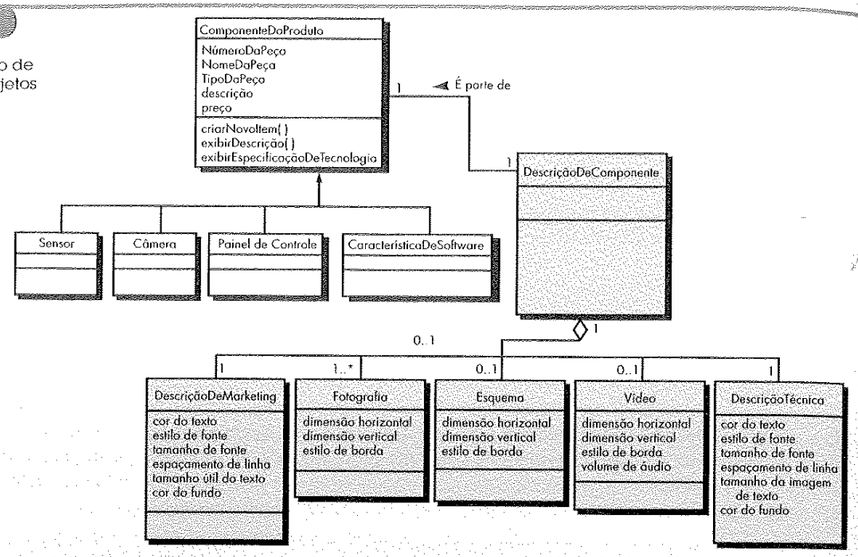
Associação e agregação[9] UML podem ser usadas para representar relacionamentos entre objetos de conteúdo. Por exemplo, a associação UML mostrada na Figura 19.4 indica que uma **DescriçãoDeComponente** é usada para cada instância da classe **ComponenteDoProduto**. **DescriçãoDeComponente** é composta de cinco objetos de conteúdo mostrados. No entanto, a notação de multiplicidade apresentada indica que **Esquema** e **Vídeo** são opcionais (0 ocorrência é possível), uma **DescriçãoDeMarketing** e **DescriçãoTécnica** é necessária, e uma ou mais instâncias de **Fotografia** é usada.

9 Ambas as representações são discutidas no Capítulo 8.

Document shared on www.docsity.com

FIGURA 19.4

Representação de projeto dos objetos de conteúdo



19.5.2 Tópicos de Projeto de Conteúdo

AVISO

Usuários tendem a tolerar mais prontamente rolagem vertical do que rolagem horizontal. Evite formatos de página muito largos.

Uma vez modelados todos os objetos de conteúdo, a informação que cada um deles deve entregar precisa ser criada e depois formatada para melhor atender às necessidades do cliente. A autoria do conteúdo é de especialistas que projetam o objeto de conteúdo fornecendo um esboço da informação a ser entregue e uma indicação de tipos de objetos de conteúdo genéricos (por exemplo, texto descritivo, imagens gráficas, fotografias) a ser usados para entregar a informação. O projeto estético (Seção 19.4) também pode ser aplicado para representar o aspecto adequado do conteúdo.

À medida que objetos de conteúdo são projetados, eles são "juntados" [POW00] para formar as páginas da WebApp. O número de objetos de conteúdo incorporados em uma única página é função das necessidades do usuário, das restrições impostas pela velocidade de *download* das conexões Internet e das restrições impostas pela quantidade de rolagem que o usuário vai tolerar.

19.6 Projeto de Arquitetura

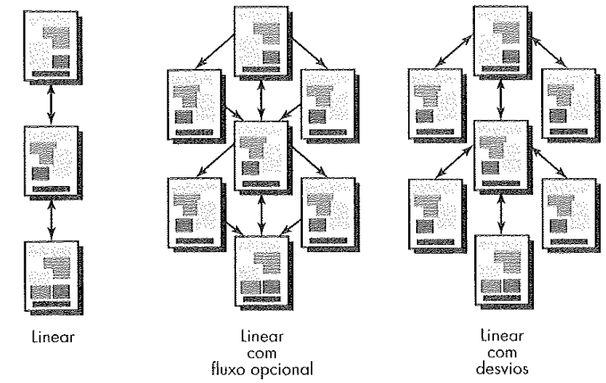
Projeto de arquitetura está ligado aos objetivos estabelecidos para uma WebApp, o conteúdo a ser apresentado, os usuários visitantes, e a filosofia de navegação estabelecida. O projetista arquitetural deve identificar a arquitetura de conteúdo e a arquitetura da WebApp. *Arquitetura de conteúdo*[10] concentra-se na maneira pela qual objetos de conteúdo (ou composição de objetos tais como as páginas da Web) são estruturados para apresentação e navegação. *Arquitetura da WebApp* trata do modo pelo qual a aplicação é estruturada para gerir a interação com o usuário, manipular tarefas de processamento interno, efetuar navegação e apresentar conteúdo.

> "[A] estrutura arquitetural de um site bem projetado nem sempre é aparente ao usuário — nem deveria ser."
>
> Thomas Powell

10 O termo *arquitetura de informação* também é usado para conotar estruturas que levam à melhor organização, rotulagem, navegação e busca de objetos de conteúdo.

FIGURA 19.5

Estruturas lineares



Na maioria dos casos, o projeto de arquitetura é conduzido em paralelo com o projeto de interface, estético e de conteúdo. Como a arquitetura da WebApp pode ter uma forte influência na navegação, as decisões feitas durante essa atividade de projeto vão influenciar o trabalho conduzido durante o projeto de navegação.

19.6.1 Arquitetura de Conteúdo

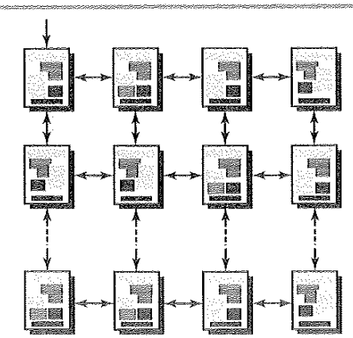
O projeto de *conteúdo da arquitetura* focaliza a definição da estrutura global de hipermídia da WebApp. O projeto pode escolher uma das quatro diferentes estruturas [POW00]:

Estruturas lineares (Figura 19.5) são encontradas quando uma seqüência de interações previsível (com alguma variação ou diversificação) for comum. Um exemplo clássico poderia ser um tutorial de apresentação em que páginas de informação, junto com gráficos relacionados, vídeos curtos ou áudios são exibidos apenas depois de ter sido fornecida a informação de pré-requisito. A seqüência de apresentação de conteúdo é geralmente linear e predefinida. Outro exemplo poderia ser uma seqüência de entrada de encomenda de produto, na qual a informação específica deve ser fornecida numa ordem pré-especificada. Em tais casos, as estruturas mostradas na Figura 19.5 são adequadas. À medida que o conteúdo e o processamento tornam-se mais complexos, o fluxo puramente linear, mostrado à esquerda da figura, dá lugar a estruturas lineares mais sofisticadas, nas quais o conteúdo alternativo pode ser requisitado ou ocorrer um desvio para adquirir um conteúdo complementar (estrutura mostrada à direita na Figura 19.5).

Estruturas em malha (Figura 19.6) são uma opção arquitetural a ser aplicada quando o conteúdo da WebApp pode ser organizado em categorias de duas (ou mais) dimensões. Por exemplo, considere uma situação na qual um site de comércio eletrônico vende tacos de golfe. A dimensão horizontal

FIGURA 19.6

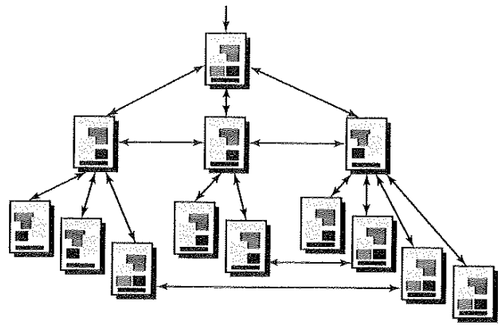
Estrutura em malha



Document shared on www.docsity.com

FIGURA 19.7

Estrutura hierárquica



da malha representa o tipo de taco a ser vendido (por exemplo, de madeira, de ferro, embocador, suporte "T"). A dimensão vertical representa as ofertas fornecidas por vários fabricantes de tacos de golfe. Assim, um usuário poderia navegar na malha horizontalmente para encontrar a coluna de suportes e, depois, verticalmente para examinar as ofertas dos fabricantes de suportes. Essa arquitetura de WebApp é usada apenas quando é encontrado um conteúdo altamente regular [POW00].

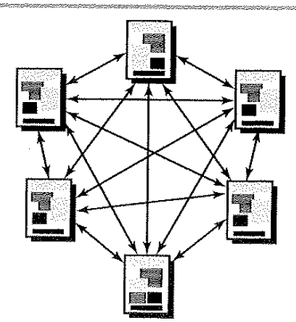
Estruturas hierárquicas (Figura 19.7) são sem dúvida a arquitetura mais comum de WebApp. Diferentemente das hierarquias de software particionadas, discutidas no Capítulo 10, que incentivam o fluxo de controle apenas ao longo dos ramos verticais da hierarquia, uma estrutura hierárquica de WebApp pode ser projetada de maneira que permita um fluxo de controle horizontalmente (por meio de ramificação de hipertexto), ao longo dos ramos verticais da estrutura. Assim, o conteúdo apresentado no ramo mais à esquerda da hierarquia pode ter ligações de hipertexto que levem ao conteúdo existente no meio, ou no ramo mais à direita da estrutura. Deve-se notar, no entanto, que tal ramificação permite uma navegação rápida ao longo do conteúdo da WebApp e pode causar confusão para o usuário.

Uma *estrutura em rede* ou *"pura teia"* (Figura 19.8) é semelhante em muitos pontos à arquitetura que evolui de sistemas orientados a objetos. Os componentes arquiteturais (neste caso, páginas da Web) são projetados para que possam passar o controle (através de vínculos de hipertexto) virtualmente para qualquer outro componente do sistema. Essa abordagem permite uma flexibilidade de navegação considerável, mas, ao mesmo tempo, pode ser confusa para o usuário.

As estruturas arquiteturais discutidas nos parágrafos anteriores podem ser combinadas para formar *estruturas compostas*. A arquitetura global de uma WebApp pode ser hierárquica, mas uma parte da estrutura pode exibir características lineares, enquanto a outra parte pode ser em rede. A meta do projetista de arquitetura é adaptar a estrutura da WebApp ao conteúdo a ser apresentado e ao processamento a ser conduzido.

FIGURA 19.8

Estrutura em rede



19.6.2 Arquitetura da WebApp

Arquitetura da WebApp descreve uma infra-estrutura que possibilita a um sistema ou aplicação baseado na Web atingir seus objetivos de negócio. Jacyntho e seus colegas [JAC02] descrevem as características básicas dessa infra-estrutura da seguinte maneira:

> Aplicações devem ser construídas usando camadas nas quais diferentes preocupações são levadas em conta; em particular, dados da aplicação devem ser separados dos conteúdos de página (nós de navegação) e esses conteúdos, por sua vez, devem ser claramente separados dos aspectos da interface (páginas).

PONTO CHAVE

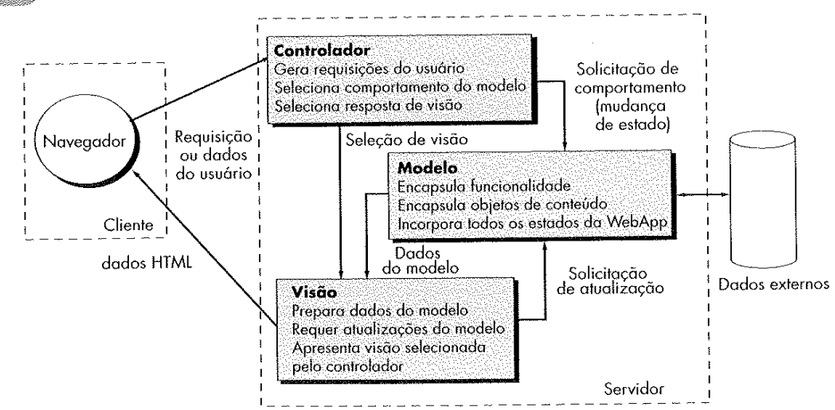
A arquitetura MVC desacopla a interface com o usuário da funcionalidade da WebApp e do conteúdo da informação.

Os autores sugerem um projeto de arquitetura em três camadas que desacopla a interface da navegação e do comportamento da aplicação, e argumentam que manter a interface, aplicação e navegação separadas simplifica a implementação e aumenta o reuso.

A arquitetura *Modelo-Visão-Controle* (*Model-View-Controller*, MVC) [KRA88][11] é uma de um certo número de modelos de infra-estruturas de WebApp sugeridos que desacoplam a interface com o usuário da funcionalidade e conteúdo informacional da WebApp. O *modelo* (algumas vezes referido como "objeto modelo") contém todo o conteúdo e lógica de processamento específicos da aplicação, incluindo todos os objetos de conteúdo, acesso a dados/fontes de informação externos, e toda a funcionalidade de processamento específica da aplicação. A *visão* contém todas as funções específicas da interface e permite a apresentação do conteúdo e lógica de processamento, inclusive todos os objetos de conteúdo, acesso a dados/fontes de informação externos, e toda a funcionalidade de processamento requerida pelo usuário final. O *controlador* gera o acesso ao *modelo* e à *visão* e coordena o fluxo de dados entre eles. Em uma WebApp, a "visão é atualizada pelo controlador com dados do modelo baseado na entrada do usuário" [WMT02]. Uma representação esquemática da arquitetura MVC é mostrada na Figura 19.9.

Com referência à figura, requisições de usuário ou dados são manipulados pelo controlador, que também seleciona o objeto de visão aplicável com base na requisição do usuário. Uma vez determinado o tipo de requisição, uma requisição de comportamento é transmitida para o modelo, que implementa a funcionalidade ou recupera o conteúdo requerido para acomodá-la. O objeto modelo pode ter acesso a dados armazenados em um banco de dados da empresa, como parte de um depósito de dados local ou como uma coleção de arquivos independentes. Os dados desenvolvidos pelo modelo devem ser formatados e organizados pelo objeto de visão adequado e depois transmitidos do servidor da aplicação de volta para o navegador com base no cliente para exibição na máquina do usuário.

FIGURA 19.9 A arquitetura MVC (adaptada de [JAC02])



11 Deve-se observar que MVC é na verdade um padrão de projeto arquitetural desenvolvido para o ambiente Smalltalk (visite *http://www.setus-links.org/oo_smalltalk.html*) e pode ser usado para qualquer aplicação interativa.

Document shared on www.docsity.com

Em muitos casos, a arquitetura da WebApp é definida dentro do contexto do ambiente de desenvolvimento no qual a aplicação deve ser implementada (por exemplo, ASPNET, JWAA ou J2EE). O leitor interessado deve consultar [FOW03] para uma discussão mais profunda sobre ambientes de desenvolvimento modernos e seu papel no projeto das arquiteturas de aplicações na Web.

19.7 Projeto de Navegação

Uma vez estabelecida a arquitetura da WebApp e identificados os seus componentes (páginas, *scripts*, *applets* e outras funções de processamento), o projetista deve definir caminhos de navegação que permitam ao usuário ter acesso ao conteúdo e aos serviços da WebApp. Para conseguir isso, ele precisa (1) identificar a semântica de navegação para diferentes usuários do site e (2) definir a mecânica (sintaxe) para realizar a navegação.

> "Espere, Gretel, até que a lua nasça, e depois nós veremos as migalhas de pão que eu espalhei por aí, elas vão nos mostrar o caminho de volta para casa."
>
> **de Hansel e Gretel**

19.7.1 Semântica Navegacional

Como muitas atividades de engenharia da Web, o projeto de navegação começa com a consideração da hierarquia do usuário e os casos de uso relacionados (Capítulo 18) desenvolvidos para cada categoria de usuário (ator). Cada ator pode usar a WebApp de forma um tanto diferente e assim ter diferentes necessidades de navegação. Além disso, os casos de uso desenvolvidos para cada ator vão definir um conjunto de classes que englobam um ou mais objetos de conteúdo ou funções da WebApp. À medida que cada usuário interage com a WebApp, encontra uma série de *unidades de semânticas de navegação* (*navigation semantic units* — NSUs), ou seja, "um conjunto de informação e estruturas de navegação relacionadas que colaboram para o atendimento de um subconjunto de requisitos de usuário relacionados" [CAC02].

Gnaho e Larcher [GNA99] descrevem a NSU do seguinte modo:

> A estrutura de uma NSU é composta de um conjunto de subestruturas navegacionais que chamamos de *modos de navegação* (*ways of navigating* — WoN). Um WoN representa o melhor modo ou caminho de navegação para usuários com certo perfil atingirem sua meta ou submeta desejada. Assim, o conceito de WoN está associado ao conceito de Perfil de Usuário.
>
> A estrutura de um WoN é feita com base em um conjunto de *nós navegacionais* (*navigational nodes* — NN) relevantes conectados por *vínculos navegacionais*, incluindo algumas vezes outras NSUs. O que significa que NSUs podem, por sua vez, ser agregadas para formar uma NSU de nível mais alto, ou podem ser aninhados em qualquer profundidade.

Para ilustrar o desenvolvimento de uma NSU, considere o caso de uso *selecionar componentes do CasaSegura*, descrito na Seção 18.1.2 e reproduzido aqui:

> **Caso de uso:** *selecionar componentes do CasaSegura*
>
> A WebApp vai recomendar componentes de produto (por exemplo, painéis de controle, sensores, câmeras) e outras características (por exemplo, funcionalidade baseada em PC implementada em software) para cada cômodo e entrada externa. Se solicito alternativas, a WebApp vai fornecê-las, se existirem. Poderei obter informação descritiva e de preço de cada componente de produto. A WebApp vai criar e mostrar uma lista de materiais à medida que seleciono os vários componentes. Poderei dar à lista de materiais um nome e salvá-la para referência futura (veja o caso de uso: *salvar configuração*).

Os itens sublinhados na descrição do caso de uso representam classes e objetos de conteúdo que serão incorporados em uma ou mais NSUs que vão permitir a um novo cliente realizar o cenário descrito no caso de uso *selecionar componentes do CasaSegura*.

PONTO CHAVE

Uma NSU descreve os requisitos de navegação de cada caso de uso. Na verdade, a NSU mostra como um ator se movimenta entre objetos de conteúdo ou funções da WebApp.

A Figura 19.10 mostra uma análise semântica parcial da navegação implícita no caso de uso *selecionar componentes do CasaSegura*. Usando a terminologia anteriormente introduzida, a figura também representa um modo de navegação (WoN) para a WebApp *CasaSeguraGarantida.com*. Importantes classes do domínio do problema são apresentadas com objetos de conteúdo selecionados (neste caso, o pacote de objetos de conteúdo é denominado **DescriçãoDeComp**, que é atributo da classe **ComponenteDeProduto**). Esses itens são nós de navegação. Cada uma das setas representa um vínculo de navegação[12] e é rotulado com a ação iniciada pelo usuário que provoca a ocorrência do vínculo.

O projetista da WebApp cria uma NSU para cada caso de uso associado a cada papel de usuário [GNA99]. Por exemplo, um **novo cliente** (Figura 18.1) pode ter três diferentes casos de uso, todos resultando em acesso a informação e funções da WebApp diferentes. Uma NSU é criada para cada objetivo.

Durante os estágios iniciais do projeto de navegação, a arquitetura de conteúdo da WebApp é avaliada para determinar um ou mais WoNs para cada caso de uso. Como mencionado, um WoN identifica nós de navegação (por exemplo, conteúdo) e os vínculos que permitem a navegação entre eles. Os WoNs são então organizados em NSUs.

> "O problema de navegação em um site da Web é conceitual, técnico, espacial, filosófico e logístico. Conseqüentemente, as soluções tendem a exigir improvisação complexa de combinações de arte, ciência e psicologia organizacional."
>
> **Tim Horgan**

19.7.2 Sintaxe Navegacional

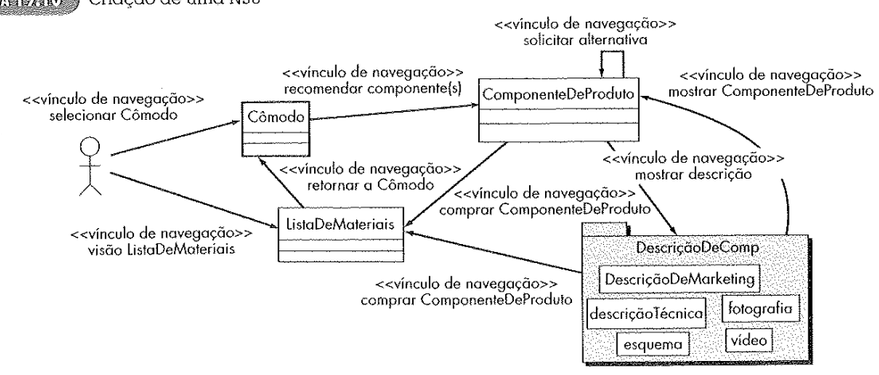
AVISO

Na maioria das situações, escolha mecanismos de navegação horizontal ou vertical, mas não ambos.

À medida que o projeto prossegue, os mecanismos de navegação são definidos. Entre as muitas opções possíveis estão:

- *Link individual de navegação* — vínculos baseados em texto, botões e chaves, e metáforas gráficas.
- *Barra de navegação horizontal* — relaciona as principais categorias de conteúdo ou funcionais em uma barra que contém os links adequados. Em geral, são apresentadas entre quatro e sete categorias.
- *Coluna vertical de navegação* — (1) disponibiliza as principais categorias de conteúdo ou funcionais, ou (2) exibe virtualmente todos os principais objetos de conteúdo da WebApp.

FIGURA 19.10 Criação de uma NSU



12 Esses links são algumas vezes referidos como *vínculos semânticos de navegação (navigation semantics links-NSL)* [CAC02].

Document shared on www.docsity.com

Se a segunda opção é escolhida, tais colunas de navegação podem ser "expandidas" para apresentar objetos de conteúdo como parte de uma hierarquia.

- *Tabs* — uma metáfora que nada mais é do que uma variante da barra ou coluna de navegação, representando categoria de conteúdo ou funcionais como separadores de tab selecionados quando uma ligação é solicitada.
- *Mapas de site* — fornecem um sumário inclusivo para navegação de todos os objetos de conteúdo e de funcionalidade contida na WebApp.

AVISO

O mapa do site deve ser acessível a cada página. O mapa em si deve ser organizado de modo que a estrutura da informação da WebApp seja prontamente aparente.

Além disso, para escolher os mecanismos de navegação, o projetista deve também estabelecer convenções e ajudas de navegação adequadas. Por exemplo, ícones e ligações gráficas devem parecer "clicáveis" por elevação das arestas para dar à imagem uma aparência tridimensional. Realimentação de áudio ou visual deve ser projetada para fornecer ao usuário indicação de que uma opção de navegação foi escolhida. Para navegação baseada em texto, a cor deve ser usada para indicar as ligações de navegação e fornecer indicação das ligações já percorridas. Essas são apenas algumas de dezenas de convenções de projeto que tornam a navegação amigável ao usuário.

19.8 Projeto no Nível de Componente

Aplicações modernas da Web entregam incrementalmente funções de processamento sofisticadas que (1) realizam processamento localizado para gerar capacidade de conteúdo e de navegação de modo dinâmico; (2) fornecem capacidade de cálculo ou de processamento de dados apropriadas para o domínio de negócio da WebApp; (3) fornecem acesso e consulta a banco de dados sofisticado; (4) estabelecem interfaces de dados a sistemas externos à empresa. Para conseguir essas (e muitas outras) capacidades, o engenheiro da Web deve projetar e construir componentes de programa idênticos em forma aos componentes de software convencional.

No Capítulo 11, consideramos projeto no nível de componente em algum detalhe. Os métodos de projeto discutidos no Capítulo 11 aplicam-se a componentes da WebApp, com pouca ou nenhuma modificação. O ambiente de implementação, linguagens de programação e padrões reutilizáveis, arcabouços e software podem variar um pouco, mas a abordagem de projeto global permanece a mesma.

19.9 Padrões de Projeto de Hipermídia

Padrões de projeto usados na engenharia da Web englobam duas classes principais: (1) padrões de projeto genéricos aplicáveis a todos os tipos de software (por exemplo, [BUS96] e [GAM95]) e (2) *padrões de projeto de hipermídia* específicos de WebApps. Padrões de projeto genéricos foram discutidos no Capítulo 9. Vários catálogos e repositórios de padrões de hipermídia podem ser acessados via Internet.[13]

> "Cada padrão é uma regra de três partes que expressa um relacionamento entre um certo contexto, um problema e uma solução."
>
> Christopher Alexander

Como mencionamos anteriormente, padrões de projeto são uma abordagem genérica para resolver algum pequeno problema de projeto que pode ser adaptado a uma ampla variedade de problemas específicos. No contexto de sistemas baseados na Web, German e Cowan [GER00] sugerem as seguintes categorias de padrões:

Padrões arquiteturais. Apóiam o projeto de conteúdo e de arquitetura da WebApp. As Seções 19.6.1 e 19.6.2 apresentam padrões arquiteturais para conteúdo e arquitetura da WebApp. Além disso, muitos padrões arquiteturais relacionados estão disponíveis (por exemplo, *Java Blueprints* em java.sun.com/blueprints/) para engenheiros da Web que precisam projetar WebApps em uma variedade de domínios de negócio.

13 Veja o quadro no fim desta seção.

Padrões de construção de componente. Recomendam métodos para combinar componentes da WebApp (por exemplo, objetos de conteúdo, funções) com componentes de composição. Quando a funcionalidade de processamento de dados é necessária em uma WebApp, os padrões de projeto arquiteturais e no nível de componente propostos por [BUS96], [GAM95] e outros são aplicáveis.

Padrões de navegação. Apóiam-se no projeto de NSUs, links de navegação e fluxo de navegação global da WebApp.

Padrões de apresentação. Apóiam a apresentação de conteúdo à medida que este é exibido ao usuário via interface. Padrões nessa categoria tratam do modo como organizar a interface com o usuário e controlar funções para melhor usabilidade; como mostrar o relacionamento entre uma ação de interface e objetos de conteúdo que ela afeta; como estabelecer hierarquias de conteúdo efetivas e muitas outras coisas.

Padrões de comportamento/usuário-interação. Apóiam o projeto de interação do usuário com a máquina. Padrões nesta categoria tratam do modo como a interface informa o usuário das conseqüências de uma ação específica; como um usuário expande o conteúdo baseado no contexto de uso e desejos do usuário; como melhor descrever o destino que está implícito em uma ligação; como informar o usuário sobre o estado de uma interação em andamento e outras.

Fontes de informação sobre padrões de projeto hipermídia têm-se expandido enormemente nos últimos anos. Leitores interessados devem consultar [GAR97], [PER99] e [GER00].

Ferramentas de Software

Repositórios de Padrões de Projeto de Hipermídia

O site IAWiki (*http://iawiki.net/WebsitePatterns*) é um espaço de discussão colaborativa para arquitetos de informação que contém muitos recursos úteis. Entre eles, estão links para um certo número de catálogos e repositórios de padrões de hipermídia úteis. Centenas de padrões de projeto estão representados:

Repositório de Padrões de Projeto de Hipermídia *http://www.designpattern.lu.unisi.ch/*

Padrões de Interação de Tom Erickson *http://www.pliant.org/personal/Tom_Erickson/InteractionPatterns.html*

Padrões de Projeto na Web de Martijn vanWelie *http://www.welie.com/patterns/*

Melhoria de Sistemas de Informação na Web com Padrões Navegacionais *http://www8.org/w8-papers/5b-hypertextmedia/improving/improving.html*

Linguagem de Padrões HTML 2.0 *http://www.anamorph.com/docs/patterns/default.html*

Uso Comum *http://www.mit.edu/_jtidwell/interaction_patterns.html*

Padrões para Sites Pessoais na Web *http://www.rdrop.com/_half/Creations/Writings/Web.patterns/index.html*

Indexação de Linguagem de Padrões *http://www.cs.brown.edu/_rms/InformationStructures/Indexing/Overview.html*

19.10 Método de Projeto de Hipermídia Orientado a Objetos (Object-Oriented Hypermedia Design Method — OOHDM)

Um certo número de métodos de projeto para aplicações na Web foi proposto na década passada. Até agora, nenhum método simples conseguiu sobressair-se. Nesta seção apresentamos um breve panorama de um dos mais discutidos métodos de projeto de WebApps — OOHDM.[14]

Método de Projeto de Hipermídia Orientado a Objetos (*Object-Oriented Hypermedia Design Method* — OOHDM) foi originalmente proposto por Daniel Schwabe e seus colegas [SCH95, SCH98]. OOHDM é composto de quatro diferentes atividades de projeto: projeto conceitual, projeto nave-

14 Uma comparação abrangente de dez métodos de projeto de hipermídia foi desenvolvida por Kock [KOC99].

Document shared on www.docsity.com

gacional, projeto de interface abstrata e implementação. Um resumo dessas atividades de projeto é mostrado na Figura 19.11 e discutido brevemente nas seções seguintes.

19.10.1 Projeto Conceitual para OOHDM

O *projeto conceitual* de OOHDM cria uma representação dos subsistemas, classes e relacionamentos que define o domínio da aplicação da WebApp. UML pode ser usada[15] para criar diagramas de classe apropriados, representações de classes de agregação e composição, diagramas de colaboração e outras informações que descrevem o domínio de aplicação (veja a Parte 2 deste livro para mais detalhes).

Como um exemplo simples de projeto conceitual OOHDM, vamos considerar outra vez a aplicação de e-comércio do *CasaSeguraGarantida.com*. Um "esquema conceitual" parcial do *CasaSeguraGarantida.com* é apresentado na Figura 19.12. Os diagramas de classe, agregações e informação relacionada desenvolvidos como parte da análise da WebApp são reusados durante o projeto conceitual para representar relacionamentos entre classes.

19.10.2 Projeto Navegacional de OOHDM

Projeto Navegacional identifica um conjunto de "objetos" derivados das classes definidas no projeto conceitual. Uma série de "classes navegacionais" ou "nós" são definidos para encapsular esses objetos. A UML pode ser usada para criar casos de uso apropriados, diagramas de estado e diagramas de seqüência — todas as representações que apóiam o projetista no melhor entendimento dos requisitos navegacionais. Além disso, padrões de projeto para projeto de navegação podem ser usados à medida que o projeto é desenvolvido. OOHDM usa um conjunto de classes de navegação predefinido — nós, vínculos âncoras e estruturas de acesso [SCH98]. Estruturas de acesso são mais elaboradas e incluem mecanismos como um índice da WebApp, um mapa do site ou um passeio guiado.

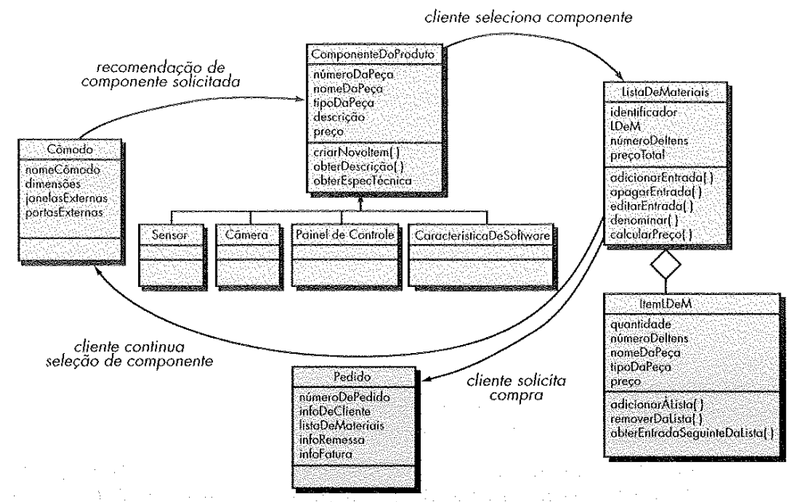
Uma vez definidas as classes de navegação, OOHDM "estrutura o espaço navegacional pelo agrupamento de objetos de navegação em conjuntos denominados contextos" [SCH98]. Schwabe descreve um *contexto* da seguinte maneira:

FIGURA 19.11 Resumo do método OOHDM (adaptado de ([SCH95])

	Projeto conceitual	Projeto navegacional	Projeto de interface abstrata	Implementação
Produtos de trabalho	Classes, subsistemas, relacionamentos, atributos	Ligações de nós, estruturas de acesso, contextos navegacionais, transformações navegacionais	Objetos de interface abstrata, respostas a eventos externos, transformações	WebApp executável
Mecanismos de projeto	Classificação, composição, agregação, generalização, especialização	Mapeamento entre objetos conceituais e navegacionais	Mapeamento entre objetos navegacionais e perceptíveis	Recurso fornecido pelo ambiente alvo
Interesses de projeto	Modelagem semântica do domínio da aplicação	Leva em conta o perfil e a tarefa do usuário Ênfase em aspectos cognitivos	Modelagem de objetos perceptíveis, implementação de metáforas escolhidas	Correção, desempenho da aplicação, completeza

15 OOHDM não prescreve uma notação específica, no entanto, o uso de UML é comum quando esse método é aplicado.

FIGURA 19.12 Esquema conceitual parcial do *CasaSeguraGarantida.com*



Cada definição de contexto abrange, além dos elementos incluídos nele, a especificação de sua estrutura de navegação interna, um ponto de entrada, restrições de acesso em termos de classes e operações de usuário, e uma estrutura de acesso associada.

Um gabarito de contexto (análogo às fichas CRC discutidas no Capítulo 8) é desenvolvido e pode ser usado para rastrear os requisitos de navegação de cada categoria de usuário pelos vários contextos definidos no OOHDM. Fazendo isso, caminhos de navegação específicos (que chamamos de WoN na Seção 19.7.1) aparecem.

19.10.3 Projeto e Implementação de Interface Abstrata

A atividade de *projeto de interface abstrata* especifica os objetos da interface que o usuário vê à medida que a interação da WebApp ocorre. Um modelo formal de objetos de interface, chamado de *visão abstrata de dados* (*abstract data view* — ADV), é usado para representar o relacionamento entre objetos de interface e objetos de navegação, e as características comportamentais dos objetos de interface.

O modelo ADV define um "leiaute estático" [SCH98] que representa a metáfora de interface e inclui uma representação de objetos de navegação na interface e a especificação dos objetos de interface (por exemplo, menus, botões, ícones) que apóiam a interação e navegação. Além disso, o modelo ADV contém um componente comportamental (similar ao diagrama de estado UML) que indica como os eventos externos "disparam a navegação e quais transformações na interface ocorrem quando o usuário interage com a aplicação" [SCH01]. Para uma discussão detalhada do ADV, consulte [SCH98] e [SCH01].

A atividade de *implementação* do OOHDM representa uma iteração de projeto específica do ambiente no qual a WebApp vai operar. Classes, navegação e interface são todas caracterizadas de modo que possam ser construídas para o ambiente cliente/servidor, sistemas operacionais, software de apoio, linguagens de programação e outras características ambientais relevantes ao problema.

Document shared on www.docsity.com

19.11 Métricas de Projeto de WebApp

Métricas de projeto devem ser caracterizadas para que forneçam aos engenheiros da Web uma indicação de qualidade em tempo real. Em essência, um conjunto útil de medidas e métricas apresenta respostas quantitativas às seguintes questões:

- A interface do usuário favorece a usabilidade?
- A estética da WebApp é apropriada para o domínio da aplicação e agradável ao usuário?
- O conteúdo é projetado de modo que conduza a maioria da informação com o mínimo de esforço?
- A navegação é eficiente e direta?
- A arquitetura da WebApp foi projetada para acomodar as metas e os objetivos especiais dos usuários da WebApp, a estrutura de conteúdo e funcionalidade, e o fluxo de navegação necessário para usar o sistema efetivamente?
- Os componentes são projetados para que reduzam a complexidade procedimental e enfatizem a correção, confiabilidade e desempenho?

Hoje, cada uma dessas questões pode ser tratada qualitativamente,[16] mas um conjunto de métricas validadas que poderiam fornecer repostas quantitativas ainda não existe. Métricas para projeto de WebApp estão ainda começando, e poucas têm sido amplamente validadas. O leitor interessado pode consultar [IVO01] e [MEN01] para uma amostra de métricas propostas de projeto de WebApp.

Ferramentas de Software

Métricas Técnicas de WebApps

Objetivo: Apoiar engenheiros da Web no desenvolvimento de métricas significativas de WebApp que fornecem visão aprofundada da qualidade global de uma aplicação.

Mecânica: A mecânica das ferramentas varia.

Ferramentas Representativas[17]

Netmechanic Tools, desenvolvida por Netmechanic (*www.netmechanic.com*), é uma coleção de ferramentas que ajudam a melhorar o desempenho do site Web, abordando tópicos específicos de implementação.

NIST Web Metrics Testbed, desenvolvida por The National Institute of Standards and Technology (*zing.ncsl.nist.gov/WebTools/*), engloba a seguinte coleção de ferramentas úteis que estão disponíveis para serem baixadas de:

Web Static Analyzer Tool (WebSAT) — checa as páginas HTML da Web com base em diretrizes típicas de usabilidade.

Web Category Analysis Tool (WebCAT) — permite ao engenheiro de usabilidade construir e conduzir uma análise de categoria Web.

Web Variable Instrumenter Program (WebVIP) — instrumenta um site Web para captar um registro da interação do usuário.

Framework for Logging Usability Data (FLUD) — implementa um formatador e analisador de arquivo para representação de registros de interação do usuário.

VisVIP Tool — produz uma visualização 3D dos caminhos de navegação do usuário ao longo do site.

TreeDec — adiciona apoio de navegação às páginas de um site Web.

16 Veja o Capítulo 16 (Seção 16.4) e neste capítulo na Seção 19.1.1 para uma discussão da qualidade de WebApp.

17 As ferramentas mencionadas aqui não representam uma recomendação, mas, em vez disso, uma exemplificação de ferramentas dessa categoria.

19.12 Resumo

A qualidade de uma WebApp — definida em termos de usabilidade, funcionalidade, confiabilidade, eficiência, manutenibilidade, segurança, escalabilidade e prazo para colocação no mercado — é introduzida durante o projeto. Para atingir esses atributos de qualidade, um bom projeto de WebApp deve apresentar simplicidade, consistência, identidade, robustez, navegabilidade e atração visual.

Projeto de interface descreve a estrutura e organização da interface com o usuário. Ele inclui a representação de leiaute de tela, uma definição dos modos de interação e uma descrição dos mecanismos de navegação.

Projeto de estética, também chamado de projeto gráfico, descreve o "aspecto" da WebApp e inclui esquemas de cor, leiaute geométrico, tamanho de texto, fonte e colocação, uso de gráficos e decisões estéticas relacionadas. Um conjunto de diretrizes de projeto gráfico fornece a base para uma abordagem de projeto.

Projeto de conteúdo define o leiaute, a estrutura e o esboço para todo o conteúdo apresentado como parte da WebApp e estabelece os relacionamentos entre objetos de conteúdo. Projeto de conteúdo começa com a representação dos objetos de conteúdo, suas associações e relacionamentos. Um conjunto de navegadores primitivos estabelece a base para o projeto de navegação.

Projeto de arquitetura identifica a estrutura global de hipermídia da WebApp e engloba tanto a arquitetura de conteúdo quanto a arquitetura da WebApp. Estilos arquiteturais de conteúdo incluem estruturas linear, em malha, hierárquica e em rede. A arquitetura da WebApp descreve uma infra-estrutura que possibilita a um sistema ou aplicação baseada na Web atingir seus objetivos de negócio.

Projeto de navegação representa o fluxo navegacional entre objetos de conteúdo e todas as funções da WebApp. Navegação é definida pela descrição de um conjunto de unidades de semântica navegacional. Cada unidade é composta de modos de navegação, vínculos e nós navegacionais. Mecanismos de sintaxe de navegação são usados para efetivar a navegação descrita como parte da semântica.

Projeto de componente desenvolve a lógica de processamento detalhado necessário para implementar componentes funcionais da WebApp. Técnicas de projeto descritas no Capítulo 11 aplicam-se à engenharia de componentes de WebApp.

Padrões de projeto de WebApp englobam padrões de projeto genéricos que se aplicam a todos os tipos de software e padrões de hipermídia especialmente relevantes para WebApps. Padrões de projeto de arquitetura, navegação, componente, apresentação e comportamento/usuário têm sido propostos.

O Método de Projeto de Hipermídia Orientado a Objetos (Object-Oriented Hypermedia Design Method — OOHDM) é um de vários métodos propostos para projeto de WebApp. OOHDM sugere um processo de projeto que inclui projeto conceitual, projeto navegacional, projeto de interface abstrata e implementação.

Métricas de projeto de engenharia da Web estão em seu início e têm ainda de ser completamente validadas. No entanto, uma variedade de medidas e métricas tem sido proposta para tratar cada uma das atividades de projeto discutidas neste capítulo.

Referências Bibliográficas

[AME96] Amento, B. *et al.*, "Fitt's Law", *CS 5724: Models and Theories of Human-Computer Interactions*, Virginia Tech, 1996, disponível em http://ei.cs.vt.edu/_cs5724/g1/.

[BAG01] Baggerman, L. e Bowman, S., *Web Design That Works*, Rockport Publishers, 2001.

[BUS96] Buschmann, F. *et al.*, *Pattern-Oriented Software Architecture*, Wiley, 1996.

[CAC02] Cachero, C. *et al.*, "Conceptual Navigation Analysis: a Device and Platform Independent Navigation Specification", *Proc. 2nd Intl. Workshop on Web-Oriented Technology*, jun. 2002, disponível em www.dsic.upv.es/_west/iwwost02/papers/cachero.pdf.

Document shared on www.docsity.com

[CLO01] Cloninger, C., *Fresh Styles for Web Designers*, New Riders Publishing, 2001.
[DIX99] Dix, A., "Design of User Interfaces for the Web", *Proc. Of User Interfaces to Data Systems Conference*, set. 1999, disponível em http://www.comp.lancs. ac.uk/computing/users/dixa/topics/webarch/.
[FIT54] Fitts, P., "The Information Capacity of the Human Motor System in Controlling the Amplitude of Movement", *Journal of Experimental Psychology*, v. 47, 1954, p. 381-391.
[FOW03] Fowler, M. *et al.*, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.
[GAL02] Galitz, W., *The Essential Guide to User Interface Design*, Wiley, 2002.
[GAM95] Gamma, E. *et al.*, *Design Patterns*, Addison-Wesley, 1995.
[GAR97] Garrido, A., Rossi, G. e Schwabe, D., "Patterns Systems for Hypermedia", 1997, disponível em www.inf.puc-rio.br/_schwabe/papers/PloP97.pdf.
[GER00] German, D. e Cowan, D., "Toward a Unified Catalog of Hypermedia Design Patterns", *Proc. 33rd Hawaii Intl. Conf. on System Sciences*, IEEE, v. 6, Maui, Hawaii, jun. 2000, disponível em www.turingmachine.org/_dmg/research/papers/dmg_hicss2000.pdf.
[GNA99] Gnaho, C. e Larcher, F., "A User-Centered Methodology for Complex and Customizable Web Engineering", *Proc. 1st ICSE Workshop on Web Engineering*, ACM, Los Angeles, maio 1999.
[HEI02] Heinicke, E., *Layout: Fast Solutions for Hands-On Design*, Rockport Publishers, 2002.
[IVO01] Ivory, M., Sinha, R. e Hearst, M., "Empirically Validated Web Page Design Metrics", ACM SIGCHI '01, Seattle, WA, abr. 2001, disponível em http://www.rashmisinha.com/articles/WebTangoCHI01.html.
[JAC02] Jacyntho, D., Schwabe, D. e Rossi, G., "An Architecture for Structuring Complex Web Applications", 2002, disponível em http://www2002.org/CDROM/alternate/478/.
[KAI02] Kaiser, J., "Elements of Effective Web Design", About, Inc., 2002, disponível em http://webdesign.about.com/library/weekly/aa091998.htm.
[KAL03] Kalman, S., *Web Security Field Guide*, Cisco Press, 2003.
[KOC99] Koch, N., "A Comparative Study of Methods for Hypermedia Development", Technical Report 9905, Ludwig-Maximilians Universität, Munique, Alemanha, 1999, disponível em http://www.dsic.upv.es/_west2001/iwwost01/files/contributions/NoraKoch/hypdev.pdf.
[KRA88] Krasner, G. e Pope, S., "A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80", *Journal of Object-Oriented Programming*, v. 1, n. 3, ago./set. 1988, p. 26-49.
[LOW98] Lowe, D. e Hall, W. (Eds.), *Hypertext and the Web — An Engineering Approach*, John Wiley & Sons, 1998.
[MCC01] McClure, S., Scambray, J. e Kurtz, G., *Hacking Exposed*, McGraw-Hill/Osborne, 2001.
[MEN01] Mendes, E., Mosley, N. e Counsell, S., "Estimating Design and Authoring Effort", *IEEE Multimedia*, jan./mar. 2001, p. 50-57.
[MIL00] Miller, E., "The Website Quality Challenge", Software Research, Inc., 2000, disponível em http://www.soft.com/eValid/Technology/White.Papers/website.quality.challenge.html.
[NIE96] Nielsen, J. e Wagner, A., "User Interface Design for the WWW", *Proc. CHI '96 Conf. On Human Factors in Computing Systems*, ACM Press, 1996, p. 330-331.
[NIE00] _____, *Designing Web Usability*, New Riders Publishing, 2000.
[NOR02] Northcutt, S. e Novak, J., *Network Intrusion Detection*, New Riders Publishing, 2002.
[OFF02] Offutt, J., "Quality Attributes of Web Software Applications", *IEEE Software*, mar./abr., 2002, p. 25-32.
[OLS98] Olsina, L., "Building a Web-Based Information System Applying the Hypermedia Flexible Process Modeling Strategy", Proc. *1st Intl. Workshop on Hypermedia Development*, 1998.
[OLS99] _____, *et al.*, "Specifying Quality Characteristics and Attributes for Web Sites", *Proc. 1st ICSE Workshop on Web Engineering*, ACM, Los Angeles, maio 1999.
[PER99] Perzel, K. e Kane, D. "Usability Patterns for Applications on the World Wide Web", 1999, disponível em http://jerry.cs.uiuc.edu/_plop/plop99/proceedings/Kane/perzel_kane.pdf.
[POW00] Powell, T., *Web Design*, McGraw-Hill/Osborne, 2000.
[RAS00] Raskin, J., *The Humane Interface*, Addison-Wesley, 2000.
[RHO98] Rho, Y. e Gedeon, T., "Surface Structures in Browsing the Web", *Proc. Australasian Computer Human Interaction Conference*, IEEE, dez. 1998.
[SCH95] Schwabe, D. e Rossi, G., "The Object-Oriented Hypermedia Design Model", *CACM*, v 38, n 8, ago. 1995, p. 45-46.
[SCH98] _____, Developing Hypermedia Applications Using OOHDM, *Proc. Workshop on Hypermedia Development Process, Methods and Models, Hypertext '98*, 1998, disponível em http://citeseer.nj.nec.com/schwabe 98developing.html.
[SCH01] _____ e Barbosa, S. "Systematic Hypermedia Application Design Using OOHDM", 2001, disponível em http://www-di.inf.puc-rio.br/~schwabe/HT96WWW/ section1.html.
[TIL00] Tillman, H. N, "Evaluating Quality on the Net", Babson College, maio 30, 2000, disponível em http://www.hopetillman.com/findqual.html#2.
[TOG01] Tognozzi, B., "First Principles", *askTOG*, 2001, disponível em http://www.asktog.com/basics/firstPrinciples.html.
[WMT02] Web Mapping Testbed Tutorial, 2002, disponível em http://www.webmapping.org/vcgdocuments/vcgTutorial/.
[ZHA02] Zhao, H., "Fitt's Law: Modeling Movement Time in HCI", *Theories in Computer Human Interaction*, University of Maryland, out. 2002, disponível em http://www.cs.umd.edu/class/fall2002/cmsc838s/tichi/fitts.html.

Problemas e Pontos a Considerar

19.1. Por que o "ideal artístico" é uma filosofia de projeto insuficiente quando modernas WebApps são construídas? Há algum caso em que o ideal artístico é a filosofia a ser seguida?

19.2. Neste capítulo discutimos uma ampla variedade de atributos de qualidade de WebApps. Selecione três que você acredita serem os mais importantes e faça uma argumentação que explique por que cada um deveria ser enfatizado em um trabalho de projeto de engenharia da Web.

19.3. Adicione no mínimo cinco questões ao Projeto WebApp — Checklist de Qualidade apresentado no quadro da Seção 19.1.1.

19.4. Revise os princípios de projeto de interface de Tognozzi, discutidos na Seção 19.3.1. Considere cada princípio de uma WebApp operacional com a qual você está familiarizado. Classifique a WebApp (use as classificações A, B, C, D ou F) relativa ao grau em que ela satisfez ao princípio. Explique a razão para cada classificação.

19.5. Projete um protótipo de interface para a WebApp *CasaSeguraGarantida.com*. Tente ser criativo, mas, ao mesmo tempo, esteja certo de que a interface está de acordo com os princípios de um bom projeto de interface.

19.6. Você tem encontrado mecanismos de controle de interface diferentes dos mencionados na Seção 19.3.2? Em caso afirmativo, descreva-os brevemente.

19.7. Você é um projetista de WebApp para uma empresa de ensino a distância. Você pretende implementar um "motor de aprendizado" baseado na Internet que vai lhe possibilitar enviar o conteúdo do curso a estudantes. O motor de aprendizado fornece a infra-estrutura básica para entregar conteúdo de aprendizado em qualquer assunto (projetistas de conteúdo vão preparar o conteúdo adequado). Desenvolva um protótipo de projeto de interface para o motor de aprendizado.

19.8. Qual é o site da Web esteticamente mais agradável que você já visitou e por quê?

19.9. Considere o objeto de conteúdo **pedido**, gerado depois que um usuário do *CasaSeguraGarantida.com* completou a seleção de todos os componentes e está pronto para finalizar a sua compra. Desenvolva uma descrição UML para **pedido** junto com todas as representações de projeto apropriadas.

19.10. Qual é a diferença entre arquitetura de conteúdo e arquitetura da WebApp?

19.11. Reconsidere o "motor de aprendizado" descrito no Problema 19.7, selecione uma arquitetura de conteúdo que poderia ser apropriada para a WebApp. Discuta por que você fez essa escolha.

19.12. Use a UML para desenvolver três ou quatro representações de projeto para objetos de conteúdo que poderiam ser encontrados à medida que o "motor de aprendizado" descrito no Problema 19.7 é projetado.

19.13. Faça uma pesquisa adicional sobre a arquitetura MVC e decida se ela seria apropriada para uma arquitetura WebApp para o "motor de aprendizado" discutido no Problema 19.7

19.14. Qual é a diferença entre sintaxe de navegação e semântica de navegação?

19.15. Defina duas ou três NSUs para a WebApp do *CasaSeguraGarantida.com*. Descreva cada uma com alguns detalhes.

19.16. Pesquise e apresente dois ou três padrões de projeto de hipermídia completos para sua classe.

Leituras e Fontes de Informação Adicionais

Embora centenas de livros tenham sido escritos sobre "projeto Web", muito poucos discutem métodos técnicos significativos para fazer o trabalho de projeto. Na melhor das hipóteses, uma variedade de diretrizes úteis para projeto de WebApp é apresentada, exemplos que valem a pena de páginas da Web e programação Java são mostrados, e detalhes técnicos importantes para implementação de WebApps modernas são discutidos. Entre as muitas ofertas nessa categoria, a discussão enciclopédica de Powell [POW00] vale a pena considerar. Além disso, livros de Galitz [GAL02], Heinicke [HEI02], Schmitt (*Designing CSS WebPages*, New Riders Publishing, 2002), Donnelly (*Designing Easy-to-Use Websites*, Addison-Wesley, 2001) e Nielsen [NIE00] fornecem muita diretriz útil.

A visão ágil de projeto (e outros tópicos) de WebApps é apresentada por Wallace e seus colegas (*Extreme Programming for Web Projects*, Addison-Wesley, 2003). Conallen (*Building WebApplications with UML*, segunda edição, Addison-Wesley, 2002) e Rosenberg e Scott (*ApplyingUse-Case Driven Object Modeling with UML*, Addison-Wesley, 2001) apresentam exemplos detalhados de WebApps modeladas usando UML.

Document shared on www.docsity.com

Van Duyne e seus colegas (*The Design of Sites: Patterns, Principles and Processes*, Addison-Wesley, 2002) escreveram um excelente livro que cobre os pontos mais importantes do processo de projeto de engenharia da Web. Modelos de processo de projeto e padrões de projeto são abordados em detalhe. Wodtke (*Information Architecture*, New Riders Publishing, 2003), Rosenfeld e Morville (*Information Architecture for the World Wide Web*, O'Reilly & Associates, 2002) e Reiss (*Practical Information Architecture*, Addison-Wesley, 2000) tratam da arquitetura de conteúdo e outros tópicos.

Técnicas de projeto são também mencionadas em livros escritos sobre ambientes de desenvolvimento específicos. Leitores interessados devem consultar livros sobre J2EE, Java, ASP.NET, CSS, XML, Perl, e uma variedade de aplicações de criação de WebApp (*Dreamweaver, HomePage, Frontpage, GoLive, Macro-Media Flash* etc.) com dicas de projeto úteis.

Uma ampla variedade de fontes de informação sobre projeto de engenharia da Web está disponível na Internet. Uma lista atualizada de referências da World Wide Web relevantes pode ser encontrada no site deste livro: **http://www.mhhe.com/pressman.**

Teste de Aplicações Web

CAPÍTULO 20

Conceitos-Chave

características de erros ... 457
dimensões de qualidade... 456
estratégias 457
teste
de banco de dados 462
de carga..... 475
de compatibilidade 467
de configuração 472
de conteúdo 461
de desempenho 474
de esforço 476
de interface com o usuário 463
de navegação 470
de usabilidade 466
no nível de componente 468

Há uma urgência que sempre permeia o processo de engenharia da Web. À medida que formulação, planejamento, análise, projeto e construção são conduzidos, os interessados — preocupados com a concorrência de outras WebApps, coagidos pelas exigências do cliente e temerosos de que vão perder uma janela de mercado — pressionam para obter a WebApp on-line. Como conseqüência, atividades técnicas que freqüentemente ocorrem mais tarde no processo de engenharia da Web, tais como teste de WebApp, recebem algumas vezes pouco prazo. Isso pode ser um erro catastrófico. Para evitá-lo, a equipe de engenharia da Web precisa garantir que cada produto de trabalho de WebE exiba alta qualidade. Wallace e seus colegas [WAL03] observam isso quando afirmam:

> Teste não deveria esperar até que o projeto esteja terminado. Comece a testar antes que você escreva uma linha de código. Teste constante e efetivamente, e você vai desenvolver um site Web muito mais duradouro.

Como modelos de análise e projeto não podem ser testados no sentido clássico, a equipe de engenharia da Web deve conduzir revisões técnicas formais (Capítulo 26), bem como testes executáveis. A intenção é descobrir e corrigir erros antes de a WebApp estar disponível para seus usuários finais.

Panorama

O que é? Teste de WebApp é um conjunto de atividades relacionadas com um único objetivo: descobrir erros no conteúdo, na função, na usabilidade, na navegabilidade, no desempenho, na capacidade e na segurança de WebApp. Para conseguir isso, uma estratégia de teste que englobe tanto revisões quanto teste executável é aplicada ao longo do processo de engenharia da Web.

Quem faz? Engenheiros da Web e outros interessados no projeto (gerentes, clientes, usuários finais), todos participam do teste da WebApp.

Por que é importante? Se usuários finais encontram erros que questionam sua fé na WebApp, eles vão para outro lugar em busca do conteúdo e da função de que precisam, e a WebApp falhará. Por essa razão, engenheiros da Web devem trabalhar para eliminar tantos erros quanto possível antes de a WebApp ficar on-line.

Quais são os passos? O processo de teste de WebApp começa focalizando tópicos visíveis ao usuário da WebApp e prossegue com testes que exercitam a tecnologia e a infra-estrutura. Sete passos de teste são realizados: teste de conteúdo, interface, navegação, componente, configuração, desempenho e segurança.

Qual é o produto do trabalho? Em algumas instâncias um plano de teste de uma WebApp é produzido. Em todas as instâncias, um conjunto de casos de testes é desenvolvido para cada passo e um arquivo de resultados mantido para uso futuro.

Como tenho certeza de que fiz corretamente? Embora você nunca possa estar seguro de que realizou todo o teste necessário, certifique-se de que o teste descobriu erros (e que esses foram corrigidos). Além disso, se você tiver estabelecido um plano de teste, examine-o para garantir que todos os testes planejados foram conduzidos.

Document shared on www.docsity.com

20.1 Conceitos de Teste de WebApp

No Capítulo 13, mencionamos que teste é o processo de exercitar um software com o objetivo de encontrar (e em última análise corrigir) erros. Essa filosofia fundamental não se modifica para WebApps. De fato, como sistemas e aplicações baseados na Web residem numa rede e interoperam com muitos sistemas operacionais diferentes, navegadores (ou outros dispositivos de interface tais como PDAs ou telefones móveis), plataformas de hardware, protocolos de comunicação e aplicações "de retaguarda" à procura de erros representam um desafio significativo para engenheiros da Web.

Para entendermos os objetivos de teste no contexto de engenharia da Web, devemos considerar as muitas dimensões de qualidade de WebApp.[1] No contexto dessa abordagem, consideramos dimensões de qualidade particularmente relevantes em qualquer discussão de teste para trabalho de engenharia da Web. Também consideramos a natureza dos erros encontrados como conseqüência do teste e a estratégia de teste aplicada para descobrir esses erros.

20.1.1 Dimensões de Qualidade

Qualidade é incorporada em uma aplicação Web como conseqüência de bom projeto. É avaliada pela aplicação de uma série de revisões técnicas que examinam os elementos do modelo de projeto e pela aplicação de um processo de teste discutido ao longo deste capítulo. Tanto revisões quanto teste examinam uma ou mais das seguintes dimensões de qualidade [MIL00]:

? Como avaliamos a qualidade no contexto de uma WebApp e seu ambiente?

- *Conteúdo* é examinado tanto no nível sintático quanto semântico. No nível sintático, ortografia, pontuação e gramática são avaliadas em documentos baseados em texto. No nível semântico, são verificadas correção (da informação apresentada), consistência (ao longo de todo o objeto de conteúdo e objetos relacionados) e ausência de ambigüidade.
- *Função* é testada para descobrir erros que indicam falta de conformidade com os requisitos do cliente. Cada função da WebApp é avaliada quanto à correção, instabilidade e conformidade geral a normas de implementação apropriadas (por exemplo, normas de linguagem Java ou XML).
- *Estrutura* é avaliada para garantir que forneça adequadamente o conteúdo e função da WebApp; que seja extensível e possa ser mantida à medida que novo conteúdo ou funcionalidade é adicionado.
- *Usabilidade* é testada para garantir que cada categoria de usuário seja apoiada pela interface; que o usuário possa aprender e aplicar toda sintaxe e semântica de navegação necessária.
- *Navegabilidade* é testada para garantir que toda sintaxe e semântica de navegação sejam exercitadas para descobrir quaisquer erros de navegação (por exemplo, links inativos, impróprios, errados).
- *Desempenho* é testado sob uma variedade de condições de operação, configurações e carga para garantir que o sistema responda à interação com o usuário e manipule carregamento máximo sem degradação operacional inaceitável.
- *Compatibilidade* é testada pela execução da WebApp em uma variedade de diferentes configurações hospedeiras tanto do lado do cliente quanto do lado do servidor. A intenção é encontrar erros específicos de determinada configuração hospedeira.
- *Interoperabilidade* é testada para garantir que a WebApp tenha interfaces adequadas com outras aplicações e/ou bancos de dados.
- *Segurança* é testada para avaliar potenciais vulnerabilidades e tentativa de explorar cada uma. Qualquer tentativa de invasão bem-sucedida é considerada falha de segurança.

Uma estratégia e tática de teste da WebApp tem sido desenvolvida para exercitar cada uma dessas dimensões de qualidade e é discutida posteriormente neste capítulo.

1 Qualidade de WebApp também foi considerada no Capítulo 19.

> "Inovação é um negócio agridoce para testadores de software. Quando parece que sabemos como testar uma tecnologia particular, chega uma nova [a Internet e as WebApps] e todas as possibilidades perdem sentido."
>
> James Bach

? O que torna erros encontrados durante a execução de WebApps um tanto diferentes daqueles encontrados em software convencional?

20.1.2 Erros em um Ambiente de WebApp

Já mencionamos que a intenção principal de teste em qualquer contexto de software é detectar erros (e corrigi-los). Erros encontrados como conseqüência de teste bem-sucedido de WebApp têm um certo número de características exclusivas [NGU00]:

1. Como muitos tipos de teste de WebApp detectam problemas que são primeiro evidenciados do lado do cliente (isto é, via interface implementada em um navegador, um PDA ou um telefone móvel específico), o engenheiro Web se depara com o efeito do erro, não o erro em si.
2. Como uma WebApp é implementada em um certo número de diferentes configurações e diferentes ambientes, pode ser difícil ou impossível reproduzir um erro fora do ambiente no qual foi originalmente encontrado.
3. Embora alguns erros sejam resultado de projeto incorreto ou de codificação HTML (ou outra linguagem de programação) imprópria, muitos erros podem ser rastreados até a configuração da WebApp.
4. Como WebApps residem em uma arquitetura cliente/servidor, erros podem ser difíceis de ser rastreados pelas três camadas arquiteturais: o cliente, o servidor ou a rede em si.
5. Alguns erros são devidos ao *ambiente operacional estático* (isto é, a configuração específica na qual o teste é conduzido), enquanto outros são atribuíveis ao ambiente operacional dinâmico (isto é, carregamento instantâneo de recurso ou erros relacionados ao tempo).

Esses cinco atributos sugerem que o ambiente desempenha um importante papel no diagnóstico de todos os erros encontrados durante o processo de engenharia da Web. Em algumas situações (por exemplo, teste de conteúdo), a posição do erro é óbvia, mas em muitos outros tipos de teste de WebApp (por exemplo, teste de navegação, teste de desempenho, teste de segurança) a causa subjacente do erro pode ser consideravelmente mais difícil de determinar.

20.1.3 Estratégia de Teste

A estratégia de teste de WebApp adota os princípios básicos de todos os testes de software (Capítulo 13) e aplica uma estratégia e táticas recomendadas para sistemas orientados a objetos (Capítulo 14). Os seguintes passos resumem a abordagem:

PONTO CHAVE

A estratégia global de teste de WebApp pode ser resumida nos dez passos mencionados aqui.

1. O modelo de conteúdo da WebApp é revisado para descobrir erros.
2. O modelo de interface é revisado para garantir que todos os casos de uso possam ser acomodados.
3. O modelo de projeto da WebApp é revisado para descobrir erros de navegação.
4. A interface com o usuário é testada para descobrir erros nos mecanismos de apresentação e/ou navegação.
5. Componentes funcionais selecionados são submetidos a teste de unidade.
6. Navegação ao longo da arquitetura é testada.
7. A WebApp é implementada por uma variedade de diferentes configurações ambientais e testada quanto à compatibilidade com cada configuração.
8. Testes de segurança são conduzidos em uma tentativa de explorar vulnerabilidades na WebApp ou em seu ambiente.
9. Testes de desempenho são conduzidos.
10. A WebApp é testada por uma população de usuários finais controlada e monitorada; os resultados de sua interação com o sistema são avaliados quanto a erros de conteúdo e navegação, preocupações com usabilidade, compatibilidade e confiabilidade, e desempenho da WebApp.

Veja na Web

Excelentes artigos sobre teste de WebApp podem ser encontrados em www.stickyminds.com/testing.asp.

Document shared on www.docsity.com

Como muitas WebApps evoluem continuamente, teste de WebApp é uma atividade constante conduzida pela equipe de suporte da Web que usa testes de regressão derivados dos desenvolvidos quando a WebApp foi inicialmente construída.

20.1.4 Planejamento de Teste

O uso da palavra *planejamento* (em qualquer contexto) é anátema para alguns desenvolvedores da Web. Como mencionamos nos capítulos anteriores, esses desenvolvedores começam esperando que uma WebApp arrasadora surja. Um engenheiro da Web reconhece que o planejamento estabelece um guia para todo o trabalho que se segue. Isso vale o esforço. Em seu livro sobre teste de WebApp, Splaine e Jaskiel [SPL01] afirmam:

> Exceto para os sites mais simples da Web, torna-se rapidamente aparente que alguma espécie de planejamento de teste seja necessária. Com muita freqüência, o número de erros inicialmente encontrados em um teste *ad hoc* é tão grande que nem todos eles são corrigidos na primeira vez em que são detectados. Isso impõe uma responsabilidade adicional naqueles que testam sites e aplicações da Web. Eles não apenas precisam conceber novos testes imaginativos, mas também lembrar como os testes anteriores foram executados a fim de retestar confiavelmente o site/aplicação da Web, e garantir que os erros conhecidos foram removidos e que nenhum erro novo foi introduzido.

PONTO CHAVE

O plano de teste identifica um conjunto de tarefas de teste, os produtos de trabalho a ser desenvolvidos e o modo pelo qual os resultados serão avaliados, registrados e reutilizados.

A questão para todo engenheiro da Web é: Como fazemos para "criar testes imaginativos", e o que esse teste deveria enfocar? A resposta a essas questões estão contidas no plano de teste.

Um plano de teste da WebApp identifica (1) um conjunto de tarefas[2] a ser aplicado quando o teste se inicia, (2) os produtos de trabalho a ser produzidos à medida que cada tarefa de teste é executada, e (3) o modo pelo qual os resultados de teste são avaliados, registrados e reutilizados quando teste de regressão é conduzido. Em alguns casos, o plano de teste é integrado com o plano de projeto. Em outros, o plano de teste é um documento separado.

20.2 O Processo de Teste — Um Resumo

O processo de teste de engenharia da Web começa com testes que exercitam o conteúdo e a funcionalidade da interface imediatamente visível aos usuários finais. À medida que o teste prossegue, tópicos do projeto de arquitetura e de navegação são exercitados. O usuário pode ou não estar ciente desses elementos da WebApp. Finalmente, o foco desloca-se para testes que exercitam capacidades tecnológicas que nem sempre são aparentes aos usuários finais — tópicos de infra-estrutura e de instalação/implementação da WebApp.

> "Em geral, as técnicas de teste de software [Capítulos 13 e 14] usadas com outras aplicações são as mesmas que aquelas usadas com aplicações baseadas na Web... A diferença entre os dois tipos de testes é que as variáveis tecnológicas do ambiente Web se multiplicam."
>
> **Hung Nguyen**

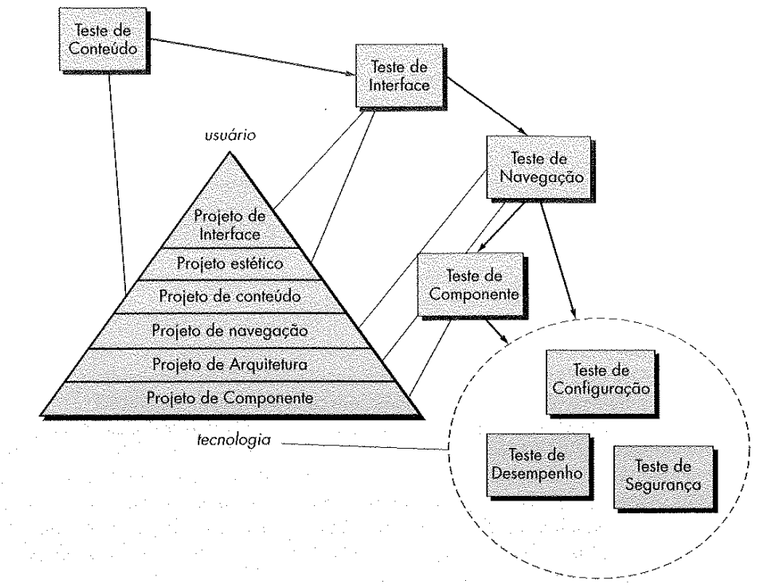
A Figura 20.1 justapõe o processo de teste de WebApp com a pirâmide de projeto discutida no Capítulo 19. Observe que, à medida que o fluxo de teste prossegue da esquerda para a direita e de cima para baixo, os elementos visíveis ao usuário do projeto de WebApp (elementos do topo da pirâmide) são primeiro testados, seguidos pelos elementos do projeto de infra-estrutura.

Teste de conteúdo (e revisões) tenta descobrir erros de conteúdo. Essa atividade de teste é semelhante em muitos pontos à revisão de um documento escrito. De fato, um site de grande porte da Web poderia contratar os serviços de um editor profissional de textos para descobrir erros de grafia, erros gramaticais, de consistência no conteúdo, de representação gráfica e de referência cruzada.

2 Conjuntos de tarefas são discutidos no Capítulo 2. Um termo relacionado — *fluxo de trabalho* — também é usado neste livro para descrever uma série de tarefas necessárias para realizar uma atividade de engenharia de software.

FIGURA 20.1

O processo de teste



Além disso, para examinar o conteúdo estático quanto a erros, esse passo de teste também considera o conteúdo dinâmico derivado dos dados mantidos como parte de um sistema de banco de dados que tenha sido integrado à WebApp.

Teste de interface exercita os mecanismos de interação e valida os tópicos estéticos da interface com o usuário. O objetivo é descobrir erros que resultam dos mecanismos de interação fracamente implementados ou omissões, inconsistências ou ambigüidades que tenham sido introduzidas inadvertidamente na interface.

Teste de navegação aplica casos de uso, derivados como parte da atividade de análise, no projeto de casos de teste que exercitam cada cenário de uso com base no projeto de navegação. Mecanismos de navegação (por exemplo, barras de menu) implementados no leiaute da interface são testados com base nos casos de uso e nas NSUs (Capítulo19) para garantir que quaisquer erros que impeçam a finalização de um caso de uso sejam identificados e corrigidos.

Teste de componente exercita as unidades de conteúdo e funcional na WebApp. Quando as WebApps são consideradas, o conceito de unidade (introduzido no Capítulo 13) se modifica. A "unidade" de escolha dentro da arquitetura de conteúdo (Capítulo 19) é a página da Web. Cada página da Web encapsula conteúdo, ligações de navegação e elementos de processamento (formulários, scripts e *applets*). Uma "unidade" dentro da arquitetura da WebApp pode ser um componente funcional definido que fornece serviço diretamente a um usuário final ou a um componente de infra-estrutura que habilita a WebApp a realizar todas as suas capacidades. Cada componente funcional é testado de modo muito semelhante ao que um módulo individual é testado em software convencional. Na maioria dos casos, testes são orientados a caixa-preta. No entanto, se o processamento é complexo, testes caixa-branca também podem ser usados.[3] Além do teste funcional, capacidades de banco de dados também são exercitadas.

PONTO CHAVE

A estratégia para teste de integração depende da arquitetura da WebApp que foi escolhida durante o projeto.

À medida que a arquitetura da WebApp é construída, testes de navegação e componente são usados como *testes de integração*. A estratégia de teste de integração depende do conteúdo e da arquitetura escolhidos para a WebApp (Capítulo 19). Se a arquitetura de conteúdo tiver sido projetada com uma estrutura linear, em malha ou simplesmente hierárquica, é possível integrar páginas

3 Técnicas de testes caixa-preta e caixa-branca são discutidas no Capítulo 14.

Document shared on www.docsity.com

da Web de um modo muito semelhante ao que usamos para integrar módulos de software convencional. No entanto, se uma arquitetura de hierarquia mista, ou de rede (Web), é usada, o teste de integração é semelhante à abordagem usada para sistemas OO. O teste de linha de controle (*threads*) (Capítulo14) pode ser usado para integrar o conjunto de páginas Web (uma NSU pode ser usada para definir o conjunto apropriado) necessário para responder a um evento do usuário. Cada linha de controle é integrada e testada individualmente. Teste de regressão é aplicado para garantir que efeitos colaterais não ocorram. O teste de agregados integra um conjunto de páginas que colaboram (determinado pelo exame de casos de uso e SNU). Casos de testes são derivados para descobrir erros nas colaborações.

Cada elemento da arquitetura da WebApp é submetido a teste de unidade até onde for possível. Por exemplo, em uma arquitetura MVC (Capítulo 19), os componentes *modelo, visão* e *controlador* são testados individualmente. A partir da integração, o fluxo de controle e de dados através de cada um desses elementos é avaliado em detalhe.

Teste de configuração tenta descobrir erros específicos a um particular ambiente cliente ou servidor. É criada uma matriz de referência cruzada que define todos os prováveis sistemas operacionais, navegadores,[4] plataformas de hardware e protocolos de comunicação. Então, são realizados testes para descobrir erros associados com cada possível configuração.

Teste de segurança incorpora uma série de testes projetados para explorar a vulnerabilidade na WebApp e em seus ambientes. A intenção é demonstrar que uma quebra de segurança é possível.

Teste de desempenho engloba uma série de testes projetados para avaliar (1) como o tempo de resposta e a confiabilidade da WebApp são afetados pelo aumento de tráfego de usuário, (2) que componentes da WebApp são responsáveis pela degradação de desempenho e que características de uso fazem a degradação ocorrer, e (3) como a degradação de desempenho impacta os objetivos e requisitos globais da WebApp.

CONJUNTO DE TAREFAS

Teste de WebApp

1. Revise os requisitos dos interessados. Identifique metas e objetivos-chave do usário. Revise os casos de uso para cada categoria de usuário.
2. Estabeleça prioridades para garantir que cada meta e objetivo do usuário seja adequadamente testado.
3. Defina estratégia de teste de WebApp descrevendo os tipos de testes (Seção 20.2) que serão conduzidos.
4. Desenvolva um plano de teste.
   - Defina um cronograma de teste e atribua responsabilidades a cada teste.
   - Especifique ferramentas automáticas para teste. Defina critérios de aceitação para cada classe de teste.
   - Especifique mecanismos de rastreamento de defeitos.
   - Defina mecanismos de relato de problemas.
5. Execute testes de "unidade".
   - Revise o conteúdo quanto a erros sintáticos e semânticos.
   - Revise o conteúdo quanto a liberações e permissões adequadas.
   - Teste mecanismos de interface quanto à operação correta.
   - Teste cada componente (por exemplo, script) para garantir função apropriada.
6. Realize testes de "integração".
   - Teste a semântica da interface com base nos casos de uso.
   - Conduza testes de navegação.
7. Realize testes de configuração.
   - Avalie a compatibilidade da configuração do lado do cliente.
   - Avalie configurações do lado do servidor.
8. Conduza testes de desempenho.
9. Conduza testes de segurança.

4 Navegadores são conhecidos por implementar suas próprias interpretações "padrão" sutilmente diferentes de HTML e Javascript.

20.3 TESTE DE CONTEÚDO

AVISO

Embora revisões técnicas formais não sejam parte de teste, revisão de conteúdo deve ser realizada para garantir que o conteúdo tenha qualidade.

PONTO CHAVE

Os objetivos de teste de conteúdo são (1) descobrir erros sintáticos no conteúdo; (2) descobrir erros semânticos; (3) encontrar erros estruturais.

? Que questões devem ser formuladas e respondidas para descobrir erros semânticos no conteúdo?

Erros no conteúdo da WebApp podem ser tão triviais como pequenos erros tipográficos ou tão significantes como informação incorreta, organização inadequada ou violação das leis de propriedade intelectual. *Teste de conteúdo* procura descobrir esses e muitos outros problemas antes que o usuário os encontre.

Teste de conteúdo combina tanto revisões quanto a geração de casos de teste executáveis. Revisão é aplicada para descobrir erros semânticos no conteúdo (discutidos na Seção 20.3.1). Teste executável é usado para descobrir os erros de conteúdo que podem ser ligados a conteúdo dinamicamente derivado em função de dados adquiridos de um ou mais bancos de dados.

20.3.1 Objetivos do Teste de Conteúdo

O teste de conteúdo tem três importantes objetivos: (1) descobrir erros sintáticos (por exemplo, erros tipográficos, gramaticais) em documentos com base em texto, representações gráficas e outra mídia, (2) descobrir erros semânticos (erros de precisão ou completeza da informação) em qualquer objeto de conteúdo apresentado à medida que a navegação ocorre, e (3) encontrar erros na organização ou estrutura do conteúdo que é apresentado ao usuário final.

Para atingir o primeiro objetivo, verificadores de ortografia e gramática automatizados podem ser usados. No entanto, muitos erros sintáticos escapam da detecção por tais ferramentas e devem ser descobertos por um revisor humano (testador). Como mencionamos na seção anterior, a edição do texto é a abordagem individual mais adequada para encontrar erros sintáticos.

Teste semântico enfoca a informação apresentada em cada objeto de conteúdo. O revisor (testador) deve responder às seguintes questões:

- A informação é precisa quanto aos fatos?
- A informação é concisa e conclusiva?
- O leiaute do objeto de conteúdo é fácil de o usuário entender?
- A informação embutida em um objeto de conteúdo pode ser encontrada facilmente?
- Referências adequadas são fornecidas para toda informação derivada de outras fontes?
- A informação apresentada é consistente internamente e consistente com a informação apresentada em outros objetos de conteúdo?
- O conteúdo é ofensivo, enganoso ou abre caminho para litígio?
- O conteúdo infringe direitos autorais ou marcas registradas existentes?
- O conteúdo contém ligações internas que complementam o conteúdo existente? As ligações estão corretas?
- O estilo estético do conteúdo conflita com o estilo estético da interface?

Obter respostas para cada uma dessas questões para uma grande WebApp (que contém centenas de objetos de conteúdo) pode ser uma tarefa assustadora. No entanto, falhas em descobrir erros semânticos vão abalar a credibilidade na WebApp e podem levar à falha da aplicação baseada na Web.

Objetos de conteúdo existem dentro de uma arquitetura que tem um estilo específico (Capítulo 19). Durante o teste de conteúdo, a estrutura e a organização da arquitetura de conteúdo são testadas para garantir que o conteúdo necessário seja apresentado ao usuário final na ordem e nos relacionamentos adequados. Por exemplo, a WebApp[5] do *CasaSeguraGarantida.com* apresenta uma variedade de informação sobre sensores usados como parte dos produtos de segurança e vigilância. Objetos de conteúdo fornecem informação descritiva, especificações técnicas, representação foto-

5 A WebApp do *CasaSeguraGarantida.com* tem sido usada como um exemplo ao longo da Parte 3 deste livro.

Document shared on www.docsity.com

gráfica e informação relacionada. Os testes da arquitetura de conteúdo do *CasaSeguraGarantida.com* tentam descobrir erros na apresentação dessa informação (por exemplo, uma descrição do Sensor *X* é apresentada com uma foto do Sensor *Y*).

20.3.2 Teste de Banco de Dados

Aplicações Web modernas fazem muito mais do que apresentar objetos de conteúdo estáticos. Em muitos domínios de aplicação, WebApps têm interface com sofisticados sistemas de gestão de bancos de dados e constroem objetos de conteúdo dinamicamente, que são criados em tempo real, por meio de dados adquiridos de um banco de dados.

Por exemplo, uma WebApp de serviços financeiros pode produzir informação complexa baseada em texto, tabular e gráfica sobre um fundo específico (por exemplo, fundo de ações ou mútuo). O objeto de conteúdo composto que apresenta essa informação é criado dinamicamente depois que o usuário faz uma solicitação de informação sobre um fundo específico. Para tanto, os seguintes passos são necessários: (1) um grande banco de dados de fundos é consultado, (2) dados relevantes são extraídos do banco de dados, (3) os dados extraídos devem ser organizados como um objeto de conteúdo, e (4) esse objeto de conteúdo (representando uma informação personalizada solicitada por um usuário final) é transmitido para o ambiente do cliente para exibição. Erros podem ocorrer e efetivamente ocorrem como conseqüência de cada um desses passos. O objetivo do teste de banco de dados é descobrir esses erros.

Teste de banco de dados de WebApps é complicado por vários fatores:

1. *A solicitação de informação original do lado do cliente é raramente representada no formulário (por exemplo, linguagem de consulta estruturada, SQL) que pode ser inserido em um sistema de gestão de banco de dados (database management system — DBMS).* Assim, testes devem ser projetados para descobrir erros feitos na tradução da solicitação do usuário para um formulário que pode ser processado por esse DBMS.
2. *O banco de dados pode ser remoto para o servidor que hospeda a WebApp.* Assim, testes que descobrem erros na comunicação entre a WebApp e o banco de dados remoto devem ser desenvolvidos.[6]
3. *Dados brutos adquiridos do banco de dados devem ser transmitidos para o servidor da WebApp e adequadamente formatados para transmissão subseqüente ao cliente.* Assim, testes que demonstram a validade dos dados brutos recebidos pelo servidor da WebApp devem ser desenvolvidos, e testes adicionais que demonstram a validade das transformações aplicadas aos dados brutos para criar objetos de conteúdo válidos devem também ser criados.
4. *Os objetos de conteúdo dinâmico devem ser transmitidos para o cliente em um formulário que possa ser exibido ao usuário final.* Assim, uma série de testes deve ser projetada para (a) descobrir erros no formato do objeto de conteúdo, e (b) testar a compatibilidade com diferentes configurações do ambiente do cliente.

? Que tópicos complicam o teste de banco de dados de WebApps?

Considerando esses quatro fatores, os métodos de projeto de casos de teste devem ser aplicados para cada uma das "camadas de interação" [NGU01] observadas na Figura 20.2. O teste deve garantir que (1) a informação válida seja passada entre o cliente e o servidor vinda da camada de interface; (2) a WebApp processe corretamente e extraia adequadamente scripts ou formate os dados do usuário; (3) dados do usuário são passados corretamente para uma função de transformação de dados do lado do servidor, que formata adequadamente as consultas (por exemplo, SQL); (4) consultas são passadas para uma camada[7] de gestão de dados que se comunica com rotinas de acesso a banco de dados (potencialmente localizadas em outras máquinas).

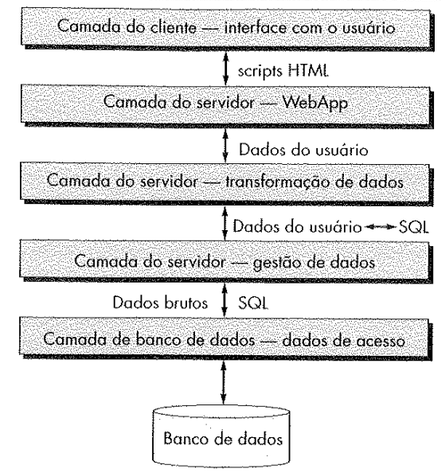
Transformação de dados, gestão de dados e camadas de acesso a banco de dados mostradas na Figura 20.2 são freqüentemente construídas com componentes reusáveis que foram validados

6 Esses testes podem se tornar complexos quando bancos de dados distribuídos são encontrados ou quando o acesso a um armazém de dados (Capítulo 10) é solicitado.

7 A camada de gestão de dados incorpora uma interface de nível de chamada de SQL (SQL-CLI), tal como Microsoft OLE/ADO ou Java Database Connectivity (JDBC).

FIGURA 20.2

Camadas de interação



separadamente e como um pacote. Se esse é o caso, teste de WebApp enfoca o projeto de casos de teste para exercitar as interações entre a camada do cliente e as duas primeiras camadas do servidor (WebApp e transformação de dados) mostradas na figura.

A camada de interface com o usuário é testada para garantir que scripts HTML sejam adequadamente construídos para cada consulta do usuário e adequadamente transmitidos para o lado do servidor. A camada da WebApp do lado do servidor é testada para garantir que dados do usuários sejam adequadamente extraídos de scripts HTML e adequadamente transmitidos para a camada de transformação de dados do lado do servidor.

As funções de transformação de dados são testadas para garantir que SQL correta seja criada e passada para os componentes de gestão de dados adequados.

Uma discussão detalhada da tecnologia subjacente que deve ser entendida para projetar adequadamente esses testes de banco de dados está fora do escopo deste livro. O leitor interessado deve consultar [SCE02], [NGU01] e [BRO01].

> "Como e-clientes (seja de negócio ou consumidor), é pouco provável que tenhamos confiança em um site Web que sofra de freqüentes interrupções de funcionamento, caia no meio de uma transação ou tenha baixo senso de usabilidade. Assim, o teste tem um papel crucial no processo global de desenvolvimento."
>
> **Wing Lem**

20.4 Teste de Interface com o Usuário

Verificação e validação de uma interface com o usuário de uma WebApp ocorre em três pontos distintos do processo de engenharia da Web. Durante a formulação e análise de requisitos (Capítulos 17 e 18), o modelo de interface é revisado para garantir que esteja de acordo com os requisitos do cliente e com outros elementos do modelo de análise. Durante o projeto (Capítulo 19), o modelo de projeto de interface é revisado para garantir que critérios genéricos de qualidade estabelecidos para todas as interfaces com o usuário foram satisfeitos e que tópicos de projeto de interface específicos da aplicações foram adequadamente tratados. Durante o teste, o foco desloca-se para a execução de tópicos específicos da aplicação relativos à interação com o usuário à medida que eles são manifestados pela sintaxe e semântica da interface. Além disso, o teste fornece uma avaliação final de usabilidade.

Document shared on www.docsity.com

20.4.1 Estratégia de Teste de Interface

A estratégia global de teste de interface é (1) descobrir erros relacionados a mecanismos de interface específicos (por exemplo, erros na execução adequada de uma ligação de menu ou no modo pelo qual os dados são introduzidos em um formulário), e (2) descobrir erros no modo pelo qual a interface implementa a semântica de navegação, funcionalidade ou conteúdo exibido na WebApp. Para alcançar essa estratégia um certo número de objetivos devem ser atingidos.

AVISO

Com exceção de especificidades orientadas à WebApp, a estratégia de interface mencionada aqui é aplicável a todos os tipos de software cliente/servidor.

- *Características de interface são testadas para garantir que regras de projeto, estética e conteúdo visual relacionado estejam disponíveis para o usuário, sem erro.* As características incluem fontes de tipo, o uso de cor, molduras, imagens, bordas, tabelas e elementos relacionados que são gerados à medida que a execução da WebApp prossegue.
- *Mecanismos individuais de interface são testados de modo análogo ao teste de unidade.* Por exemplo, testes são projetados para exercitar todos os formulários, scripts do lado do cliente, HTML dinâmico, scritps CGI, conteúdo concatenado e mecanismos específicos da interface da aplicação (por exemplo, um carrinho de compras para uma aplicação de e-comércio). Em muitos casos, o teste pode enfocar exclusivamente um desses mecanismos (a "unidade") em detrimento das outras características e funções da interface.
- *Cada mecanismo de interface é testado no contexto de um caso de uso ou uma NSU (Capítulo19) para uma categoria específica de usuário.* Essa abordagem de teste é análoga ao teste de integração (Capítulo 13) no sentido de que testes são conduzidos à medida que os mecanismos de interface são integrados para permitir que um caso de uso ou uma NSU sejam executados.
- *A interface completa é testada com base em casos de uso e NSU selecionados para descobrir erros na semântica da interface.* Essa abordagem de teste é análoga ao teste de validação (Capítulo 13) porque o objetivo é demonstrar conformidade com a semântica de um caso de uso ou uma NSU específicos. É nesse estágio que uma série de testes de usabilidade é conduzida.
- *A interface é testada dentro de uma variedade de ambientes (por exemplo, navegadores) para garantir que eles sejam compatíveis.* Na realidade, essa série de testes pode também ser considerada parte do teste de configuração.

20.4.2 Mecanismos de Teste de Interface

Quando um usuário interage com uma WebApp, a interação ocorre por meio de um ou mais mecanismos de interface. Nos parágrafos que se seguem, apresentamos um breve panorama de considerações de teste para cada mecanismo de interface [SPL01].

AVISO

Teste de links externos deve ocorrer ao longo da vida da WebApp. Como parte de uma estratégia de manutenção, testes de links devem ser regularmente programados.

Links. Cada link de navegação é testado para garantir que o objeto de conteúdo ou a função apropriados sejam alcançados.[8] O engenheiro da Web constrói uma lista de todos os links associados com o leiaute da interface (por exemplo, barras de menu, itens indexados) e então executa cada um individualmente. Além disso, links dentro de cada objeto de conteúdo precisam ser exercitados para descobrir URLs ruins ou links para objetos de conteúdo ou funções inadequados. Finalmente, links para WebApps externas devem ser testados quanto à precisão e também avaliados para determinar o risco de ficarem inválidos ao longo do tempo.

Formulários. Em um nível macroscópico, testes são realizados para garantir que (1) os rótulos identifiquem corretamente campos do formulário e que campos obrigatórios sejam identificados visualmente para o usuário; (2) o servidor receba toda informação contida no formulário e que nenhum dado seja perdido na transmissão entre o cliente e o servidor; (3) *defaults* apropriados sejam usados quando o usuário não seleciona de um menu *pull-down* ou conjunto de botões; (4) funções do navegador (por exemplo, a seta de retorno) não corrompam dados introduzidos em um formulário; e (5) scripts que realizam verificação de erros nos dados introduzidos funcionem adequadamente e forneçam mensagem de erros significativas.

8 Os testes podem ser realizados tanto como parte do teste de interface quanto do teste de navegação.

Em um nível mais focado, testes devem garantir que (1) campos do formulário tenham largura e tipos de dados adequados; (2) o formulário estabeleça proteções adequadas que impeçam o usuário de introduzir cadeias de texto maiores do que algum máximo predefinido; (3) todas as opções adequadas para menus *pull-down* sejam especificadas e ordenadas de um modo significativo para o usuário final; (4) características de "preenchimento automático" de navegadores não levem a erros de entrada de dados; e (5) a tecla tab (ou alguma outra tecla) inicie o movimento adequado entre os campos do formulário.

AVISO

Testes de script do lado do cliente e testes associados com a HTML dinâmica devem ser repetidos sempre que uma nova versão de um navegador popular é lançada.

Script do lado do cliente. Testes caixa-preta são conduzidos para revelar qualquer erro no processamento à medida que o script (por exemplo, Javascript) é executado. Eles são freqüentemente acoplados a teste de formulário, porque a entrada de script é freqüentemente derivada de dados fornecidos como parte do processamento de formulários. Um teste de compatibilidade deve ser conduzido para garantir que a linguagem de script escolhida vai trabalhar adequadamente na configuração ambiental que apóia a WebApp. Além disso, para testar o próprio script, Splaine e Jaskiel [SPL01] sugerem que "você deve garantir que as normas da sua empresa relativas a [WebApp] declarem a linguagem preferida e a versão da linguagem de script a ser usada para script do lado do cliente (e do lado do servidor)".

HTML dinâmico. Cada página da Web que contém HTML dinâmico é executada para garantir que a exibição dinâmica esteja correta. Além disso, um teste de compatibilidade deve ser conduzido para garantir que o HTML dinâmico funcione adequadamente na configuração(ões) ambiental(is) que apóia(m) a WebApp.

Janelas pop-up.[9] Uma série de testes garante que (1) a pop-up esteja adequadamente dimensionada e posicionada; (2) a pop-up não cubra a janela original da WebApp; (3) o projeto estético da pop-up seja consistente com o projeto estético da interface; e (4) barras de rolagem e outros mecanismos de controle que são parte da pop-up trabalhem, sejam localizados adequadamente e funcionem como o desejado.

Scripts CGI. Testes caixa-preta são conduzidos com ênfase na integridade dos dados (à medida que os dados são passados para o script CGI) e no processamento do script, uma vez validados os dados recebidos. Além disso, teste de desempenho pode ser conduzido para garantir que a configuração do lado do servidor possa acomodar as demandas de processamento de múltiplas chamadas de scripts CGI [SPL01].

Conteúdo encadeado. Testes devem demonstrar que dados encadeados estejam atualizados, adequadamente exibidos e possam ser suspensos sem erro e restaurados sem dificuldade.

Cookies. Testes tanto do lado do cliente quando do lado do servidor são necessários. Do lado do servidor, os testes devem garantir que um *cookie* esteja construído adequadamente (contenha dados corretos) e transmitido adequadamente para o lado do cliente quando um conteúdo ou funcionalidade específica seja solicitado. Além disso, a persistência adequada do *cookie* é testada para garantir que sua data de validade esteja correta. Do lado do cliente, testes determinam se a WebApp acopla adequadamente *cookies* existentes a uma solicitação específica (enviada ao servidor).

Mecanismos de interface específicos da aplicação. Testes obedecem a uma checklist de funcionalidade e características definidas pelos mecanismos da interface. Por exemplo, Splaine e Jaskiel [SPL01] sugerem a seguinte checklist para a funcionalidade do carrinho de compras definida para uma aplicação de e-commerce:

- Testar as fronteiras (Capítulo 14) do número de itens mínimo e máximo que podem ser colocados no carrinho de compras.
- Testar uma solicitação de "saída" de um carrinho de compras vazio.
- Testar a remoção adequada de um item do carrinho de compras.
- Testar para determinar se uma compra esvazia o carrinho do seu conteúdo.
- Testar para determinar a persistência do conteúdo do carrinho de compras (isso deve ser especificado como parte dos requisitos do cliente).

9 Janelas pop-ups tornaram-se difundidas e são fortemente irritantes para muitos usuários. Elas devem ser usadas com parcimônia ou totalmente evitadas.

Document shared on www.docsity.com

- Testar para determinar se a WebApp pode restaurar o conteúdo do carrinho de compras em alguma data futura (considerando que nenhuma compra tenha sido feita) se o usuário solicita que o conteúdo seja salvo.

20.4.3 Teste de Semântica de Interface

Uma vez que cada mecanismo de interface tenha sido testado em termos de "unidade", o enfoque do teste de interface muda para uma consideração da semântica da interface. Teste de semântica de interface "avalia se o projeto cuida adequadamente dos usuários, oferece diretrizes claras, apresenta realimentação e mantém consistência de linguagem e abordagem" [NGU01].

Uma rigorosa revisão do modelo de projeto de interface pode fornecer respostas parciais às questões implícitas no parágrafo precedente. No entanto, cada cenário de caso de uso (para cada categoria de usuário) deve ser testado quando a WebApp estiver implementada. Em essência, um caso de uso torna-se a entrada para o projeto de uma seqüência de teste. O objetivo da seqüência de teste é descobrir erros que impeçam um usuário de atingir o objetivo associado ao caso de uso.

À medida que cada caso de uso é testado, a equipe de engenharia da Web mantém uma checklist para garantir que cada item de menu seja exercitado pelo menos uma vez e que cada ligação embutida em um objeto de conteúdo tenha sido usada. Além disso, a seqüência de teste deve incluir uso inadequado de seleção de menu e ligação. O objetivo é determinar se a WebApp fornece efetiva manipulação e recuperação de erro.

20.4.4 Testes de Usabilidade

Veja na Web

Uma diretriz que vale a pena para o teste de usabilidade pode ser encontrada em www. www.ahref.com/guides/design/199806/0615jef.html.

Teste de usabilidade é similar ao teste de semântica de interface (Seção 20.4.3), porque também avalia o grau em que os usuários podem interagir efetivamente com a WebApp e o grau em que a WebApp dirige as ações dos usuários, fornece realimentação significativa e impõe uma abordagem de interação consistente. Em vez de enfocar propositadamente a semântica de algum objetivo interativo, as revisões e testes de usabilidade são projetados para determinar o grau em que a interface da WebApp torna a vida do usuário mais fácil.[10]

Testes de usabilidade podem ser projetados por uma equipe da engenharia da Web, mas os testes em si são conduzidos por usuários finais. A seguinte seqüência de passos é aplicada [SPL01]:

1. Defina um conjunto de categorias de teste de usabilidade e identifique as metas de cada uma.
2. Projete testes que permitam que cada meta seja avaliada.
3. Selecione participantes que vão conduzir os testes.
4. Instrumente a interação dos participantes com a WebApp enquanto o teste é conduzido.
5. Desenvolva um mecanismo para avaliar a usabilidade da Webapp.

Teste de usabilidade pode ocorrer em diversos níveis de abstração: (1) a usabilidade de um mecanismo de interface específico (por exemplo, um formulário) pode ser avaliado; (2) a usabilidade de uma página completa da Web (abrangendo mecanismos de interface, objetos de dados e funções relacionadas) pode ser avaliada; ou (3) a usabilidade da WebApp completa pode ser considerada.

O primeiro passo no teste é identificar um conjunto de categorias de usabilidade e estabelecer os objetivos de teste para cada uma delas. As seguintes categorias e objetivos de teste (escritos na forma de questão) ilustram essa abordagem:[11]

? Que características de usabilidade tornam-se o foco do teste e que objetivos específicos são visados?

Interatividade — Mecanismos de interação (por exemplo, menus *pull-down*, botões e ponteiros) são fáceis de entender e usar?

Leiaute — Mecanismos de navegação, conteúdo e funções estão colocados de modo que permitam ao usuário encontrá-los rapidamente?

Legibilidade — O texto é bem escrito e inteligível?[12] As representações gráficas são fáceis de entender?

10 O termo "amigável ao usuário" tem sido usado nesse contexto. O problema certamente é que a percepção de um usuário de interface "amigável" pode ser radicalmente diferente da de outro.

11 Para questões adicionais de usabilidade, veja o quadro sobre "usabilidade" no Capítulo 12.

12 O FOG Readability Index e outros podem ser usados para fornecer uma avaliação quantitativa da legibilidade. Visite o endereço *http://developer.gnome.org/documents/usability/usability-readability.html* para mais detalhes.

Estética — Leiaute, cor, tamanho e estilo de caracteres e características relacionadas levam a facilidade de uso? Os usuários "sentem-se confortáveis" com o aspecto da WebApp?

Características de exibição — A WebApp faz uso otimizado do tamanho e da resolução da tela?

Sensibilidade ao tempo — Características, funções e conteúdo importantes podem ser usados ou adquiridos a tempo?

Personalização — A WebApp se personaliza para necessidades específicas de diferentes categorias de usuário ou de usuários individuais?

Acessibilidade — A WebApp é acessível a portadores de deficiência?

Em cada uma dessas categorias, uma série de testes é projetada. Em alguns casos, o "teste" pode ser uma revisão visual de uma página da Web. Em outros casos, testes de semântica de interface podem ser executados novamente, mas, nessa instância, preocupações com usabilidade são essenciais.

Como exemplo, consideramos a avaliação da usabilidade de mecanismos de interação e interface. Constantine and Lockwood [CON03] sugerem que a seguinte lista de características de interface deve ser revisada e testada quanto à usabilidade: animação, botões, cor, controle, diálogo, campos, formulários, molduras, gráficos, rótulos, ligações, menus, mensagens, navegação, páginas, seletores, texto e barras de ferramenta. À medida que cada característica é avaliada, ela é classificada em uma escala qualitativa pelos usuários que estão fazendo o teste. A Figura 20,3 mostra um possível conjunto de "graus" de avaliação que podem ser selecionados pelos usuários. Esses graus são aplicados a cada característica individualmente, a uma página completa da Web ou à WebApp como um todo.

20.4.5 Testes de Compatibilidade

PONTO CHAVE

WebApps executam em uma variedade de ambientes do lado do cliente. O objetivo dos testes de compatibilidade é descobrir erros associados com um ambiente específico (por exemplo, navegador).

WebApps precisam operar em ambientes que diferem uns dos outros. Diferentes computadores, dispositivos de exibição, sistemas operacionais, navegadores e velocidade de conexão com a rede têm uma influência significativa na operação da WebApp. Cada configuração de computação pode resultar em diferenças nas velocidades de processamento, resolução de exibição e velocidades de conexão no lado do cliente. Excentricidades de sistemas operacionais podem causar problemas de processamento de WebApp. Diferentes navegadores, algumas vezes, produzem resultados ligeiramente diferentes, independentemente do grau de normalização de HTML na WebApp. Conectáveis (*plugins*) exigidos podem ou não estar prontamente disponíveis para determinada configuração.

Em alguns casos, pequenos tópicos de compatibilidade não apresentam problemas significativos, mas, em outros, sérios erros podem ser encontrados. Por exemplo, velocidades baixas de *download* podem tornar-se inaceitáveis, falta de um *plugin* exigido pode indisponibilizar conteúdo, diferenças de navegador podem modificar leiaute de página dramaticamente, estilos de fonte podem ser alterados e tornar-se ilegíveis ou formulários podem ser inadequadamente organizados. *Teste de compatibilidade* busca descobrir esses problemas antes que a WebApp fique on-line.

FIGURA 20.3

Avaliação qualitativa da usabilidade

Document shared on www.docsity.com

O primeiro passo no teste de compatibilidade é definir um conjunto de configurações de computação do lado do cliente e suas variantes "comumente encontradas". Em essência, uma estrutura de árvore é criada, identificando cada plataforma de computação, dispositivos de exibição típicos, sistemas operacionais apoiados na plataforma, navegadores disponíveis, provável velocidade de conexão com a Internet e informação similar. Em seguida, a equipe de engenharia da Web deriva uma série de testes de validação de compatibilidade, derivados dos testes de interface, testes de navegação, testes de desempenho e testes de segurança existentes. O objetivo desses testes é descobrir erros ou problemas de execução que podem ser atribuídos a diferenças de configuração.

CASASEGURA

Teste de WebApp

A cena: Escritório de Doug Miller.

Os personagens: Doug Miller (gerente do grupo de engenharia de software do *CasaSegura*) e Vinod Raman, membro da equipe de engenharia de software do produto.

A conversa:

Doug: O que você acha da WebApp de e-commerce *CasaSeguraGarantida.com* V.0.0?

Vinod: O fornecedor terceirizado fez um bom trabalho. Sharon [gerente de desenvolvimento do fornecedor] contou-me que eles estão testando enquanto estamos aqui conversando.

Doug: Eu gostaria que você e o resto da equipe fizessem um pouco de teste informal no site de e-commerce.

Vinod (fazendo careta): Pensei que fôssemos contratar uma empresa de teste para validar a WebApp. Nós ainda estamos tentando terminar o software.

Doug: Nós vamos contratar um fornecedor de teste para testar desempenho e segurança, e nosso fornecedor terceirizado já está testando. Eu apenas pensei que outro ponto de vista ajudaria e, além disso, gostaríamos de manter os custos sob controle, assim...

Vinod (suspira): O que você está procurando?

Doug: Quero certificar-me de que a interface e toda a navegação estão sólidas.

Vinod: Acho que podemos começar com os casos de uso de cada uma das principais funções de interface.

Aprenda sobre *CasaSegura*

Especifique o sistema *CasaSegura* de que você precisa

Compre o sistema *CasaSegura*

Obtenha apoio técnico

Doug: Bom. Mas avalie os caminhos de navegação em toda a sua extensão até a conclusão.

Vinod (folheando um caderno de casos de usos): É, quando você seleciona **Especifique o sistema *CasaSegura* de que você precisa**, isso vai levá-lo a:

Selecione componentes do *CasaSegura* e Obtenha recomendações de componentes do *CasaSegura*.

Podemos exercitar a semântica de cada caminho.

Doug: Enquanto você está aí, verifique o conteúdo que aparece em cada nó de navegação.

Vinod: Lógico... e os elementos funcionais também. Quem está testando a usabilidade?

Doug: Humm... o fornecedor de teste vai coordenar o teste de usabilidade. Contratamos uma firma de pesquisa de marketing para levantar 20 usuários típicos para o estudo de usabilidade, mas se vocês descobrirem algum tópico de usabilidade...

Vinod: Já sei, passamos para eles.

Doug: Obrigado, Vinod.

20.5 TESTE NO NÍVEL DE COMPONENTE

Teste no nível de componente, também chamado de *teste de função*, tenta descobrir erros em funções de WebApp. Cada função de WebApp é um módulo de software (implementada em uma variedade de linguagens de programação ou scripts) e pode ser testada por meio de técnicas caixa-preta (e, em alguns casos, caixa-branca) discutidas no Capítulo 14.

Casos de teste no nível de componente são freqüentemente orientados por entradas no nível de formulários. Uma vez definidos os dados dos formulários, o usuário seleciona um botão ou outro mecanismo de controle para iniciar a execução. Os seguintes métodos de projeto de casos de teste (Capítulo14) são típicos:

- *Partição de equivalência* — O domínio de entrada da função é dividido em categorias ou classes de entrada das quais os casos de teste são derivados.
- *O formulário de entrada* é avaliado para determinar quais classes de dados são relevantes para a função. Casos de teste para cada classe de entrada são derivados e executados enquanto outras classes de entrada são mantidas constantes. Por exemplo, uma aplicação de e-comércio pode implementar uma função que calcula taxas de remessa. Entre uma variedade de informações de remessa fornecida via formulário, está o código postal do usuário. Casos de teste são projetados em uma tentativa de descobrir erros no processamento do código postal pela especificação de valores de código postal que podem descobrir diferentes classes de erro (por exemplo, um código postal incompleto, um código postal correto ou inexistente, um formato de código postal errado).
- *Análise de valor-limite* — Dados dos formulários são testados nos seus limites. Por exemplo, a função de cálculo de remessa mencionada anteriormente exige um número de máximo de dias para entrega do produto. Um mínimo de dois e um máximo de 14 dias são mencionados no formulário. No entanto, o teste de valor-limite poderia entrar com valores 0, 1, 2, 13, 14 e 15 para determinar como a função reage a dados dentro e fora dos limites de entrada válida.[13]
- *Teste de caminho* — Se a complexidade lógica da função é alta,[14] teste de caminho (um método de projeto de casos de teste caixa-branca) pode ser usado para garantir que cada caminho independente no programa foi exercitado.

Além desses métodos de projeto de casos de teste, uma técnica chamada de *teste de erro forçado* [NGU01] é usada para derivar casos de teste que premeditadamente guiam o componente da WebApp para uma condição de erro. O objetivo é descobrir erros que ocorrem durante a manipulação de erros (por exemplo, mensagens de erro incorretas ou inexistentes, falha da WebApp como conseqüência do erro, saída errônea provocada por entrada errada, efeitos colaterais relacionados ao processamento do componente).

Cada caso de teste no nível de componente especifica todos os valores de entrada e a saída esperada a ser fornecida pelo componente. A saída real produzida como conseqüência do teste é registrada para referência futura durante apoio e manutenção.

Em muitas situações, a execução correta de uma função de WebApp está ligada a uma interface adequada com um banco de dados que pode ser externo à WebApp. Assim, teste de banco de dados torna-se uma parte integral do regime de teste de componente. Hower [HOW97] discute isso quando escreve:

> Sites Web orientados a banco de dados podem envolver uma interação complexa entre navegadores Web, sistemas operacionais, aplicações plugáveis, protocolos de comunicação, servidores Web, bancos de dados, programas [em linguagens de script]..., aperfeiçoamentos de segurança e bloqueadores contra ataques (firewalls). Essa complexidade torna impossível testar cada dependência e tudo o que pode estar errado em um site. O projeto de desenvolvimento de um site típico da Web também estará em um cronograma agressivo, assim, a melhor abordagem de teste vai empregar análise de risco para determinar onde concentrar os esforços de teste. Análise de risco deve incluir consideração de quanto o ambiente de teste vai se aproximar do ambiente real de produção... Outras considerações típicas de análise de risco incluem:
>
> - Que funcionalidade do site é mais crítica para esse objetivo?
> - Que áreas do site necessitam de interação mais pesada com o banco de dados?
> - Que pontos do CGI, *applets*, componentes Active X e assim por diante do site são mais complexos?

Veja na Web

Um variedade de recursos de teste pode ser encontrada em www.pantos.org/atw/xref.html.

13 Nesse caso, um melhor projeto de entrada poderia eliminar erros potenciais. O número máximo de dias poderia ser selecionado de um menu *pull-down*, o que evitaria que o usuário especificasse entrada fora dos limites.

14 A complexidade lógica pode ser determinada pelo cálculo da complexidade ciclomática do algoritmo. Veja o Capítulo 14 para detalhes adicionais.

Document shared on www.docsity.com

- Que tipos de problemas poderiam causar mais reclamações ou pior publicidade?
- Que áreas do site serão as mais populares?
- Que pontos do site têm os mais altos riscos de segurançca?

Cada um dos tópicos relacionados a risco discutidos por Hower deve ser considerado quando se projeta casos de teste para componentes de WebApp e funções de banco de dados relacionadas.

20.6 Teste de Navegação

Um usuário navega por uma WebApp de modo muito semelhante ao que um visitante caminha por uma loja ou museu. Há muitos caminhos que podem ser trilhados, muitas paradas que podem ser feitas, muitas coisas para aprender e observar, atividades a iniciar e decisões a tomar. Como discutimos anteriormente, esse processo de navegação é previsível no sentido de que cada visitante tem um conjunto de objetivos quando chega. Ao mesmo tempo, o processo de navegação pode ser imprevisível porque o visitante, influenciado por algo que vê ou aprende, pode escolher um caminho ou iniciar uma ação que não é típica do objetivo original. A tarefa do teste de navegação é (1) garantir que os mecanismos que permitem ao usuário navegar pela WebApp estejam todos em funcionamento e (2) certificar-se de que cada unidade semântica de navegação (*navigation semantic unit* — NSU) possa ser alcançada pela categoria de usuário adequada.

> "Não estamos perdidos. Estamos desafiados pela localização."
>
> **John M. Ford**

20.6.1 Sintaxe do Teste de Navegação

A primeira fase do teste de navegação começa realmente durante o teste de interface. Mecanismos de navegação são testados para garantir que cada um execute sua pretensa função. Splaine e Jaskiel [SPL01] sugerem que cada um dos seguintes mecanismos de navegação deva ser testado:

- *Links de navegação* — links internos à WebApp, links externos para outras WebApps e âncoras dentro de uma página da Web específica devem ser testados para garantir que conteúdo ou funcionalidade adequados sejam alcançados quando o link é escolhido.
- *Redirecionamentos* — esses links entram em ação quando um usuário solicita uma URL inexistente ou seleciona um link cujo destino tenha sido removido ou cujo nome tenha mudado. Uma mensagem é exibida para o usuário, e a navegação é redirecionada para outra página (por exemplo, a página principal). Redirecionamentos devem ser testados pela solicitação de links incorretos internos ou URLs externas e avaliação de como a WebApp trata essas solicitações.
- *Marcadores de páginas* — embora marcadores de páginas sejam uma função do navegador, a WebApp deve ser testada para garantir que um título de página significativo possa ser extraído quando o marcador é criado.
- *Molduras e conjuntos de molduras* — cada moldura contém o conteúdo de uma página da Web específica; um conjunto de molduras contém várias molduras e permite a exibição de várias páginas da Web ao mesmo tempo. Como é possível aninhar molduras e conjuntos de molduras uns com os outros, esses mecanismos de navegação e exibição devem ser testados quanto a conteúdo correto, leiaute e tamanho adequado, desempenho de baixa e compatibilidade com o navegador.
- *Mapas de sites* — entradas devem ser testadas para garantir que o link leve o usuário ao conteúdo ou à funcionalidade adequados.

- *Motores de busca internos* — WebApps complexas freqüentemente contêm centenas ou mesmo milhares de objetos de conteúdo. Um motor de busca interno permite ao usuário realizar uma busca por palavra-chave na WebApp para encontrar o conteúdo desejado. Teste de motor de busca valida a precisão e completeza da busca, as propriedades de tratamento de erro do motor de busca e características avançadas de busca (por exemplo, o uso de operadores booleanos no campo de busca).

Alguns dos testes mencionados podem ser realizados com ferramentas automatizadas (por exemplo, verificadores de link), enquanto outros são projetados e executados manualmente. O objetivo global é garantir que erros em mecanismos de navegação sejam encontrados antes que a WebApp esteja on-line.

20.6.2 Teste de Semântica de Navegação

No Capítulo 19 definimos uma unidade de semântica de navegação (*navigation semantic unit* — NSU) como "um conjunto de informação e estruturas de navegação relacionados que colaboram na satisfação de um subconjunto de requisitos de usuário relacionados" [CAC02]. Cada NSU é definida por um conjunto de caminhos de navegação (chamado de "modos de navegação") que conectam nós de navegação (por exemplo, páginas da Web, objetos de conteúdo ou funcionalidade). Tomada como um todo, cada NSU permite ao usuário atingir os requisitos específicos definidos por um ou mais casos de uso para uma categoria de usuário. Teste de navegação exercita cada NSU para garantir que esses requisitos possam ser atingidos. À medida que cada NSU é testada, a equipe de engenharia da Web deve responder às seguintes questões:

? Que questões devem ser formuladas e respondidas quando cada NSU é testada?

- A NSU é alcançada na sua totalidade, sem erro?
- Cada nó de navegação (definido para uma NSU) é atingível no contexto dos caminhos de navegação definidos para a NSU?
- Se a NSU pode ser atingida usando mais do que um caminho de navegação, todos os caminhos relevantes foram testados?
- Se diretriz é fornecida pela interface com o usuário para dar suporte à navegação, as diretrizes estão corretas e inteligíveis à medida que a navegação prossegue?
- Há um mecanismo (diferente da seta "de retorno" do navegador) para retornar ao nó de navegação precedente e ao começo do caminho de navegação?
- Os mecanismos para navegação dentro de um nó grande de navegação (isto é, uma longa página da Web) funcionam adequadamente?
- Se uma função deve ser executada em um nó e o usuário prefere não fornecer entrada, o restante da NSU pode ser completado?
- Há um modo de interromper a navegação antes que todos os nós tenham sido alcançados, mas depois voltar para onde a navegação foi descontinuada e prosseguir daí para a frente?
- Todo nó é alcançável pelo mapa do site? Os nomes dos nós são significativos aos usuários finais?
- Se um nó dentro de uma NSU é atingido por alguma fonte externa, é possível prosseguir até o próximo nó do caminho de navegação? É possível voltar ao nó anterior do caminho de navegação?
- O usuário entende sua localização na arquitetura de conteúdo à medida que a NSU é executada?

AVISO

Se NSUs não tiverem sido criadas como parte da análise ou projeto da engenharia da Web, você pode aplicar casos de uso para o projeto de casos de testes de navegação. O mesmo conjunto de questões é formulado e respondido.

Teste de navegação, como teste de interface e de usabilidade, deve ser conduzido por clientela tão variada quanto possível. Estágios iniciais de teste são conduzidos por engenheiros da Web, mas estágios posteriores devem ser conduzidos por outros interessados no projeto, uma equipe de teste independente e, finalmente, por usuários não técnicos. O objetivo é exercitar a navegação da WebApp profundamente.

Document shared on www.docsity.com

20.7 Teste de Configuração

Variabilidade e instabilidade de configuração são fatores importantes que tornam a engenharia da Web um desafio. Hardware, sistema(s) operacional(is), navegadores, capacidade de armazenamento, velocidades de comunicação em rede e uma variedade de outros fatores do lado do cliente são difíceis de prever para cada usuário. Além disso, a configuração para um determinado usuário pode se modificar (por exemplo, atualizações de SO, novas velocidades de conexão e ISP) regularmente. O resultado pode ser um ambiente do lado do cliente que seja propenso a erros que são tanto sutis quanto significativos. Uma impressão do usuário da WebApp e o modo pelo qual ele interage com ela pode diferir significativamente de outra experiência de usuário, se ambos os usuários não estiverem trabalhando com a mesma configuração do lado do cliente.

A tarefa de *teste de configuração* não é exercitar cada possível configuração do lado do cliente. Em vez disso, é testar um conjunto de prováveis configurações do lado do cliente e do lado do servidor para garantir que a experiência do usuário seja a mesma em todos eles e para isolar erros que podem ser específicos de uma particular configuração.

20.7.1 Tópicos do Lado do Servidor

Do lado do servidor, casos de teste de configuração são projetados para verificar se a configuração projetada do servidor (isto é, servidor de WebApp, servidor de banco de dados, sistema(s) operacional(is), software de bloqueio contra ataques, aplicações concorrentes) pode suportar a WebApp sem erro. Em essência, a WebApp é instalada no ambiente do lado do servidor e testada com a intenção de encontrar erros relacionados à configuração.

À medida que testes de configuração do lado do servidor são projetados, o engenheiro da Web deve considerar cada componente da configuração do servidor. Entre as questões que precisam ser formuladas e respondidas durante o teste de configuração do lado do servidor estão:

? Que questões precisam ser formuladas e respondidas quando teste de configuração do lado do servidor é conduzido?

- A WebApp é plenamente compatível com o SO do servidor?
- Os arquivos do sistema, diretórios e dados relacionados ao sistema estão criados corretamente quando a WebApp fica operacional?
- As medidas de segurança do sistema (por exemplo, bloqueadores contra ataque ou criptografia) permitem que a WebApp seja executada e sirva aos usuários sem a interferência ou degradação de desempenho?
- A WebApp tem sido testada com a configuração[15] distribuída do servidor (se existir uma) que foi escolhida?
- A WebApp está integrada adequadamente com o software de banco de dados? A WebApp é sensível a diferentes versões de software de banco de dados?
- Os scripts de WebApp do lado do servidor são executados adequadamente?
- Os erros do administrador do sistema têm sido examinados quanto aos seus efeitos nas operações da WebApp?
- Se servidores substitutos são usados, as diferenças em suas configurações têm sido tratadas com teste local?

20.7.2 Tópicos do Lado do Cliente

Do lado do cliente, testes de configuração concentram-se mais pesadamente na compatibilidade da WebApp com configurações que contêm uma ou mais permutações dos seguintes componentes [NGU01]:

- *Hardware* — CPU, memória, armazenamento e dispositivos de impressão.
- *Sistemas Operacionais* — Linux, Macintosh OS, Microsoft Windows, um SO móvel.

15 Por exemplo, podem ser usados um servidor de aplicação e um servidor de banco de dados separados. A comunicação entre as duas máquinas ocorre por meio de uma conexão de rede.

- *Software navegador* — Internet Explorer, Mozilla/Netscape, Opera, Safari e outros.
- *Componentes de interface com o usuário* — Active X, Java *applets* e outros.
- *Plug-ins* — QuickTime, RealPlayer e muitos outros.
- *Conectividade* — cabo, DSL, modem regular, T1.

Além desses componentes, outras variáveis incluem software de rede, as excentricidades do ISP e aplicações em execução concorrentemente.

Para projetar testes de configuração do lado do cliente, a equipe de engenharia da Web deve reduzir o número de variáveis de configuração para um número[16] gerenciável. Para conseguir isso, cada categoria de usuário é avaliada para determinar a probabilidade de a configuração ser encontrada dentro da categoria. Além disso, os dados de divisão do mercado da indústria podem ser usados para prever a combinação de componentes mais prováveis. A WebApp é então testada dentro desses ambientes.

20.8 Teste de Segurança

Segurança de WebApp é um assunto complexo que deve ser completamente entendido antes que teste de segurança efetivo possa ser realizado.[17] WebApps e ambientes do lado do cliente e do lado do servidor nos quais elas estão hospedadas representam um alvo atrativo para invasores (*hackers*) externos, empregados insatisfeitos, competidores desonestos e qualquer outro que deseje roubar informação confidencial, modificar conteúdo com má intenção, degradar desempenho, desmontar funcionalidade ou embaraçar uma pessoa, organização ou negócio.

> "A Internet é um lugar arriscado para conduzir negócios ou guardar bens. Invasores, plagiadores (*crackers*), xeretas, enganadores... ladrões, vândalos, lançadores de vírus e fornecedores de programas piratas estão à solta."
>
> **Dorothy e Peter Denning**

AVISO

Se a WebApp é crítica para o negócio, mantém dados confidenciais ou é um alvo provável para invasores, é uma boa idéia terceirizar o teste de segurança para um fornecedor que se especializa nisso.

Testes de segurança são projetados para encontrar vulnerabilidades no ambiente do lado do cliente, nas comunicações de rede que ocorrem quando dados são passados do cliente para o servidor e de volta e no ambiente do lado do servidor. Cada um desses domínios pode ser atacado, e é trabalho do testador de segurança descobrir fraquezas que podem ser exploradas por aqueles que pretendem fazê-lo.

No lado do cliente, vulnerabilidades podem freqüentemente ser rastreadas até erros preexistentes nos navegadores, programas de e-mail ou software de comunicação. Nguyen [NGU01] descreve uma típica brecha de segurança:

> Um dos erros comumente mencionados é Transbordamento de Buffer (*Buffer Overflow*) que permite que código mal-intencionado seja executado na máquina do cliente. Por exemplo, a entrada de uma URL em um navegador muito maior do que o tamanho do buffer alocado à URL vai causar um erro de sobreescrita de memória (*buffer overflow*) se o navegador não tiver código de detecção de erro para validar o tamanho da URL de entrada. Um invasor experiente pode explorar maliciosamente esse erro escrevendo uma longa URL com código para ser executado que pode provocar a quebra do navegador ou alterar as medidas de segurança (de alta para baixa), ou , pior, corromper os dados do usuário.

Outra potencial vulnerabilidade no lado do cliente é acesso não autorizado a *cookies* colocados no navegador. Sites criados com má intenção podem adquirir informação contida em *cookies* legítimos, e usar essa informação para comprometer a privacidade do usuário ou, pior, armar a cena para roubo de identidade.

16 Aplicar testes em cada possível combinação de componentes de configuração consome tempo excessivo.
17 Livros de Trivedi [TRI03], McClure e seus colegas [MCC03] e Garfinkel e Spafford [GAR02] fornecem informação útil sobre o assunto.

Document shared on www.docsity.com

Dados comunicados entre o cliente e o servidor são vulneráveis à *enganação*. Essa ocorre quando um extremo do caminho de comunicação é subvertido por uma entidade mal-intencionada. Por exemplo, um usuário pode ser enganado por um site Web que age como se fosse o legítimo servidor da WebApp (aspecto idêntico). O objetivo é roubar senhas, informação proprietária ou dados de crédito.

No lado do servidor, as vulnerabilidades incluem ataques de negativa de serviço e scripts mal-intencionados que podem ser passados para o lado do cliente ou usados para danificar as operações do servidor. Além disso, bancos de dados do lado do servidor podem dar acesso sem autorização (roubo de dados).

Para proteger contra essas (e muitas outras) vulnerabilidades um ou mais dos seguintes elementos de segurança é implementado [NGU01]:

PONTO CHAVE

Testes de segurança devem ser projetados para exercitar bloqueadores contra ataques, autenticação, criptografia e autorização.

- *Firewalls (Bloqueadores contra ataques)* — mecanismo de filtragem que é uma combinação de hardware e software que examina cada pacote de informação garantindo que esteja vindo de uma fonte legítima, bloqueando qualquer dado que seja suspeito.
- *Autenticação* — mecanismo de verificação que valida a identidade de todos os clientes e servidores, permitindo que a comunicação ocorra somente quando os dois lados tiverem sido verificados.
- *Criptografia* — mecanismo de codificação que proteje dados confidenciais modificando-os de um modo que torna impossível serem lidos por aqueles que têm má intenção. A criptografia é reforçada pelo uso de *certificados digitais* que permitem ao cliente verificar o destinatário para o qual os dados são transmitidos.
- *Autorização* — mecanismo de filtragem que permite acesso ao ambiente do cliente ou do servidor somente aos indivíduos com códigos de autorização adequados (por exemplo, ID de usuário e senha).

O objetivo do teste de segurança é expor brechas nesses elementos de segurança que possam ser exploradas pelos mal-intencionados. O projeto real de testes de segurança exige profundo conhecimento interno de cada elemento de segurança e um entendimento abrangente de tecnologias de rede. Em muitos casos, o teste de segurança é terceirizado para empresas que se especializam nessas tecnologias.

20.9 Teste de Desempenho

Nada é mais frustrante do que uma WebApp que leva muitos minutos para carregar conteúdo quando sites concorrentes baixam conteúdo similar em segundos. Nada é mais desagradável do que tentar entrar em uma WebApp e receber uma mensagem "servidor ocupado", com a sugestão para tentar mais tarde. Nada é mais desconcertante do que uma WebApp que responde instantaneamente em algumas situações e, depois parece passar para um estado de espera infinita em outras situações. Todas essas ocorrências acontecem no dia-a-dia da Web e todas estão relacionadas à desempenho.

Teste de desempenho é usado para descobrir problemas de desempenho que podem resultar da falta de recursos do lado do servidor, largura de banda de rede inadequada, capacidades de banco de dados inadequadas, falha ou fragilidade das capacidades do sistema operacional, funcionalidade de WebApp mal projetada e outros tópicos de hardware ou software que podem levar à degradação do desempenho cliente/servidor. O objetivo é duplo: (1) entender como o sistema responde a *carregamento* (isto é, número de usuários, número de transações ou volume global de dados), e (2) coletar métricas que vão levar a modificações de projeto para melhorar o desempenho.

20.9.1 Objetivos do Teste de Desempenho

Testes de desempenho são projetados para situações simulares de carregamento do mundo real. À medida que o número de usuários simultâneos da WebApp cresce, ou o número de transações

on-line aumenta, ou a quantidade de dados (baixados ou carregados [uploaded]) aumenta, o teste de desempenho vai ajudar a responder às seguintes questões:

AVISO

Alguns tópicos de desempenho de WebApp, pelo menos como são percebidos pelo usuário final, são difíceis de testar, inclusive congestionamento da rede, as excentridades do hardware de interface com a rede e tópicos similares.

- O tempo de resposta do servidor degrada até um ponto em que é notável e inaceitável?
- Em que ponto (em termos de usuários, transações ou carregamento de dados) o desempenho torna-se inaceitável?
- Que componentes do sistema são responsáveis pela degradação de desempenho?
- Qual é o tempo médio de resposta para usuários sob uma variedade de condições de carregamento?
- A degradação de desempenho tem impacto na segurança do sistema?
- A confiabilidade ou precisão da WebApp é afetada à medida que a carga do sistema cresce?
- O que acontece quando cargas maiores do que a capacidade máxima do servidor são aplicadas?

Para desenvolver respostas a essas questões, dois diferentes testes de desempenho são conduzidos:

- *Teste de carga* — o carregamento do mundo real é testado em diversos níveis de carga e em diversas combinações.
- *Teste de esforço* — o carregamento é aumentado até o ponto de quebra para determinar que capacidade o ambiente da WebApp pode tratar.

Cada uma dessas estratégias de teste é considerada nas seções que se seguem.

20.9.2 Teste de Carga

O objetivo do *teste de carga* é determinar como a WebApp e seu ambiente do lado do servidor vão responder a várias condições de carregamento. À medida que o teste prossegue, permutações das variáveis seguintes definem um conjunto de condições de teste:

N, o número de usuários concorrentes.
T, o número de transações on-line por usuários por unidade de tempo.
D, a carga de dados processada pelo servidor por transação.

AVISO

Se uma WebApp usa múltiplos servidores para fornecer capacidade significativa, o teste de carga deve ser realizado em um ambiente multisservidor.

Em cada caso, essas variáveis são definidas dentro dos limites de operação normal do sistema. À medida que cada condição de teste é executada, uma ou mais das seguintes medidas são coletadas: resposta média de usuário, tempo médio para baixar uma unidade de dados normalizada ou tempo médio para processar uma transação. A equipe de engenharia da Web examina essas medidas para determinar se uma súbita diminuição de desempenho pode ser atribuída a uma combinação específica de *N*, *T* e *D*.

Teste de carga pode também ser usado para avaliar velocidades de conexão recomendadas para os usuários da WebApp. A vazão global de dados, *P*, é calculada do seguinte modo:

$$P = N \times T \times D.$$

Como exemplo, considere um site popular de notícias esportivas. Em um dado momento, 20.000 usuários concorrentes submetem uma solicitação (uma transação, *T*), em média, a cada dois minutos. Cada transação requer que a WebApp baixe um novo artigo que tem, em média, 3Kbytes de tamanho. Assim, a vazão pode ser calculada como:

$$P = [20.000 \times 0,5 \times 3 \text{ Kb}]/60 = 500 \text{ Kbytes/seg}$$
$$= 4 \text{ megabits por segundo}$$

A conexão de rede para o servidor deveria, desse modo, ter de suportar essa taxa de dados e ser testada para garantir que o faz.

Document shared on www.docsity.com

PONTO CHAVE

O objetivo do teste de esforço é melhor entender como um sistema falha à medida que é forçado além dos seus limites operacionais.

20.9.3 Teste de Esforço

Teste de esforço (Capítulo 13) é uma continuação do teste de carga, mas, nessa instância, as variáveis *N, T* e *D* são forçadas a atingir e depois exceder os limites operacionais. O objetivo desses testes é responder a cada uma das seguintes questões:

- O sistema degrada-se "suavemente" ou o servidor entra em colapso quando a capacidade é excedida?
- O software do servidor gera mensagens "servidor não disponível"? De forma mais geral, os usuários estão conscientes de que não podem alcançar o servidor?
- O servidor enfileira as solicitações de recursos e esvazia a fila quando as demandas de capacidade diminuem?
- As transações são perdidas quando a capacidade é excedida?
- A integridade de dados é afetada quando a capacidade é excedida?
- Quais valores de *N, T* e *D* forçam o ambiente do servidor a falhar? Como a falha se manifesta? Notificações automáticas são enviadas à equipe de apoio técnico no local do servidor?
- Se o sistema falhar, quanto tempo vai levar para voltar a operar?
- Certas funções da WebApp (por exemplo, funcionalidade intensiva de cálculos, capacidades de encadeamento de dados) são interrompidas quando a capacidade atinge o nível de 80% ou 90%?

Uma variante de teste de esforço é algumas vezes referida como teste de alternância de pico (*spike/bounce testing*) [SPL01]. Nesse regime de teste, a carga é elevada até a capacidade, depois abaixada rapidamente para as condições de operação normal, e depois elevada novamente. Pela alternância da carga do sistema, um testador pode determinar como o servidor pode mobilizar recursos para encarar demanda muito alta e depois liberá-las quando condições normais reaparecem (para que estejam prontas para o próximo pico).

FERRAMENTAS DE SOFTWARE

Taxonomia de Ferramentas de Teste para WebApp

Em seu artigo sobre teste de sistemas de e-comércio, Lam [LAM01] apresenta uma taxonomia útil de ferramentas automatizadas que têm aplicabilidade direta para teste em um contexto de engenharia da Web. Incluímos ferramentas representativas em cada categoria.[18]

Ferramentas de gestão de configuração e conteúdo gerenciam o controle de versão e modificação dos objetos de conteúdo e componentes funcionais de WebApp.

Ferramentas Representativas:
Uma lista abrangente está em *www.daveeaton.com/scm/CMTools.html*

Ferramentas de desempenho de banco de dados medem desempenho de banco de dados, por exemplo, o tempo para realizar consultas selecionadas ao banco de dados. Essas ferramentas facilitam *otimização* do banco de dados.

Ferramentas Representativas: BMC Software (*www.bmc.com*)

Depuradores são ferramentas típicas de programação que encontram e resolvem defeitos de software no código. Elas são parte dos mais modernos ambientes de desenvolvimento de aplicação.

Ferramentas Representativas:
Accelerated Technology (*www.acceleratedtechnology.com*)
IBM *VisualAge Environment* (*www.ibm.com*)
JDebugTool (*www.debugtools.com*)

Sistemas de gestão de defeitos registram defeitos e monitoram seu estado e solução. Algumas incluem ferramentas de relato para fornecer informação gerencial sobre taxas de disseminação dos defeitos e de resolução de defeitos.

Ferramentas Representativas:
EXCEL *Quickbugs* (*www.excelsoftware.com*)
McCabe *TRUETrack* (*www.mccabe.com*)
Rational *ClearQuest* (*www.rational.com*)

Ferramentas de monitoramento de rede observam o nível de tráfego da rede. São úteis para identificar engarrafamentos da rede e testar a ligação entre os sistemas frontal e de retaguarda (front-end back-end systems).

Ferramentas Representativas:
Uma lista abrangente está em *www.slac.stanford.edu/xorg/nmtf/nmtf-tools.html*

Ferramentas de teste de regressão armazenam casos de teste e dados de teste, e podem reaplicar os casos de teste depois de modificações sucessivas no software.

Ferramentas Representativas:
Compuware *QARun* (*www.compuware.com/products/qacenter/qarun*)
Rational *VisualTest* (*www.rational.com*)
Seque Software (*www.seque.com*)

Ferramentas de monitoração de site monitoram o desempenho de sites freqüentemente a partir da perspectiva do usuário. Use-as para compilar estatísticas tais como tempo de resposta de ponta a ponta e vazão, e para verificar periodicamente a disponibilidade de um site.

Ferramentas Representativas:
Keynote Systems (*www.keynote.com*)

Ferramentas de esforço ajudam os desenvolvedores a explorar o comportamento do sistema sob altos níveis de uso operacional e encontrar os pontos de quebra de um sistema.

Ferramentas Representativas:
Mercury Interactive (*www.merc-int.com*)
Scapa Technologies (*www.scapatech.com*)

Monitores de recursos do sistema são parte da maioria do software de SO de servidor e servidor da Web; monitoram recursos tais como espaço em disco, uso de CPU e memória.

Ferramentas Representativas:
Successful Hosting.com (*www.successfulhosting.com*)
Quest *Software Foglight* (*www.quest.com*)

Ferramentas de geração de dados de teste apóiam os usuários na geração de dados de teste.

Ferramentas Representativas:
Uma lista abrangente está em *www.softwareqatest.com/qatweb1.html*

Comparadores de resultado de teste ajudam a comparar os resultados de um conjunto de testes aos de outro conjunto. Use-os para verificar se alterações de código não introduziram modificações adversas no comportamento do sistema.

Ferramentas Representativas:
Uma lista útil está em *www.aptest.com/resources.html*

Monitores de transação medem o desempenho de sistemas de processamento de alto volume de transações.

Ferramentas Representativas:
QuotiumPro (*www.quotium.com*)
Software Research *eValid* (*www.soft.com/eValid/index.html*)

Ferramentas de segurança de site da Web ajudam a detectar problemas de segurança em potencial. Você pode, freqüentemente, estabelecer ferramentas de prova e monitoramento de segurança para serem executadas com base em um cronograma.

Ferramentas Representativas:
Uma lista abrangente está em *www.timberlinetechnologies.com/products/www.html*

18 As ferramentas mencionadas aqui não representam uma recomendação, mas, em vez disso, uma exemplificação de ferramentas dessa categoria. Além disso, seus nomes são marcas registradas das empresas mencionadas.

20.10 RESUMO

O objetivo do teste de WebApp é exercitar cada uma das várias dimensões de qualidade da WebApp com a intenção de encontrar erros ou descobrir tópicos que podem levar a falhas de qualidade. O teste enfoca conteúdo, função, estrutura, usabilidade, navegabilidade, desempenho, compatibilidade, interoperabilidade, capacidade e segurança. O teste também incorpora revisões que ocorrem enquanto a WebApp é projetada.

A estratégia do teste de WebApp exercita cada uma das dimensões de qualidade inicialmente examinando as "unidades" de conteúdo, funcionalidade ou navegação. Uma vez validadas as unidades individuais, o foco desloca-se para testes que exercitam a WebApp como um todo. Para conseguir isso, muitos testes são derivados das perspectivas de usuários e são guiados por informação contida em casos de uso. O plano de teste de engenharia da Web é desenvolvido e identifica passos de teste, produtos de trabalho (por exemplo, casos de teste) e mecanismos para avaliação de resultados do teste. O processo engloba sete diferentes tipos de teste.

Teste de conteúdo (e revisões) enfoca várias categorias de conteúdo. O objetivo é descobrir tanto erros semânticos quanto sintáticos que afetam a precisão do conteúdo ou o modo pelo qual ele é apresentado ao usuário final. Teste de interface exercita os mecanismos de interação que permitem

Document shared on www.docsity.com

a um usuário se comunicar com a WebApp e validar a aparência estética da interface. O objetivo é descobrir erros que resultem de mecanismos de interação mal implementados, ou omissões, inconsistências ou ambigüidades na semântica de interface.

Teste de navegação aplica casos de uso, derivados como parte da atividade de análise, no projeto de casos de teste que exercitam cada cenário de uso com base no projeto de navegação. Mecanismos de navegação são testados para garantir que quaisquer erros que impedem a conclusão de um caso de uso sejam identificados e corrigidos. Teste de componente exercita unidades de conteúdo e funcionais da WebApp. Cada página da Web encapsula conteúdo, links de navegação e elementos de processamento que formam uma "unidade" na arquitetura da WebApp. Essas unidades precisam ser testadas.

Teste de configuração tenta descobrir erros e/ou problemas de compatibilidade específicos de um ambiente particular de cliente ou servidor. Testes são conduzidos para descobrir erros associados com cada configuração possível. Teste de segurança incorpora uma série de testes projetados para explorar vulnerabilidades na WebApp e em seu ambiente. O objetivo é encontrar brechas de segurança. Teste de desempenho engloba uma série de testes projetados para avaliar o tempo de resposta e confiabilidade da WebApp à medida que as demandas de capacidade de recursos do lado do servidor aumentam.

Referências Bibliográficas

[BRO01] Brown, B., *Oracle9i Web Development*, McGraw-Hill, 2ª ed., 2001.

[CAC02] Cachero, C. *et al.*, "Conceptual Navigation Analysis: A Device and Platform Independent Navigation Specification", *Proc. 2nd Intl. Workshop on Web-Oriented Technology*, jun. 2002, disponível em www.dsic.upv.es/_west/iwwost02/papers/cachero.pdf.

[CON03] Constantine, L. e Lockwood, L., *Software for Use*, Addison-Wesley, 1999; veja também http://www.foruse.com/.

[GAR02] Garfinkel, S. e Spafford, G., *Web Security, Privacy and Commerce*, O'Reilly & Associates, 2002.

[HOW97] Hower, Rick, "Beyond Broken Links", *Internet Systems*, 1997, disponível em http://www.dbmsmag.com/9707i03.html.

[LAM01] Lam, W., "Testing E-Commerce Systems: A Practical Guide", *IEEE IT Pro*, mar./abr. 2001, p. 19-28.

[MCC03] McClure, S., Shah, S., *Web Hacking: Attacks and Defense*, Addison-Wesley, 2003.

[MIL00] Miller, E., "WebSite Testing", 2000, disponível em http://www.soft.com/eValid/Technology/ White.Papers/website.testing.html.

[NGU00] Nguyen, H., *Testing Web-Based Applications, Software Testing and Quality Engineering*, maio/jun. 2000, disponível em http://www.stqemagazine.com.

[NGU01] _____, *Testing Applications on the Web*, Wiley, 2001.

[SCE02] Sceppa, D., *Microsoft ADO.NET*, Microsoft Press, 2002.

[SPL01] Splaine, S. e Jaskiel, S., *The Web Testing Handbook*, STQE Publishing, 2001.

[TRE03] Trivedi, R., *Professional Web Services Security*, Wrox Press, 2003.

[WAL03] Wallace, D., Raggett I., e Aufgang, J., *Extreme Programming for Web Projects*, Addison-Wesley, 2003.

Problemas e Pontos a Considerar

20.1. Existem situações nas quais o teste de WebApp deve ser totalmente descartado?

20.2. Em suas próprias palavras, discuta os objetivos do teste em um contexto de engenharia da Web.

20.3. Compatibilidade é uma importante dimensão de qualidade. O que precisa ser testado para garantir que exista compatibilidade em uma WebApp?

20.4. Que erros tendem a ser mais sérios — erros do lado do cliente ou erros do lado do servidor? Por quê?

20.5. Que elementos de WebApp podem ser submetidos ao "teste de unidade"? Que tipos de testes devem ser conduzidos apenas depois que os elementos da WebApp tenham sido integrados?

20.6. É sempre necessário desenvolver um plano de teste formal por escrito? Explique.

20.7. É justo dizer que a estratégia global do teste de WebApp começa com os elementos visíveis ao usuário e desloca-se para os elementos de tecnologia? Há exceções para essa estratégia?

20.8. O teste de conteúdo é *realmente* teste no sentido convencional? Explique.

20.9. Descreva os passos associados com teste de banco de dados para uma WebApp. O teste de banco de dados é uma atividade predominantemente do lado do cliente ou do lado do servidor?

20.10. Qual é a diferença entre teste associado a um mecanismo de interface e teste que trata da semântica de interface?

20.11. Considere que você está desenvolvendo uma farmácia on-line (*CornerPharmacy.com*) que se destina a idosos. A farmácia fornece funções típicas, mas também mantém um banco de dados para cada cliente de modo que possa fornecer informação sobre remédios e alertar possíveis efeitos colaterais do remédio? Discuta quaisquer testes especiais de usabilidade para essa WebApp.

20.12. Considere que você implementou uma função de verificação de efeito colateral de remédio para *CornerPharmacy.com* (Problema 20.11). Discuta os tipos de teste no nível de componente que teriam de ser conduzidos para garantir que essa função funcione adequadamente. [Nota: um banco de dados teria que ser usado para implementar essa função.]

20.13. Qual é a diferença entre teste de sintaxe de navegação e de semântica de navegação?

20.14. É possível testar toda a configuração que uma WebApp pode encontrar no lado do servidor? No lado do cliente? Se não for, como um engenheiro da Web seleciona um conjunto significativo de testes de configuração?

20.15. Qual é o objetivo de teste de segurança? Quem executa essa atividade de teste?

20.16. *CornerPharmacy.com* (Problema 20.11) tornou-se muitíssimo bem-sucedida e o número de usuários aumentou dramaticamente nos primeiros dois meses de operação. Faça um gráfico que mostre os prováveis tempo de resposta como função do número de usuários para um conjunto fixo de recursos do lado do servidor. Rotule o gráfico para indicar os pontos de interesse da "curva de resposta".

20.17. Em resposta ao seu sucesso, *CornerPharmacy.com* (Problema 20.11) implementou um servidor especial somente para manipular repetições de receita. Em média, mil usuários concorrentes submetem um pedido de repetição a cada 2 minutos. A WebApp baixa um bloco de dados de 500 bytes em resposta. Qual é a vazão aproximada requerida para esse servidor em megabits por segundo?

20.18. Qual a diferença entre carga de teste e carga de esforço?

Leituras e Fontes de Informação Adicionais

A literatura de teste de WebApp ainda está evoluindo. Livros de Ash (*The Web Testing Companion*, Wiley, 2003), Dustin e seus colegas (*Quality Web Systems*, Addison-Wesley, 2002), Nguyen [NGU01] e Splaine e Jaskiel [SPL01] estão entre as abordagens mais completas do assunto publicadas até hoje. Mosley (*Client-Server Software Testing on the Desktop and the Web*, Prentice-Hall, 1999) trata tanto de tópicos de teste do lado do cliente quanto do lado do servidor.

Informação útil sobre estratégias e métodos de teste de WebApp, bem como uma valiosa discussão de ferramentas automatizadas de teste, é apresentada por Stottlemeyer (*Automated Web Testing Toolkit*, Wiley, 2001). Graham e seus colegas (*Software Test Automation*, Addison-Wesley, 1999) apresentam material adicional sobre ferramentas automatizadas.

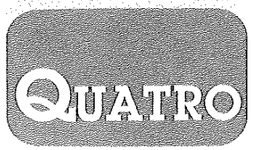
Nguyen e seus colegas (*Testing Applications for the Web*, segunda edição, Wiley, 2003) desenvolveram uma atualização importante para [NGU01] e fornecem diretrizes exclusivas para teste de aplicações móveis. Embora a Microsoft (*Performance Testing Microsoft .NET Web Applications*, Microsoft Press, 2002) enfoque predominantemente o seu ambiente.NET, seus comentários sobre testes de desempenho podem ser úteis para qualquer interessado no assunto.

Splaine (*Testing Web Security*, Wiley, 2002), Klevinsky e seus colegas (*Hack I.T.: Security through Penetration Testing*, Addison-Wesley, 2002), Chirillo (*Hack Attacks Revealed*, segunda edição, Wiley, 2003) e Skoudis (*Counter Hack*, Prentice-Hall, 2001) fornecem muita informação útil para aqueles que precisam projetar testes de segurança.

Uma ampla variedade de fontes de informação sobre teste de engenharia da Web está disponível na Internet. Uma lista atualizada de referências da World Wide Web pode ser encontrada no site deste livro: **http://www.mhhe.com/pressman**.

Document shared on www.docsity.com

PARTE



GESTÃO DE PROJETOS DE SOFTWARE

Nesta parte de *Engenharia de Software* vamos considerar as técnicas de gestão necessárias para planejar, organizar, monitorar e controlar projetos de software. As questões seguintes são tratadas nos próximos capítulos:

- Como as pessoas, processos e problemas precisam ser geridos durante um projeto de software?
- Como as métricas de software podem ser usadas para gerir projeto de software e processo de software?
- Como vamos estimar esforço, custo e duração de projeto?
- Que técnicas podem ser usadas para avaliar formalmente os riscos que podem causar impacto no sucesso do projeto?
- Como um gerente de projeto de software seleciona um conjunto de tarefas de trabalho de engenharia de software?
- Como é criado um cronograma de projeto?
- O que é gestão de qualidade?
- Por que revisões técnicas formais são tão importantes?
- Como é gerida a modificação durante o desenvolvimento de software de computador e depois da entrega ao cliente?

Uma vez respondidas essas questões, você estará mais bem preparado para gerir projetos de software de um modo que leve à entrega de produto de alta qualidade, dentro do prazo.

Document shared on www.docsity.com

CAPÍTULO

21 CONCEITOS DE GESTÃO DE PROJETOS

CONCEITOS-CHAVE

equipes ágeis 488
coordenação 489
práticas críticas 495
pessoal 484
decomposição do problema 491
processo 491
projeto 493
escopo do software 490
equipe de software 486
interessados 485
líderes de equipe 485
princípio W^5HH 494

Meiler Page-Jones [PAG85], no prefácio do seu livro sobre gestão de projeto de software, faz uma afirmação que pode encontrar eco em muitos consultores de engenharia de software:

> Visitei dezenas de instalações comerciais, tanto boas como ruins, e observei um grande número de gerentes de processamento de dados, novamente, tanto bons quanto ruins. Freqüentemente, observei como esses gerentes lutavam inutilmente com projetos que constituíam um verdadeiro pesadelo, espremidos por prazos de entrega inexequíveis, ou entregavam sistemas que irritavam seus usuários e prosseguiam devorando enormes parcelas de tempo de manutenção.

O que Page-Jones descreve são sintomas que resultam de um conjunto de problemas técnicos e de gestão. Todavia, se fosse conduzida uma autópsia para cada projeto, é muito provável que um tema consistente fosse encontrado: a *gestão* do projeto era fraca.

Neste capítulo e nos seis que se seguem, consideramos os conceitos-chave que levam à *gestão* efetiva de projetos de software. Este capítulo considera os conceitos e princípios básicos de gestão de projetos de software. O Capítulo 22 apresenta métricas de processo e de projeto que são a base para a tomada de decisões gerenciais efetivas. As técnicas usadas para estimar o custo e definir um cronograma realístico, e estabelecer um plano de projeto efetivo, são discutidas nos Capítulos 23 e 24. As atividades de gerência que levam à monitoração, redução e gestão de risco efetivas são apresentadas no Capítulo 25. Finalmente, os Capítulos 26 e 27 consideram técnicas para garantir qualidade à medida que o projeto é desenvolvido e controlar as modificações ao longo da vida de uma aplicação.

PANORAMA

O que é? Apesar de muitos de nós (em nossos piores momentos) adotarmos a visão de "gestão" de Dilbert, ela continua uma atividade muito necessária quando sistemas e produtos baseados em computador são construídos. A gestão do projeto envolve o planejamento, a monitoração e o controle do pessoal, processo e eventos que ocorrem à medida que o software evolui de um conceito preliminar para uma implementação operacional.

Quem faz? Todos "gerenciam" em uma certa medida, mas o escopo das atividades de gestão varia com a pessoa que as executa. Um engenheiro de software gerencia suas atividades do dia-a-dia, planejando, monitorando e controlando tarefas técnicas. Gerentes de projeto planejam, monitoram e controlam o trabalho de uma equipe de engenheiros de software. Gerentes seniores coordenam a interface entre o negócio e os profissionais de software.

Por que é importante? A construção de software de computador é um empreendimento complexo, particularmente se envolver muitas pessoas trabalhando durante um período relativamente longo. Essa é a razão pela qual projetos de software precisam ser geridos.

Quais são os passos? Entenda os quatro Ps — pessoal, produto, processo e projeto. Pessoas precisam ser organizadas para realizar o trabalho de software efetivamente. A comunicação com o cliente precisa ocorrer para que o escopo e os requisitos do produto sejam entendidos. Um processo precisa ser selecionado a fim de se adequar ao pessoal e ao produto. O projeto precisa ser planejado, estimando o esforço e o tempo para executar as tarefas de trabalho: definição de produtos do trabalho, estabelecimento de marcos de qualidade e estabelecimento de mecanismos para monitorar e controlar o trabalho definido pelo plano.

Qual é o produto do trabalho? O plano de projeto é produzido à medida que as atividades de gerência são iniciadas. O plano define os processos e as tarefas a serem conduzidas, o pessoal que executará o trabalho e os mecanismos para avaliar riscos, controlar modificações e avaliar qualidade.

Como tenho certeza de que fiz corretamente? Você nunca vai estar completamente certo de que o plano de projeto está correto, até que tenha entregue um produto de alta qualidade, pontualmente, e dentro do orçamento. Todavia, um gerente de projeto acerta quando encoraja o pessoal de software a trabalhar junto, como uma equipe efetiva, focalizando sua atenção nas necessidades do cliente e na qualidade do produto.

21.1 O ESPECTRO DE GESTÃO

A *gestão* efetiva de projetos de software focaliza os quatros Ps: pessoal, produto, processo e projeto. A ordem não é arbitrária. O gerente que esquece que o trabalho de engenharia de software é um empreendimento intensamente humano nunca vai ter sucesso na gestão de projetos. Um gerente que desde cedo na evolução de um projeto deixa de encorajar uma comunicação ampla com os interessados se arrisca a construir uma solução elegante para o problema errado. O gerente que presta pouca atenção no processo corre o risco de empregar métodos e ferramentas técnicas competentes em um vazio. O gerente que começa sem um plano de projeto sólido compromete o sucesso do produto.

21.1.1 O Pessoal

O cultivo de pessoal de software motivado e altamente qualificado tem sido discutido desde os anos 60 (por exemplo, [COU80], [WIT94], [DEM98]). De fato, o "fator pessoal" é tão importante que o Software Engineering Institute desenvolveu um *modelo de maturidade da capacidade de gestão de pessoal* (*people management capability maturity model*, PM-CMM) "para melhorar a capacidade de organizações de software desenvolverem aplicações cada vez mais complexas, ajudando-as a atrair, desenvolver, motivar, dispor e reter o talento necessário para melhorar sua capacidade de desenvolvimento de software" [CUR94].

O modelo de maturidade de gestão de pessoal define as seguintes áreas de práticas-chave para pessoal de software: recrutamento, seleção, gestão de desempenho, treinamento, remuneração, desenvolvimento de carreira, organização e projeto do trabalho e desenvolvimento de equipe/cultura. Organizações que atingem altos níveis de maturidade na área de gestão de pessoal têm uma grande probabilidade de implementar práticas efetivas de engenharia de software.

O PM-CMM faz par com o modelo de maturidade da capacidade de software (Capítulo 2), que orienta organizações na criação de processos de software *amadurecidos*. Tópicos associados à gestão de pessoal e estrutura de projetos de software são considerados mais adiante neste capítulo.

AVISO

Nesse contexto, o termo produto é usado para abranger qualquer software construído a pedido de outros. Inclui não apenas produtos de software empacotados, mas também sistemas baseados em computador, software embutido, aplicações da Web e software de solução de problema, (por exemplo, programas para a solução de problemas de engenharia e científicos).

21.1.2 O Produto

Antes de um projeto ser planejado, devem ser estabelecidos os objetivos e o escopo do produto, soluções alternativas devem ser consideradas e as restrições técnicas e gerenciais devem ser identificadas. Sem essas informações é impossível definir estimativas de custo razoáveis (e precisas), avaliações efetivas do risco, relações realísticas de tarefas de projeto ou cronograma de projeto gerenciável que proporcionem uma indicação significativa do progresso.

O desenvolvedor de software e o cliente devem se reunir para definir os objetivos e o escopo do produto. Em muitos casos essa atividade começa como parte da engenharia de sistemas ou da engenharia de processo do negócio (Capítulo 6) e continua como o primeiro passo da engenharia de requisitos de software (Capítulo 7). Os objetivos identificam as metas gerais para o produto (do ponto de vista do cliente) sem considerar como essas metas serão atingidas. O escopo identifica os dados principais, as funções e os comportamentos que caracterizam o produto e, mais importante, tenta *definir* essas características de forma quantitativa.

Uma vez entendidos os objetivos e o escopo do produto, soluções alternativas são consideradas. Apesar de poucos detalhes serem discutidos, as alternativas permitem a gerentes e profissionais selecionar uma abordagem "melhor", dadas as restrições impostas pelos prazos de entrega, restrições orçamentárias, disponibilidade de pessoal, interfaces técnicas e muitos outros fatores.

Document shared on www.docsity.com

AVISO

Aqueles que aderem à filosofia do processo ágil (Capítulo 4) argumentam que seu processo é mais "enxuto" do que outros. Isso pode ser verdade, mas eles ainda têm um processo e engenharia de software ágil que ainda exige disciplina.

21.1.3 O Processo

Um processo de software (Capítulos 2, 3 e 4) fornece o arcabouço a partir do qual pode ser estabelecido um plano abrangente para o desenvolvimento de software. Algumas atividades de arcabouço são aplicáveis a todos os projetos de software, independentemente de seu tamanho ou complexidade. Vários conjuntos diferentes de tarefas — tarefas, eventos importantes, produtos de trabalho e pontos de garantia de qualidade — permitem que as atividades de arcabouço sejam adaptadas às características do projeto de software e às necessidades da equipe de projeto. Finalmente, atividades guarda-chuva — como garantia de qualidade de software, gestão de configuração de software e medição — cobrem o modelo de processo. As atividades guarda-chuva são independentes de qualquer atividade de arcabouço e ocorrem ao longo de todo o processo.

21.1.4 O Projeto

Conduzimos projetos de software planejados e controlados por uma razão principal — é a única forma conhecida de gerir a complexidade. E, no entanto, ainda sofremos. Em 1998, dados da indústria indicavam que 26% dos projetos de software falharam de imediato e 46% ultrapassavam os custos e os prazos [REE99]. Apesar de a taxa de sucesso de projetos de software ter melhorado em parte, nossa taxa de falhas de projeto permanece maior do que deveria.[1]

> "Um projeto é como uma viagem em rodovia. Alguns projetos são simples e rotineiros, como dirigir até a loja em plena luz do dia. Mas, a maioria dos projetos que valem a pena é mais parecida com dirigir um caminhão fora da estrada, nas montanhas."
>
> **Cem Kaner, James Bach e Bret Pettichord**

Para evitar falha de projeto, um gerente de projeto de software e os engenheiros de software, que constroem o produto, devem evitar um conjunto de sinais de alerta comuns, entender os fatores críticos de sucesso, que levam à boa gestão de projetos, e desenvolver uma abordagem de bom senso para planejar, monitorar e controlar o projeto. Cada um desses assuntos é discutido na Seção 21.5 e nos capítulos seguintes.

21.2 PESSOAL

Em um estudo publicado pelo IEEE [CUR88], os vice-presidentes de engenharia de três importantes empresas de tecnologia foram solicitados a mencionar o fator mais importante para um projeto de software bem-sucedido. Responderam do seguinte modo:

> VP 1: Se tivesse que escolher a coisa mais importante em nosso ambiente eu diria que são as pessoas, e não as ferramentas que usamos.
>
> VP 2: O ingrediente mais importante que foi bem-sucedido neste projeto foi ter pessoal competente... pouca coisa a mais conta, em minha opinião... O mais importante que você faz para um projeto é selecionar a equipe... O sucesso de uma organização de desenvolvimento de software está muito, muito associado com a capacidade de recrutar pessoal bom.
>
> VP 3: A minha única regra na gestão é garantir que eu possa contar com pessoal bom — pessoal realmente bom — e desenvolver pessoal bom — e oferecer um ambiente no qual esse pessoal possa produzir.

Realmente, essa é uma razão convincente que atesta a importância das pessoas no processo de engenharia de software. Contudo, todos nós, de vice-presidentes seniores de engenharia aos mais simples profissionais, freqüentemente não damos valor às pessoas. Os gerentes argumentam (como o grupo anterior fez) que as pessoas são importantes, mas suas ações algumas vezes contradizem suas palavras. Nesta seção examinamos os interessados que participam no processo de software e a maneira pela qual são organizados para realizar engenharia de software efetiva.

1 Diante dessas estatísticas é razoável perguntar como o impacto dos computadores continua a crescer exponencialmente. Parte da resposta é que um número substancial desses projetos que "falharam" é mal-concebido, em primeiro lugar. Os clientes perdem o interesse rapidamente (porque o que eles solicitaram não era realmente tão importante quanto pensavam inicialmente), e os projetos são cancelados.

? O que procuramos quando selecionamos alguém para liderar um projeto de software?

21.2.1 Os Interessados

O processo de software (e todo o projeto de software) é povoado por interessados que podem ser classificados em uma das seguintes categorias:

1. *Gerentes seniores* — definem os aspectos do negócio que freqüentemente têm influência significativa sobre o projeto.
2. *Gerentes de projeto (técnicos)* — devem planejar, motivar, organizar e controlar os profissionais que fazem o trabalho de software.
3. *Profissionais* — fornecem as aptidões técnicas necessárias para fazer a engenharia de um produto ou aplicação.
4. *Clientes* — especificam os requisitos para o software submetido à engenharia e outros interessados com interesse superficial no resultado.
5. *Usuários finais* — interagem com o software depois que ele é liberado para uso.

Todo projeto de software é constituído por pessoas que se encaixam nessa taxonomia.[2] Para ser efetiva, a equipe de projeto deve ser organizada com o objetivo de maximizar as aptidões e habilidades de cada pessoa. Essa é a tarefa do líder da equipe.

21.2.2 Líderes de Equipe

A gestão de projeto é uma atividade intensa em termos de pessoal e, por essa razão, profissionais competentes freqüentemente se tornam líderes de equipe fracos. Eles simplesmente não são hábeis para lidar com pessoas. E ainda, como Edgemon afirma: "Infelizmente, e com freqüência, pessoas se vêem na posição de gerência de projeto e viram gerentes de projeto por acidente" [EDG95].

Em um excelente livro sobre técnicas de liderança, Jerry Weinberg [WEI86] sugere um modelo MOI de liderança:

Motivação. Habilidades de encorajar ("puxando ou empurrando") o pessoal técnico a produzir no melhor de sua capacidade.

Organização. Habilidade de moldar processos existentes (ou inventar novos) permitindo que o conceito inicial seja traduzido em um produto final.

Idéias ou inovação. Habilidade de encorajar o pessoal a criar e a se sentir criativo, mesmo quando precisar trabalhar dentro de limites estabelecidos para um produto ou aplicação particular de software.

Weinberg considera que líderes de projeto bem-sucedidos aplicam um estilo de gestão de solução de problemas. Isto é, um gerente de projeto de software deve se concentrar no entendimento do problema a ser resolvido, gerir o fluxo de idéias e, ao mesmo tempo, deixar todo mundo na equipe saber (por palavras e, mais significativamente, por ações) que qualidade conta e que ela não deve ser comprometida.

> "De forma simples, líder é aquele que sabe aonde quer ir, levanta-se e vai."
>
> **John Erskine**

Outra visão [EDG95] das características que definem um gerente de projeto efetivo enfatiza quatro aspectos-chave:

Solução de problemas. Um gerente de projeto de software efetivo pode diagnosticar os aspectos técnicos e organizacionais, que são os mais relevantes, estruturar sistematicamente uma solução ou motivar adequadamente outros profissionais a desenvolver a solução, aplicar lições

2 Quando aplicações da Web são desenvolvidas (Parte 3 deste livro) outro pessoal não técnico pode ser envolvido na criação do conteúdo.

Document shared on www.docsity.com

aprendidas em projetos anteriores a novas situações e, se as tentativas iniciais de solução do problema não frutificarem, ser suficientemente flexível para mudar de rumo.

Identidade gerencial. Um bom gerente de projeto deve assumir o encargo do projeto. Deve ter confiança quanto a assumir o controle quando necessário e ter segurança para permitir que o pessoal técnico de qualidade siga seus instintos.

Realização. Para otimizar a produtividade de uma equipe de projeto, o gerente deve premiar a iniciativa e a realização e demonstrar, com suas próprias ações, que assumir risco controlado não implica punição.

Influência e construção de espírito de equipe. Um gerente de projeto eficiente deve conseguir "ver" as pessoas, entender os sinais verbais e não-verbais emitidos por elas e reagir às suas necessidades. O gerente deve manter o controle em situações de alta tensão.

21.2.3 A Equipe de Software

Há quase tantas estruturas organizacionais humanas para desenvolvimento de software quanto organizações que desenvolvem software. As estruturas organizacionais não podem ser facilmente modificadas, nem para melhor nem para pior. As preocupações com as conseqüências práticas e políticas de modificações organizacionais não estão no âmbito de responsabilidade dos gerentes de projeto de software. Todavia, a organização do pessoal diretamente envolvido em um novo projeto de software é responsabilidade do gerente de projeto.

> "Nem todo grupo é uma equipe e nem toda equipe está preparada."
>
> **Glenn Parker**

A "melhor" estrutura de equipe depende do estilo de gestão de sua organização, da quantidade de pessoas que formarão a equipe e seus níveis de aptidão, e da dificuldade geral do problema. Mantei [MAN81] descreve sete fatores de projeto que devem ser considerados quando se planeja a estrutura de equipes de engenharia de software:

? Que fatores devem ser considerados quando a estrutura de uma equipe de software é escolhida?

- A dificuldade do problema a ser resolvido.
- O tamanho do(s) programa(s) resultante(s) em linhas de código ou pontos por função (Capítulo 22).
- O período durante o qual a equipe ficará junta (período de vida da equipe).
- O grau de modularização que o problema admite.
- A qualidade e a confiabilidade exigidas pelo sistema a ser construído.
- A rigidez do prazo de entrega.
- O grau de sociabilidade (comunicação) exigido pelo projeto.

> "Se você quer ser incrementalmente o melhor: seja competitivo. Se você quer ser exponencialmente o melhor: seja cooperativo."
>
> **Autor desconhecido**

Constantine [CON93] sugere quatro "paradigmas organizacionais" para equipes de engenharia de software:

? Quais opções devemos ter definimos a de uma software?

1. *Um paradigma fechado* estrutura uma equipe ao longo de uma hierarquia tradicional de autoridades. Essas equipes podem trabalhar bem quando produzem software bastante semelhante a anteriores, mas é menos provável que sejam inovativas quando trabalham dentro do paradigma fechado.
2. O *paradigma aleatório* estrutura uma equipe fracamente e depende da iniciativa individual dos seus membros. Quando é necessário inovação ou avanço tecnológico, equipes que seguem o paradigma aleatório se distinguem. Mas tais equipes podem relutar quando for necessário "desempenho ordenado".
3. O *paradigma aberto* tenta estruturar uma equipe não só para conseguir alguns dos controles associados com o paradigma fechado, mas também a maior parte da inovação que ocorre quando se usa o paradigma aleatório. O trabalho é realizado em colaboração, com intensa comunicação e tomada de decisões baseadas no consenso, que é a marca registrada das equipes do paradigma aberto. As estruturas da equipe de paradigma aberto são adequadas à solução de problemas complexos, mas podem não ter desempenho tão eficiente como o de outras equipes.
4. O *paradigma síncrono* apóia-se na compartimentalização natural de um problema e organiza os membros da equipe para trabalhar em partes desse problema, com pouca comunicação ativa entre eles.

> "Trabalhar com pessoas é difícil, mas não impossível."
>
> **Peter Drucker**

Como nota histórica, uma das primeiras organizações de equipe de software foi uma estrutura de paradigma fechado originalmente chamada de *chief programmer team* (equipe do programador-chefe). Essa estrutura foi primeiramente proposta por Harlan Mills e descrita por Baker [BAK72]. O núcleo da equipe era composto por um *engenheiro sênior* (o programador-chefe), que planeja, coordena e revê todas as atividades técnicas da equipe; *grupo técnico* (normalmente duas a cinco pessoas), que executa as atividades de análise e desenvolvimento; e um *engenheiro de retaguarda* (backup engineering), que apóia o engenheiro sênior nas suas atividades e pode substituí-lo, com perda mínima de continuidade no projeto.

O programador-chefe pode ser assessorado por um ou mais *especialistas* (por exemplo, especialista em telecomunicações, projetista de bases de dados), pessoal de apoio (por exemplo, redatores técnicos, escriturários), e por um *bibliotecário de software*.

Como contrapartida da estrutura da equipe do programador-chefe, o paradigma aleatório de Constantine [CON93] sugere uma equipe de software com independência criativa, cuja abordagem de trabalho pode ser mais bem denominada *anarquia inovativa*. Apesar de a abordagem de espírito liberal para o trabalho de software ser atraente, a canalização de energia criativa para uma equipe de alto desempenho deve ser a meta central de uma organização de engenharia de software. Para conseguir uma equipe de alto desempenho:

- Os membros da equipe devem confiar uns nos outros.
- A distribuição de aptidões deve ser adequada ao problema.
- Estrelas podem ter que ser excluídas da equipe, se a coesão tiver que ser mantida.

Independentemente da organização da equipe, o objetivo de todo gerente de projeto é ajudar a criar uma equipe coesa. Em seu livro *Peopleware*, DeMarco e Lister [DEM98] discutem esse assunto:

? O que é uma equipe "aglutinada"?

> Tendemos a usar a palavra *equipe* de maneira bastante imprecisa no mundo dos negócios, chamando qualquer grupo de pessoas, designadas para trabalhar juntas, de "equipe". Mas, muitos desses grupos simplesmente não parecem equipes. Não têm uma definição comum de sucesso, nem nenhum espírito de equipe identificável. O que está faltando é um fenômeno que chamamos *aglutinação* (jell).
>
> Uma equipe aglutinada é um grupo de pessoas tão fortemente unidas que o todo é maior que a soma das partes...
>
> Quando uma equipe começa a se aglutinar, a probabilidade de sucesso cresce muito. Pode ficar difícil de ser contida, vira um bólido em direção ao sucesso... Não precisa ser gerida de modo tradicional e certamente não precisa ser motivada. Adquire energia de movimento (*momentum*).

Document shared on www.docsity.com

DeMarco e Lister alegam que os membros de equipes aglutinadas são significativamente mais produtivos e mais motivados do que a média. Compartilham um objetivo comum, uma cultura comum e, em muitos casos, um "senso de elite" que os tornam singulares.

? Por que deixam de se aglutinar?

Mas nem todas as equipes se aglutinam. De fato, muitas equipes sofrem do que Jackman [JAC98] chama "intoxicação da equipe". Ela define cinco fatores que "provocam um ambiente de equipe potencialmente tóxico".

1. Uma atmosfera de trabalho frenética.
2. Alta frustração que causa fricção entre membros da equipe.
3. Um processo de software "fragmentado ou mal coordenado".
4. Uma definição imprecisa de papéis na equipe de software.
5. "Exposição a falhas contínua e repetidamente."

Para evitar um ambiente de trabalho frenético, o gerente de projeto deve se certificar de que a equipe tem acesso a toda informação necessária para executar o trabalho e que as metas e objetivos principais, uma vez definidos, não devem ser modificados, a menos que seja absolutamente necessário. Uma equipe de software pode evitar frustração (e tensão) se lhe for dada tanta responsabilidade de tomada de decisão quanto possível.

Um processo de software mal-escolhido (por exemplo, tarefas de trabalho desnecessárias, ou cansativas, ou ainda produtos de trabalho mal-escolhidos) pode ser evitado pelo entendimento do produto a ser construído e das pessoas que estão fazendo o trabalho, e pela permissão de que a equipe selecione o seu próprio modelo de processo.

A equipe, por sua vez, deve estabelecer seus próprios mecanismos de prestação de contas (revisões técnicas formais e programação aos pares são excelentes modos de conseguir isso) e quando um membro da equipe não alcançar bom desempenho, definir uma série de medidas corretivas. E, finalmente, a chave para evitar uma atmosfera de falha é estabelecer técnicas, baseadas na equipe, para realimentação e solução de problemas.

> "Faça ou não faça; não tente."
>
> **Yoda (*Star Wars*)**

Além das cinco toxinas descritas por Jackman, uma equipe de software luta freqüentemente com as diferenças das personalidades de seus membros. Alguns são extrovertidos, outros introvertidos. Algumas pessoas colhem informação intuitivamente, abstraindo amplos conceitos a partir de fatos disparatados. Outros processam informação linearmente, coletando e organizando pequenos detalhes a partir dos dados fornecidos. Alguns membros da equipe sentem-se seguros para tomar decisões apenas quando uma argumentação lógica e ordenada é apresentada. Outros são intuitivos, dispostos a tomar decisão com base no "sentimento". Alguns profissionais querem um cronograma detalhado constituído por tarefas organizadas, que lhes permita alcançar o fechamento para algum elemento do projeto. Outros preferem um ambiente mais espontâneo no qual são aceitáveis aspectos em aberto. Alguns trabalham duro para deixar as coisas feitas antes da data marcada, evitando assim que haja tensão à medida que a data se aproxima, enquanto outros obtêm energia na corrida para cumprir o prazo no último minuto. Discussão detalhada da psicologia dessas personalidades e dos modos pelos quais um líder de equipe competente pode ajudar pessoas com personalidades opostas a trabalhar juntas foge do objetivo deste livro.[3] Todavia, é importante notar que o reconhecimento de diferenças humanas é o primeiro passo na direção de criar equipes que se aglutinam.

21.2.4 Equipes Ágeis

Nos últimos anos, desenvolvimento ágil de software (Capítulo 4) foi proposto como antídoto para muitos dos problemas que afligiram o trabalho em projetos de software. Para lembrar: a filosofia ágil incentiva a satisfação do cliente e a entrega incremental de software desde o início; pequenas equipes de projeto altamente motivadas; métodos informais; poucos produtos de trabalho de engenharia de software e simplicidade em todo o desenvolvimento.

3 Uma introdução excelente a esses assuntos, na medida em que se relacionam com equipes de projeto de software, pode ser encontrada em [FER98].

A pequena equipe de projeto altamente motivada, também chamada *equipe ágil*, adota muitas das características de equipes de projeto de software bem-sucedidas discutidas na seção anterior e evita muitas das toxinas que criam problemas. No entanto, a filosofia ágil enfatiza a competência individual (membro da equipe) combinada com a colaboração do grupo como fatores críticos de sucesso da equipe. Cockburn e Highsmith [COC01] ressaltam isso quando escrevem:

> Se o pessoal do projeto é suficientemente bom, pode usar quase qualquer processo e cumprir a tarefa. Se não é suficientemente bom, nenhum processo vai reparar sua inadequação — "o pessoal atrapalha o processo" é um modo de dizer isso. No entanto, a falta de apoio de usuário e executivo pode matar um projeto — "a política atrapalha o pessoal". Apoio inadequado pode impedir que mesmo pessoal bom execute o serviço...

Para fazer uso efetivo das competências de cada membro da equipe e para incentivar a efetiva colaboração em um projeto de software, equipes ágeis são *auto-organizadas*. Uma equipe auto-organizada não necessariamente mantém uma única estrutura de equipe, mas em vez disso usa elementos dos paradigmas aleatório, aberto e síncrono de Constantine discutidos na Seção 21.2.3.

> "Propriedade coletiva é nada mais do que uma instanciação da idéia de que produtos devem ser atribuíveis à equipe [ágil], não aos indivíduos que formam a equipe."
>
> **Jim Highsmith**

PONTO CHAVE

Uma equipe ágil é uma equipe auto-organizada que tem autoridade para planejar e tomar decisões técnicas.

Muitos modelos de processo ágeis (por exemplo, SCRUM) dão à equipe ágil significativa autonomia para tomar as decisões de projeto gerenciais e técnicas necessárias para fazer com que o serviço seja feito. O planejamento é reduzido ao mínimo e é permitido à equipe selecionar a sua própria abordagem (por exemplo, processo, métodos, ferramentas), restrita apenas pelos requisitos de negócio e normas organizacionais. À medida que o projeto prossegue, a equipe se auto-organiza para focalizar a competência individual de um modo que seja o mais benéfico para o projeto em um dado momento. Para conseguir isso, uma equipe ágil pode conduzir rápidas reuniões da equipe para coordenar e sincronizar o trabalho que deve ser efetuado naquele dia.

Com base na informação obtida durante essas reuniões, a equipe adapta sua abordagem para realizar um incremento do trabalho. A cada dia que passa, a auto-organização e colaboração contínuas movem a equipe para completar um incremento de software.

21.2.5 Problemas de Coordenação e Comunicação

Há muitas razões pelas quais os projetos de software enfrentam problemas. A escala de muitos esforços de desenvolvimento é grande, causando complexidade, confusão e significativas dificuldades em coordenar membros da equipe. Incerteza é comum, resultando em uma contínua corrente de modificações que perturba a equipe de projeto. Interoperabilidade tornou-se característica-chave de muitos sistemas. O software novo precisa comunicar-se com o software existente e atender a restrições predefinidas, impostas pelo sistema ou produto.

Essas características do software moderno — escala, incerteza e interoperabilidade — são fatos da vida. Para lidar efetivamente com eles, uma equipe de engenharia de software precisa estabelecer métodos específicos para coordenar o pessoal que faz o trabalho. Para tanto, devem ser estabelecidos mecanismos para comunicação formal e informal entre os membros da equipe e entre diferentes equipes. Comunicação formal é conseguida por meio de "documentos escritos, reuniões estruturadas e outros canais de comunicação relativamente não interativos e impessoais" [KRA95]. Comunicação informal é mais pessoal. Membros de uma equipe de software trocam idéias naturalmente, pedem ajuda quando surgem problemas e interagem uns com os outros no dia-a-dia.

Document shared on www.docsity.com

CASASEGURA

Estrutura da Equipe

A cena: Escritório de Doug Miller antes do início do projeto de software CasaSegura.

Os personagens: Doug Miller (gerente da equipe de engenharia de software do CasaSegura) e Vinod Ramon, Jamie Lazar e outros membros da equipe do produto de engenharia de software.

A conversa:

Doug: Vocês tiveram a oportunidade de passar os olhos pela informação preliminar sobre Segurança Domiciliar que o pessoal de marketing preparou?

Vinod (balançando a cabeça afirmativamente e olhando os seus companheiros de equipe): Sim, mas nós temos várias questões.

Doug: Vamos deixar como está por um instante. Eu gostaria de falar sobre como nós vamos estruturar a equipe, quem vai ser responsável pelo quê...

Jamie: Eu conheço realmente a filosofia ágil, Doug. Acho que nós deveríamos ser uma equipe auto-organizada.

Vinod: Eu concordo. Dado o prazo apertado e alguma incerteza e o fato de que nós somos todos competentes [risadas], essa parece ser a maneira correta de proceder.

Doug: Tudo bem comigo, mas vocês conhecem o esquema.

Jamie (sorrindo e falando como se estivesse recitando alguma coisa): Nós tomamos decisões técnicas sobre quem faz o quê e quando, mas é nossa responsabilidade colocar o produto porta a fora em tempo.

Vinod: E com qualidade.

Doug: Exatamente. Mas lembrem-se de que há restrições. Marketing define os incrementos de software a serem produzidos – em consulta conosco, evidentemente.

Jamie: E?

Doug: E, nós vamos usar UML como nossa abordagem de modelagem.

Vinod: Mas limitar a documentação acessória a um mínimo absoluto.

Doug: Quem vai fazer a ligação comigo?

Jamie: Nós decidimos que Vinod vai ser o líder técnico — ele tem a maior experiência, assim ele vai fazer a ligação com você, mas fique à vontade para falar com qualquer um de nós.

Doug (rindo): Não se preocupe, vou fazer isso.

21.3 O PRODUTO

Um gerente de projeto de software depara com um dilema no início de um projeto de engenharia de software. São necessárias estimativas quantitativas e um plano organizado, mas não há informação sólida disponível. Uma análise detalhada dos requisitos de software pode fornecer a informação necessária para as estimativas, mas a análise freqüentemente precisa de semanas ou meses para ser concluída. Pior do que isso, as necessidades podem ser mutáveis, variando regularmente à medida que o projeto prossegue. No entanto, um plano é necessário "agora!".

Assim sendo, precisamos examinar o produto e o problema que ele pretende resolver no início do projeto. O escopo do produto deve ser, pelo menos, estabelecido e delimitado.

21.3.1 Escopo do Software

A primeira atividade de gestão de um projeto de software é a determinação do *escopo do software*. O escopo é definido pela resposta às seguintes questões:

Contexto. Como o software a ser construído se encaixa no contexto de um sistema maior, do produto ou do negócio, que restrições são impostas como resultado do contexto?

Objetivos da informação. Que objetos de dados visíveis para o cliente (Capítulo 8) são produzidos como saída pelo software? Que objetos de dados são necessários como entrada?

Função e desempenho. Que função o software desempenha para transformar os dados de entrada em saídas? Existem características especiais de desempenho a serem tratadas?

O escopo do projeto de software não deve ser ambíguo e deve ser inteligível para os níveis gerenciais e técnicos. Uma declaração do escopo do software deve ser delimitada. Isto é, dados quantitativos (por exemplo, quantidade de usuários simultâneos, tamanho da lista de destinatários, tempo de resposta máximo admissível) são declarados explicitamente; restrições e/ou limitações

AVISO

Se você não pode delimitar uma característica do software que pretende construir, relacione-a como um alto risco de projeto (Capítulo 25).

(por exemplo, o custo do produto restringe o tamanho da memória) são anotadas e fatores facilitadores (por exemplo, os algoritmos desejados são bem entendidos e estão disponíveis em C++) são descritos.

21.3.2 Decomposição do Problema

A decomposição do problema, algumas vezes chamada *particionamento* ou *elaboração do problema*, é uma atividade que se situa no núcleo da análise dos requisitos do software (Capítulos 7 e 8). Durante a atividade de determinação do escopo, nenhuma tentativa é feita para decompor totalmente o problema. Em vez disso, a decomposição é aplicada em duas áreas principais: (1) a funcionalidade que precisa ser entregue e (2) o processo que será usado para entregá-la.

Seres humanos tendem a aplicar a estratégia de dividir e conquistar quando deparam com um problema complexo. Ou seja, um problema complexo é particionado em problemas menores, que são mais gerenciáveis. Essa é a estratégia que se aplica quando o planejamento do problema tem início. As funções do software, identificadas na descrição do escopo, são avaliadas e refinadas para fornecer mais detalhes antes do início da estimativa (Capítulo 23). Como tanto as estimativas de custo quanto as de prazo são orientadas funcionalmente, algum grau de decomposição é sempre útil.

AVISO

Para desenvolver um plano de projeto razoável, você tem que decompor o problema. Isso pode ser conseguido usando uma lista de funções, ou com casos de uso, ou para trabalho ágil, histórias de usuário.

Considere, por exemplo, um projeto que construirá um produto de processamento de texto novo. Entre as características singulares do produto estão: entrada, tanto por voz contínua como pelo teclado, características extremamente sofisticadas de "edição automática de texto", capacidade de disposição de página, preparação automática de índices e sumário, e outras. O gerente do projeto precisa primeiro estabelecer uma declaração de escopo que delimita essas características (tanto quanto outras funções mais comuns, como edição, gestão de arquivos, produção de documentos e outras). Por exemplo, a entrada por voz contínua exige que o produto seja "treinado" pelo usuário? Especificamente, qual a capacidade oferecida pela característica de edição de texto? Qual a sofisticação da capacidade de disposição (leiaute) de página?

À medida que a declaração de escopo evolui, um primeiro nível de particionamento ocorre naturalmente. A equipe de projeto fica sabendo que o departamento de vendas falou com clientes potenciais e determinou que as seguintes funções devem fazer parte da edição automática do texto: (1) verificação da sintaxe, (2) verificação gramatical da sentença, (3) verificação de referências para documentos extensos (por exemplo, a referência correspondente a uma citação bibliográfica é encontrada na bibliografia?) e (4) validação da referência a seções e capítulos para documentos extensos. Cada uma dessas características representa uma subfunção a ser implementada no software. Cada uma pode ser mais refinada, se a decomposição facilitar o planejamento.

21.4 O PROCESSO

As atividades de arcabouço (Capítulo 2) que caracterizam o processo de software são aplicáveis a todos os projetos de software. O problema é selecionar o modelo de processo adequado para o software a ser trabalhado por uma equipe de engenharia de projeto.

O gerente do projeto deve decidir qual modelo de processo é mais apropriado para (1) o cliente que solicitou o produto e o pessoal que executará o trabalho, (2) as características do produto propriamente dito e (3) o ambiente de projeto no qual a equipe de software trabalha. Selecionado o modelo de processo, a equipe então define um plano preliminar de projeto, com base no conjunto de atividades comuns de arcabouço de processo. Uma vez estabelecido o plano preliminar, a decomposição do processo tem início. Isto é, deve ser criado um plano completo que reflita as tarefas de trabalho necessárias para preencher as atividades de arcabouço. Exploramos essas atividades resumidamente nas seções que se seguem e apresentamos um panorama mais detalhado no Capítulo 24.

21.4.1 Fusão do Produto e do Processo

O planejamento do projeto tem início com a fusão do produto e do processo. Cada função a ser trabalhada pela equipe de engenharia de software deve passar pelo conjunto de atividades de

Document shared on www.docsity.com

AVISO

Uma ferramenta automatizada de cronogramação de projetos pode ser usada para criar uma "rede de tarefas" (Capítulo 24). A rede é carregada com requisitos de recursos estimados, datas de início/fim e outros dados pertinentes. Essa rede carregada de recursos pode então ser usada para acompanhamento e controle do projeto.

PONTO CHAVE

O arcabouço de processo estabelece um esqueleto para o planejamento do projeto. Ele é adaptado alocando um conjunto de tarefas adequado ao projeto.

arcabouço que foi definido para a organização de software. Considere que a organização adotou o seguinte conjunto de atividades de arcabouço (Capítulo 2): Comunicações, Planejamento, Modelagem, Construção e Implantação.

Os membros da equipe, que trabalham em uma função do produto, aplicam a ela cada uma das atividades de arcabouço. Em resumo, é criada uma matriz semelhante àquela mostrada na Figura 21.1. Cada função principal do produto (a figura mostra as funções para o software de processamento de textos discutido anteriormente) é listada na coluna à esquerda. Atividades de arcabouço são listadas na linha superior. As tarefas de engenharia de software (para cada atividade de arcabouço) são colocadas na linha seguinte[4]. O trabalho do gerente de projeto (e dos outros membros da equipe) é estimar a necessidade de recursos para cada célula da matriz, datas de começo e fim para as tarefas associadas com cada célula e produtos de trabalho a serem produzidos como conseqüência de cada tarefa. Essas atividades são consideradas no Capítulo 24.

21.4.2 Decomposição do Processo

Uma equipe de software deve ter grau de flexibilidade significativo para escolher o modelo de processo de software que é melhor para o projeto, e as tarefas de engenharia de software que preenchem o modelo do processo, uma vez escolhido. Um projeto relativamente pequeno, semelhante a esforços anteriores, pode ser mais bem realizado usando a abordagem seqüencial linear. Se forem impostas restrições severas de tempo e o problema puder ser bastante compartimentalizado, o modelo RAD é provavelmente a opção correta. Se o prazo de entrega é tão apertado que praticamente impossibilite entregar toda a funcionalidade, uma estratégia incremental pode ser melhor. Analogamente, projetos com outras características (por exemplo, requisitos incertos, tecnologia de vanguarda, clientes difíceis, potencial de reuso significativo) vão levar à seleção de outros modelos de processo[5].

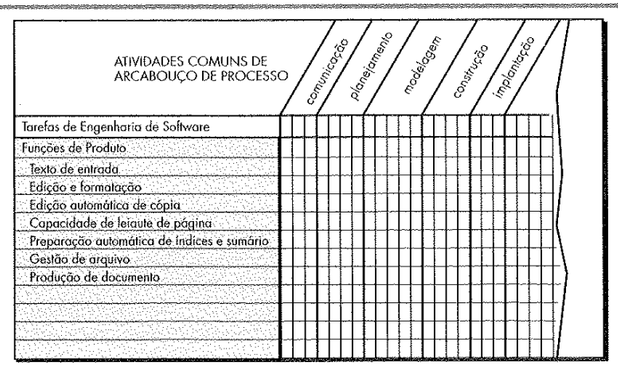
Uma vez escolhido o modelo de processo, o arcabouço de processo é adaptado a ele. Em qualquer caso, o arcabouço genérico discutido anteriormente — comunicações, planejamento, modelagem, construção e implantação — pode ser usado. Vai funcionar para modelos lineares, para modelos iterativos e incrementais, para modelos evolucionários e até para modelos concorrentes ou de montagem de componentes. O arcabouço de processo é invariante e serve como base para todo o trabalho de software desenvolvido por uma organização de software.

Mas as tarefas de trabalho reais variam. A decomposição do processo começa quando o gerente do projeto pergunta: "Como vamos realizar esta atividade de arcabouço?". Por exemplo, um projeto pequeno, relativamente simples, poderia requerer as seguintes tarefas de trabalho para a atividade comunicações:

1. Desenvolva uma lista de pontos a esclarecer.

FIGURA 21.1

Fusão do problema e do processo



4 Note-se que as tarefas de trabalho devem ser adaptadas às necessidades específicas do projeto.
5 Lembre-se de que as características do projeto também têm forte influência na estrutura da equipe de software (Seção 21.2.3).

2. Reúna-se com o cliente para discutir os pontos a esclarecer.
3. Desenvolva conjuntamente uma declaração de escopo.
4. Reveja a declaração de escopo com todos os interessados.
5. Modifique a declaração de escopo na medida do necessário.

Esses eventos podem ocorrer em um intervalo de até 48 horas. Eles representam uma decomposição de processo, que é apropriada para um projeto pequeno e relativamente simples.

Consideremos agora um projeto mais complexo, com escopo mais amplo e impacto mais significativo no negócio. Tal projeto poderia exigir as seguintes tarefas de trabalho para a atividade de comunicações:

1. Reveja a solicitação do cliente.
2. Planeje e marque um encontro facilitado e formal com o cliente.
3. Faça pesquisa para especificar a solução proposta e as abordagens existentes.
4. Prepare um "documento de trabalho" e uma agenda para a reunião formal.
5. Conduza a reunião.
6. Desenvolva, em conjunto, miniespecificações que reflitam as características do software quanto a dados, função e comportamento. Alternativamente, desenvolva casos de uso que descrevam o software do ponto de vista do usuário.
7. Reveja cada miniespecificação ou caso de uso quanto à correção, consistência e ausência de ambigüidade.
8. Monte as miniespecificações em um documento de definição de escopo.
9. Reveja o documento de definição de escopo ou coleção de casos de uso com todos os interessados.
10. Modifique o documento de definição de escopo ou casos de uso na medida do necessário.

Ambos os projetos desempenharam a atividade de arcabouço que chamamos "comunicações", mas a primeira equipe de projeto realizou 50% a mais das tarefas de trabalho de engenharia de software do que a segunda.

21.5 O Projeto

? Quais são os sinais de que um projeto de software está comprometido?

Para gerir com sucesso um projeto de software, precisamos entender o que pode dar errado (para que problemas possam ser evitados). Em um excelente trabalho sobre projetos de software, John Reel [REE99] define dez sinais que indicam que um projeto de sistema de informação está comprometido:

1. O pessoal de software não entende as necessidades de seus clientes.
2. O escopo do produto está maldefinido.
3. As modificações são mal gerenciadas.
4. A tecnologia escolhida sofre modificações.
5. As necessidades do negócio modificam-se [ou estão maldefinidas].
6. Os prazos são irreais.
7. Os usuários são resistentes.
8. O patrocínio é perdido [ou nunca foi obtido adequadamente].
9. A equipe de projeto não tem pessoal com as aptidões adequadas.
10. Gerentes [e profissionais] evitam as melhores práticas e as lições adquiridas.

Profissionais experientes da indústria se referem freqüentemente [meio facciosamente] à regra 90-90 quando discutem projetos de software particularmente difíceis: os primeiros 90% de um sistema absorvem 90% do esforço e do prazo alocados. Os últimos 10% gastam os outros 90% do esforço e prazo alocados [ZAH94]. Os princípios que levam à regra 90-90 estão contidos nos sinais anotados na lista precedente.

Document shared on www.docsity.com

> "Não temos tempo de parar para abastecer de combustível, já estamos atrasados."
>
> **M. Cleron**

Mas chega de negatividade! Como um gerente deve agir para evitar os problemas mencionados? Reel [REE99] sugere uma abordagem de bom senso, de cinco partes, para projetos de software:

1. *Comece com o pé direito.* Isso é conseguido trabalhando duro (muito duro) para entender o problema que deve ser resolvido e depois estabelecendo objetivos e expectativas realísticas para todos os que serão envolvidos no projeto. Isso é reforçado pela estruturação correta da equipe (Seção 21.2.3) e pela atribuição da autonomia, autoridade e tecnologia necessárias para conduzir o trabalho.
2. *Mantenha a energia de momento.* Muitos projetos partem de um bom princípio e depois lentamente se desintegram. Para conservar a energia de momento, o gerente de projeto deve providenciar incentivos para reduzir a rotatividade de pessoal ao mínimo absoluto, a equipe deve enfatizar a qualidade em toda tarefa que executa e a gerência sênior deve fazer tudo o que for possível para não ficar no caminho da equipe[6].
3. *Acompanhe o progresso.* Para um projeto de software, o progresso é acompanhado à medida que produtos do trabalho (por exemplo, modelos, código-fonte, conjuntos de casos de teste) são produzidos e aprovados (usando revisões técnicas formais), como parte da atividade de garantia de qualidade. Além disso, medidas do processo e do projeto de software (Capítulo 22) podem ser coletadas e usadas para avaliar o progresso comparadas com médias desenvolvidas para a organização de desenvolvimento de software.
4. *Tome decisões adequadas.* Essencialmente, as decisões do gerente de projeto e da equipe de software devem ser "manter a coisa simples". Sempre que possível decida usar software comercial de prateleira ou componentes de software existentes, decida evitar interfaces sob medida, quando abordagens padrão estão disponíveis, decida identificar e depois evitar riscos óbvios, e decida atribuir mais tempo do que você acha necessário para tarefas complexas ou arriscadas (você vai precisar de cada minuto).
5. *Faça uma análise a posteriori.* Estabeleça um mecanismo consistente para extrair as lições aprendidas com cada projeto. Avalie os cronogramas planejados e cumpridos, colete e analise métricas de projeto de software, obtenha realimentação dos membros da equipe e dos clientes, e registre as descobertas por escrito.

21.6 O Princípio W⁵HH

Em um excelente trabalho sobre processos e projetos de software, Barry Boehm [BOE96] afirma: "Você precisa de um princípio de organização que admita diminuição de escala para fornecer planos [de projeto] simples para projetos simples". Boehm sugere uma abordagem com foco nos objetivos de projeto, marcos e cronogramas, responsabilidades, abordagens gerenciais e técnicas, e recursos necessários. Ele a denomina o princípio W⁵HH, por causa de uma série de questões que levam à definição das características-chave do projeto e do plano de projeto resultante:

? Como definimos as características-chave do projeto?

Por que (Why) o sistema está sendo desenvolvido? A resposta a essa questão permite a todas as partes examinar a validade das razões comerciais para o trabalho de software. Dito de outra forma, a razão comercial justifica o gasto de pessoal, tempo e dinheiro?

O que (What) vai ser feito? A resposta a essa questão estabelece o conjunto de tarefas que vai ser necessário para o projeto.

Quando (When) vai ser feito? A resposta a essa questão ajuda a equipe a estabelecer um cronograma do projeto pela identificação de quando as tarefas do projeto devem ser realizadas e quando os marcos devem ser alcançados.

6 A implicação dessa declaração é a redução da burocracia ao mínimo, a eliminação de reuniões desnecessárias e a eliminação da obediência dogmática às regras do processo e do projeto. A equipe deve ser auto-organizada e autônoma.

Quem (Who) é responsável por uma função? Anteriormente, neste capítulo, discutiu-se que o papel e a responsabilidade de cada membro da equipe de software devem ser definidos. A resposta a essa questão ajuda a conseguir isso.

Onde (Where) estão localizados na organização? Nem todos os papéis e responsabilidades situam-se na própria equipe de desenvolvimento de software. O cliente, usuários e outros interessados também têm responsabilidades.

Como (How) o trabalho será conduzido técnica e gerencialmente? Uma vez estabelecido o escopo do produto, uma estratégia gerencial e técnica para o projeto deve ser definida.

Quanto (How much) é necessário de cada recurso? A resposta a essa questão é obtida pelo desenvolvimento de estimativas (Capítulo 23) baseadas nas respostas às questões anteriores.

O princípio W⁵HH, de Boehm, é aplicável independentemente do tamanho ou da complexidade de um projeto de software. As questões anotadas fornecem um excelente guia de planejamento para o gerente do projeto e para a equipe de software.

21.7 Práticas Críticas

O Airlie Council[7] desenvolveu uma lista de "práticas críticas de software para gerência baseada em desempenho". Essas práticas são "usadas e consideradas críticas consistentemente, por projetos e organizações de software altamente bem-sucedidos, cujo 'limite inferior' de desempenho é consistentemente muito melhor do que as médias da indústria" [AIR99].

Práticas críticas[8] incluem: gestão de projeto baseada em métricas (Capítulo 22), estimativa empírica de custo e prazo (Capítulos 23 e 24), acompanhamento do valor adquirido (Capítulo 24), gestão formal de risco (Capítulo 25), acompanhamento de defeitos em comparação com as metas de qualidade (Capítulo 26) e gestão consciente do pessoal (Seção 21.2). Cada uma dessas práticas críticas é tratada na Parte 4 deste livro.

Ferramentas de Software

Ferramentas de Software para Gestão de Projetos

As "ferramentas" listadas aqui são genéricas e aplicam-se a uma ampla gama de atividades realizadas por gerentes de projeto. Ferramentas específicas de gestão de projetos (por exemplo, ferramentas de cronogramação, ferramentas de estimação, ferramentas de análise de risco) são consideradas nos capítulos posteriores.

Ferramentas Representativas[9]

Software Program Manager's Network (*www.spmn.com*) desenvolveu uma ferramenta simples chamada *Project Control Panel* que fornece aos gerentes de projeto uma indicação direta do estado do projeto. A ferramenta tem "sensores" parecidos com um painel de controle e é implementada com Microsoft Excel, está disponível para *download* em *http://www.spmn.com/products_software.html*.

Ganthead.com tem desenvolvido um conjunto de lista de verificação (*checklist*) útil para gerentes de projeto, disponível para *download* em http://www.ganthead.com/.

Ittoolkit.com (*www.ittoolkit.com*) fornece "uma coleção de guias de planejamento, gabaritos de processo e planilhas inteligentes" disponíveis em CD-ROM.

7 O Airlie Council é uma equipe de especialistas em engenharia de software, contratada pelo Departamento de Defesa dos Estados Unidos, para ajudar a desenvolver diretrizes para melhor prática de gestão de projeto de software e de engenharia de software.

8 Apenas as práticas críticas associadas com a "integridade de projeto" são anotadas aqui.

9 As ferramentas mencionadas não representam uma recomendação, mas sim uma exemplificação de ferramentas dessa categoria. Na maioria dos casos os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

Document shared on www.docsity.com

21.8 Resumo

Gestão de projeto de software é uma atividade guarda-chuva dentro da engenharia de software. Começa antes de qualquer atividade técnica ser iniciada e continua ao longo da definição, do desenvolvimento e do apoio do software de computador.

Quatro Ps têm uma influência substancial na gestão de projetos de software — pessoal, produto, processo e projeto. O pessoal deve ser organizado em equipes efetivas, motivadas para fazer trabalho de software de alta qualidade e coordenadas para alcançar comunicação efetiva. Os requisitos do produto devem ser informados pelo cliente ao desenvolvedor, particionados (decompostos) em suas partes constituintes e posicionados para serem trabalhados pela equipe de software. O processo deve ser adaptado às pessoas e ao problema. Uma estrutura geral de processo é selecionada, um paradigma de engenharia de software adequado é aplicado e um conjunto de tarefas é escolhido para executar o serviço. Finalmente, o projeto deve ser organizado de modo que leve a equipe de software ao sucesso.

O elemento-chave em todos os projetos de software é o pessoal. Engenheiros de software podem ser organizados em várias e diferentes estruturas de equipe, que vão das tradicionais hierarquias de controle às equipes de "paradigma aberto". Diversas técnicas de coordenação e comunicação podem ser aplicadas para apoiar o trabalho da equipe. Em geral, revisões formais e comunicação informal, de pessoa a pessoa, têm maior valor para os profissionais.

A atividade de gerência de projeto inclui medidas e métricas, estimativa, análise de risco, cronogramas, acompanhamento e controle. Cada um desses tópicos será considerado nos capítulos seguintes.

Referências Bibliográficas

[AIR99] Airlie Council, "Performance Based Management: The Program Manager's Guide Based on the 16-Point Plan and Related Metrics", Draft Report, 18 mar. 1999.

[BAK72] Baker, F. T., "Chief Programmer Team Management of Production Programming", *IBM Systems Journal*, v. 11, n. 1, 1972, p. 56-73.

[BOE96] Boehm, B., "Anchoring the Software Process", *IEEE Software*, v. 13, n. 4, jul. 1996, p. 73-82.

[COC01] Cockburn, A. e Highsmith, J., "Agile Software Development: The People Factor", *IEEE Computer*, v. 34, n. 11, nov. 2001, p. 131-133.

[CON93] Constantine, L., "Work Organization: Paradigms for Project Management and Organization", *CACM*, v. 36, n. 10, out. 1993, p. 34-43.

[COU80] Cougar, J. e Zawacki, R. *Managing and Motivating Computer Personnel*, Wiley, 1980.

[CUR88] Curtis, B. *et al.*, "A Field Study of the Software Design Process for Large Systems", *IEEE Trans. Software Engineering*, v. SE-31, n. 11, nov. 1988, p. 1268-1287.

[CUR94] Curtis, B. *et al.*, *People Management Capability Maturity Model*, Software Engineering Institute, 1994.

[DEM98] DeMarco, T. e Lister, T., *Peopleware*, 2. ed., Dorset House, 1998.

[EDG95] Edgemon, J., "Right Stuff: How to Recognize It When Selecting a Projec. Manager", *Application Development Trends*, v. 2, n. 5, maio 1995, p. 37-42.

[FER98] Ferdinandi, P. L., "Facilitating Communication", *IEEE Software*, set. 1998, p. 92-96.

[JAC98] Jackman, M., "Homeopathic Remedies for Team Toxicity", *IEEE Software*, jul. 1998, p. 43-45.

[KRA95] Kraul, R. e Streeter, L., "Coordination in Software Development", *CACM*, v. 38, n. 3, mar. 1995, p. 69-81.

[MAN81] Mantei, M., "The Effect of Programming Team Structures on Programming Tasks", *CACM*, v. 24, n. 3, mar. 1981, p. 106-113.

[PAG85] Page-Jones, M., *Practical Project Management*, Dorset House, 1985, p. vii.

[REE99] Reel, J. S., "Critical Success Factors in Software Projects", *IEEE Software*, maio 1999, p. 18-23.

[WEI86] Weinberg, G., *On Becoming a Technical Leader*, Dorset House, 1986.

[WIT94] Whitaker, K., *Managing Software Maniacs*, Wiley, 1994.

[ZAH94] Zahniser, R., "Timeboxing for Top Team Performance", *Software Development*, mar. 1994, p. 35-38.

Problemas e Pontos a Considerar

21.1. Baseado na informação contida neste capítulo e na sua própria experiência, desenvolva "dez mandamentos" para fortalecer os engenheiros de software. Isto é, faça uma lista de dez diretrizes que proporcionarão ao pessoal de software trabalhar com todo o seu potencial.

21.2. O modelo de maturidade da capacidade de gestão de pessoal do Software Engineering Institute (PM-CMM) observa de forma organizada as "áreas de prática-chave", que produzem pessoal competente de software. Seu instrutor deve atribuir-lhe uma KPA para analisar e resumir.

21.3. Descreva três situações do mundo real nas quais o cliente e o usuário final são os mesmos. Descreva três situações nas quais eles são diferentes.

21.4. As decisões tomadas pelos gerentes seniores podem ter impacto significativo na efetividade de uma equipe de engenharia de software. Dê cinco exemplos para ilustrar que isso é verdade.

21.5. Leia o livro de Weinberg [WEI86] e redija um resumo de duas ou três páginas sobre os assuntos que devem ser considerados na aplicação do modelo MOI.

21.6. Você foi escolhido gerente de projeto em uma organização de sistemas de informação. Sua tarefa é construir uma aplicação bastante semelhante a outras que sua equipe já construiu, apesar de essa última ser maior e mais complexa. Os requisitos foram exaustivamente documentados pelo cliente. Que estrutura de equipe você escolheria e por quê? Que modelo de processo de software você escolheria e por quê?

21.7. Você foi nomeado gerente de projeto para uma pequena empresa de produtos de software. A tarefa consiste em construir um produto inédito, que combina hardware de realidade virtual com software no estado da arte. Como a concorrência no mercado de entretenimento em casa é intensa, há significativa pressão para que o serviço seja feito. Que estrutura de equipe você escolheria e por quê? Que modelo de processo de software você escolheria e por quê?

21.8. Você foi nomeado gerente de projeto para uma importante empresa de produtos de software. Sua tarefa é gerir o desenvolvimento da versão da próxima geração do software amplamente usado de processamento de textos. Como nova renda precisa ser gerada, prazos apertados foram estabelecidos e anunciados. Que estrutura de equipe você escolheria e por quê? Que modelo(s) de processo de software você escolheria e por quê?

21.9. Você foi nomeado gerente de processo de software para uma empresa que presta serviços no campo de engenharia genética. Sua tarefa é gerenciar o desenvolvimento de um novo produto de software que vai acelerar o passo da descoberta de tipos de genes. O trabalho é orientado para pesquisa e desenvolvimento, mas a meta é produzir um produto dentro do próximo ano. Que estrutura de equipe você escolheria e por quê? Que modelo(s) de processo de software você escolheria e por quê?

21.10. Foi solicitado a você que desenvolva uma pequena aplicação, que analisa cada curso oferecido por uma universidade e faz um relatório da nota média obtida no curso (para um determinado período). Redija uma declaração de escopo que delimite esse problema.

21.11. Faça uma decomposição funcional de primeiro nível da função de disposição (leiaute) de página discutida resumidamente na Seção 21.3.2.

Leituras e Fontes de Informação Adicionais

O livro *Guide to the Project Management Body of Knowledge*, do Project Management Institute, PMI, 2001, cobre todos os aspectos importantes de gerência de projeto. O livro *Project Management Best Practices for IT Professionals* de Murch, Prentice Hall, 2000, ensina habilidades básicas e fornece diretriz detalhada para todas as fases de um projeto de tecnologia da informação. O livro *Project Managers Desk Reference*, de Lewis, McGraw Hill, 1999, apresenta um processo com 16 passos para planejar, monitorar e controlar qualquer tipo de projeto. O livro *Professional Software Development*, de Connell, da Addison-Wesley, 2004, oferece sugestões pragmáticas para obter "cronogramas mais curtos, produtos de mais alta qualidade e projetos mais bem-sucedidos".

Uma excelente série de quatro volumes escrita por Weinberg (*Quality Software Management*, Dorset House, 1992, 1993, 1994, 1996) introduz raciocínio básico de sistemas e conceitos de gestão, explica como usar medidas efetivamente e trata da "ação congruente", da capacidade de "conciliar" as necessidades da gerência com as necessidades da equipe técnica e as necessidades do negócio. Fornece informações úteis tanto aos novos gerentes quanto aos mais experientes. Futrell e seus colegas (*Quality Software Project Management*, Prentice Hall, 2002) apresentam um tratamento volumoso da gestão de projeto.

Document shared on www.docsity.com

Philiphs (*IT Project Management: On Track from Start to Finish*, Mc-Graw-Hill/ Osborne, 2002), Charvat (*Project Management Nation*, Wiley, 2002), Schwalbe (*Information Technology Project Management*, segunda edição, Course Tecnhology, 2001) de Holtsnider e Jaffe (*IT Manager's Handbook*, Morgan Kaufmann Publishers, 2000) são representantes dos diversos livros que têm sido escritos sobre gestão de projeto de software. Brown e seus colegas (*AntiPatterns in Project Management*, Wiley, 2000) discutem o que não fazer durante a gestão de um projeto de software.

Brooks (*The Mythical Man-Month*, Anniversary Edition, Addison-Wesley, 1995) atualizou seu livro clássico para dar uma visão nova de tópicos de projeto e gestão de software. McConnell (*Software Project Survival Guide*, Microsoft Press, 1997) apresenta um excelente guia pragmático para aqueles que precisam gerir projetos de software. Purba e Shah (*How to Manage a Successful Software Project*, segunda edição, Wiley, 2000) apresentam vários estudos de caso que indicam por que alguns projetos são bem-sucedidos e outros falham. Bennatan (*On Time Within Budget*, terceira edição, Wiley, 2000) apresenta sugestões úteis e diretrizes para gerentes de projeto de sofwtare.

Pode-se dizer que o aspecto mais importante da gerência de projeto de software é a gerência de pessoal. Cockburn (*Agile Software Development*, Addison-Wesley, 2002) apresenta uma das melhores discussões de pessoal de software já escritas. O livro definitivo sobre esse assunto foi escrito por DeMarco e Lister [DEM98], mas, os seguintes livros sobre esse mesmo tema foram publicados recentemente e merecem ser examinados:

Beaudouin-Lafon, M., *Computer Supported Cooperative Work*, Wiley-Liss, 1999.

Carmel, E., *Global Software Teams: Collaborating Across Borders and Time Zones*, Prentice-Hall, 1999.

Constantine, L., *Peopleware Papers: Notes on the Hyman Side of Software*, Prentice-Hall, 2001.

Humphrey, W. S., *Managing Technical People: Innovation, Teamwork, and the Software Process*, Addison-Wesley, 1997.

Humphrey, W. S., *Introduction to the Team Software Process*, Addison-Wesley, 1999.

Jones, P. H., *Handbook of Team Design: A Practitioner's Guide to Team Systems Development*, McGraw-Hill, 1997.

Karolak, D. S., *Global Software Development: Managing Virtual Teams and Environments*, IEEE Computer Society, 1998.

Ensworth (*The Accidental Project Manager*, Wiley, 2001) fornece muitas diretrizes úteis para aqueles que precisam sobreviver "à transação entre técnico e gerente de projeto". Outro excelente livro de Weinberg [WEI86] é uma leitura obrigatória para todo gerente de projeto e para todo líder de equipe. Ele fornece visão e diretrizes detalhadas sobre como fazer seu serviço mais efetivamente.

Apesar de não estarem relacionados especificamente ao mundo do software e algumas vezes sofrerem de simplificações excessivas e generalizações demasiado amplas, os livros de grande volume de venda sobre "gerência", de Boosidy (*Execution: The Discipline of Getting Things Done*, Crown Publishing, 2002), Drucker (*Management Challenges for the 21st Century*, Harper Business, 1999), Buckingham e Coffman (*First, Break All the Rules: What the World's Greatest Managers Do Differently*, Simon and Schuster, 1999) e Christensen (*The Innovator's Dilemma*, Harvard Business School Press, 1997) enfatizam "novas regras" definidas por uma economia em rápida modificação. Títulos mais antigos tais como *Who Moved My Cheese?*, *The One-Minute Manager* e *In Search of Excellence* continuam a dar valiosas idéias que podem ajudá-lo a gerenciar mais eficientemente o pessoal.

Uma grande variedade de fontes de informação sobre gestão de projetos de software está disponível na Internet. Uma lista atualizada das referências da World Wide Web pode ser encontrada no site deste livro em: **http://www.mhhe.com/pressman**.

MÉTRICAS DE PROCESSO E PROJETO

CAPÍTULO 22

CONCEITOS-CHAVE

DRE 510
métricas
de caso de uso 507
de processo 500
de projeto 502
de qualidade 509
de WebApp 507
orientadas a tamanho .. 503
orientadas a função ... 504
orientadas a objetos ... 509
privadas 501
públicas 501
referências para métricas .. 512
programas de métricas ... 514
SSPI 502

A medição nos permite obter entendimento do processo e projeto, dando-nos um mecanismo para avaliação objetiva. Lord Kelvin certa vez disse:

> Você tem algum conhecimento sobre o que você fala quando pode medir e expressar isso em números; mas quando não pode medir, quando não pode expressar isso em números, seu conhecimento é fraco e insatisfatório: pode ser o princípio do conhecimento, mas você mal avançou em seus pensamentos para o estágio de uma ciência.

A comunidade de engenharia de software levou a sério as palavras de Lord Kelvin, mas não sem frustração e mais do que um pouco de controvérsia!

Medição pode ser aplicada ao processo de software com o objetivo de melhorá-lo de forma contínua. Pode ser usada ao longo de um projeto de software para auxiliar na estimativa, no controle de qualidade, na avaliação de produtividade e no controle do projeto. Finalmente, medição pode ser usada pelos engenheiros de software para ajudar a avaliar a qualidade dos produtos do trabalho e a auxiliar na tomada de decisões táticas, à medida que o projeto evolui (Capítulo 15).

Em seu livro de diretrizes sobre medição de software, Park, Goethert e Florac [PAR96] discutem as razões pelas quais nós medimos: (1) para *caracterizar* em um esforço a fim de obter entendimento de "processos, produtos, recursos e ambientes, e para estabelecer referências, para comparação com futuras avaliações"; (2) para *avaliar* "a fim de determinar o estado em relação aos planos"; (3) para *prever* pela "obtenção de entendimento de relacionamentos entre processos e produtos e construção de modelos desses relacionamentos"; (4) para *aperfeiçoar* pela "identificação de bloqueios, causas fundamentais, ineficiências e outras oportunidades, para melhorar a qualidade do produto e o desempenho do processo.

PANORAMA

O que é? Métricas de processo e de projeto de software são medidas quantitativas que permitem aos engenheiros de software ter idéia da eficácia do processo de software e dos projetos que são conduzidos usando o processo como arcabouço. Dados básicos de qualidade e de produtividade são coletados. Esses dados são então analisados, comparados com médias anteriores e avaliados para determinar se ocorreram melhorias de qualidade e produtividade. Métricas também são usadas para detectar áreas de problema, de modo que soluções possam ser desenvolvidas, e que o processo de software possa ser melhorado.

Quem faz? As métricas de software são analisadas e avaliadas por gerentes de software. Medidas são freqüentemente coletadas por engenheiros de software.

Por que é importante? Se você não faz medições, o julgamento pode ser baseado somente em avaliação subjetiva. Com medições, tendências (boas ou más) podem ser detectadas, melhores estimativas podem ser feitas e aperfeiçoamentos reais podem ser obtidos ao longo do tempo.

Quais são os passos? Comece definindo um conjunto limitado de medições de processo e projeto que são fáceis de coletar. Essas medições são freqüentemente normalizadas usando métricas orientadas a tamanho ou função. O resultado é analisado e comparado com médias anteriormente obtidas para projetos semelhantes desenvolvidos dentro da organização. Tendências são avaliadas e conclusões são geradas.

Qual é o produto do trabalho? Um conjunto de métricas de software que dá uma idéia do processo e entendimento do projeto.

Como tenho certeza de que fiz corretamente? Aplicando um esquema de medição consistente, mas simples, que nunca deve ser usado para avaliar, premiar ou punir desempenho individual.

Document shared on www.docsity.com

Medição é uma ferramenta de gestão. Se conduzida adequadamente, fornece conhecimento a um gerente de projeto. E, como resultado, apóia o gerente de projeto e a equipe de software na tomada de decisões que irão conduzir a um projeto de sucesso.

22.1 Métricas nos Domínios do Processo e do Projeto

PONTO CHAVE

Métricas de processo têm impacto no longo prazo. Sua intenção é melhorar o processo em si. Métricas de projeto freqüentemente contribuem para o desenvolvimento de métricas de processo.

Métricas de processo são coletadas no decorrer de todos os projetos e durante longos períodos. Seu objetivo é fornecer um conjunto de indicadores de processo que leva a aperfeiçoamentos do processo de software no longo prazo. *Métricas de projeto* permitem ao gerente de projeto de software (1) avaliar o estado de um projeto em andamento, (2) acompanhar riscos potenciais, (3) descobrir áreas-problema antes que elas se tornem "críticas", (4) ajustar fluxo de trabalho ou tarefas e (5) avaliar a capacidade da equipe de projeto de controlar a qualidade dos produtos do trabalho de software.

Medidas que são coletadas por uma equipe de projeto e convertidas em métricas, para uso durante um projeto, podem também ser transmitidas para aqueles que têm responsabilidade sobre o aperfeiçoamento do processo de software. Por essa razão, muitas métricas são usadas, tanto no domínio de processo quanto de projeto.

22.1.1 Métricas de Processo e Aperfeiçoamento do Processo de Software

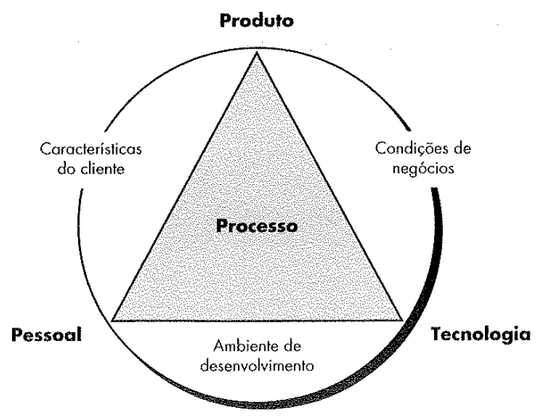
O único modo racional para aperfeiçoar qualquer processo é medir os atributos específicos, desenvolver um conjunto de métricas significativas, baseadas nesses atributos, e depois usar as métricas para fornecer indicadores que levarão a uma estratégia de aperfeiçoamento. Contudo, antes de discutirmos métricas de software e seu impacto na melhoria do processo de software, é importante notar que o processo é apenas um dos "fatores controláveis para melhoria da qualidade de software e do desempenho da organização"[PAU94].

Com referência à Figura 22.1, o processo situa-se no centro de um triângulo que liga três fatores com profunda influência na qualidade de software e no desempenho da organização. A capacidade e a motivação do pessoal foram evidenciadas [BOE81] como sendo o fator individual que mais influencia a qualidade e o desempenho. A complexidade do produto pode ter um impacto substancial sobre a qualidade e o desempenho da equipe. A tecnologia (por exemplo, os métodos e ferramentas de engenharia de software) que preenche o processo também tem impacto.

Além disso, o triângulo do processo encontra-se dentro de um círculo de condições ambientais, que incluem o ambiente de desenvolvimento (por exemplo, ferramentas CASE), condições de negócio (por exemplo, prazos, regras de negócio) e características do cliente (por exemplo, facilidade de comunicação).

FIGURA 22.1

Determinantes da qualidade de software e da efetividade organizacional (adaptados de [PAU94])



PONTO CHAVE

A capacidade e a motivação do pessoal que faz o trabalho são os fatores mais importantes que influenciam a qualidade do software.

Nós medimos a eficácia de um processo de software indiretamente, isto é, originamos um conjunto de métricas, baseadas nas saídas que podem ser derivadas do processo. As saídas incluem medidas dos erros descobertos antes da entrega do software, defeitos entregues aos usuários finais e por eles relatados, produtos de trabalho entregues (produtividade), esforço humano despendido, tempo gasto, cumprimento do cronograma e outras medidas. Também originamos métricas do processo, medindo características de tarefas específicas de engenharia de software. Por exemplo, poderíamos medir o esforço e o tempo gastos com a realização de atividades genéricas de engenharia de software descritas no Capítulo 2.

> "Métricas de software permitem que você saiba quando rir e quando chorar."
>
> Tom Gilb

Veja na Web

Qual é a diferença entre os usos privado e público para métricas de software?

Grady [GRA92] diz que há usos "públicos e privados" para diferentes tipos de dados do processo. Como é natural que engenheiros de software individualmente sejam sensíveis ao uso de métricas coletadas com base individual, esses dados devem ser privados e servir como indicador apenas para o indivíduo. Exemplos de *métricas privadas* incluem proporção de defeitos por indivíduo e por componente de software e erros encontrados durante o desenvolvimento.

A filosofia "dados privados de processo" está bem de acordo com a abordagem pessoal de processo de software (Capítulo 2) proposta por Humphrey [HUM95]. Ele reconhece que o aperfeiçoamento do processo de software pode e deve começar no nível individual. Dados privados de processo podem servir como direcionador importante, à medida que o engenheiro de software individual trabalha para se aperfeiçoar.

Algumas métricas de processo são privadas para a equipe de projeto de software, mas são públicas para todos os membros da equipe. Exemplos incluem defeitos encontrados para funções importantes do software (que foram desenvolvidas por um certo número de profissionais), erros encontrados durante revisões técnicas formais e linhas de código, ou pontos por função, por componente ou função[1]. Esses dados são revisados pela equipe para descobrir indicadores que podem melhorar o seu desempenho.

Métricas públicas geralmente assimilam informações que eram originalmente privadas, de indivíduos e de equipes. Proporções de defeitos de projeto (absolutamente não atribuíveis a um certo indivíduo), esforço, tempo transcorrido e dados relacionados são coletados e avaliados em uma tentativa de descobrir indicadores que possam aperfeiçoar o desempenho do processo organizacional.

As métricas de processo de software podem fornecer benefícios significativos, à medida que a organização trabalha para aperfeiçoar seu nível geral de maturidade de processo. Todavia, como todas as métricas, essas podem ser mal utilizadas, criando mais problemas do que conseguem resolver. Grady [GRA92] sugere uma "etiqueta de métricas de software", que é apropriada tanto para gerentes quanto para profissionais, quando eles instituem um programa de métricas de processo:

? Que diretrizes devem ser aplicadas quando coletamos métricas de software?

- Use bom senso e sensibilidade empresarial quando interpretar dados de métricas.
- Forneça regularmente realimentação aos indivíduos e equipes que coletam medidas e métricas.
- Não use métricas para avaliar indivíduos.
- Trabalhe com profissionais e equipes para estabelecer metas claras e métricas que devem ser usadas para alcançá-las.
- Nunca use métricas para ameaçar indivíduos ou equipes.
- Dados de métricas que indicam uma área problemática não devem ser considerados "negativos". Esses dados são meramente um indicador para melhoria do processo.
- Não fique obcecado com uma única métrica em detrimento de outras importantes.

1 Linhas de código e métricas de pontos por função são discutidas nas Seções 22.2.1 e 22.2.2.

Document shared on www.docsity.com

À medida que uma organização sente-se mais confortável, coletando e usando métricas de processo, a derivação de indicadores simples dá lugar a uma abordagem mais rigorosa, chamada de *melhoria estatística do processo de software* (*statistical software process improvement*, SSPI). Essencialmente, a SSPI usa a análise de falhas de software para coletar informação sobre todos os erros e defeitos[2] encontrados à medida que uma aplicação, sistema, ou produto é desenvolvido e usado.

22.1.2 Métricas de Projeto

Diferentemente de métricas de processo de software que são usadas com finalidade estratégica, métricas de projeto de software são táticas. Isto é, métricas de projeto e indicadores derivados delas são usados por um gerente de projeto e por uma equipe de software, para adaptar o fluxo de trabalho e as atividades técnicas do projeto.

A primeira aplicação das métricas de projeto, na maioria dos projetos de software, ocorre durante a estimativa. Métricas coletadas de projetos anteriores são usadas como base, a partir da qual estimativas de esforço e de tempo são feitas para o trabalho atual de software. Conforme o projeto prossegue, medidas de esforço e de tempo despendidos são comparadas com as estimativas originais (e o cronograma do projeto). O gerente do projeto usa esses dados para monitorar e controlar o progresso.

Quando o trabalho técnico inicia-se, outras métricas de projeto começam a ter importância. A taxa de produção, representada em termos de modelos criados, horas de revisão, pontos por função e linhas de código-fonte entregue, é medida. Além disso, os erros descobertos durante cada tarefa de engenharia de software são registrados. À medida que o software evolui, da especificação para o projeto, métricas técnicas (Capítulo 15) são coletadas para avaliar a qualidade do projeto e fornecer indicadores que irão influenciar a abordagem adotada para geração de código e teste.

? Como devemos usar métricas durante o projeto propriamente dito?

O objetivo das métricas de projeto é duplo. Primeiro, elas são usadas para minimizar o cronograma de desenvolvimento, fazendo os ajustes necessários para evitar atrasos e problemas, e riscos em potencial. Segundo, métricas de projetos são usadas para avaliar a qualidade do produto durante sua evolução e, quando necessário, modificar a abordagem técnica para aperfeiçoar a qualidade.

À medida que a qualidade é aperfeiçoada, os defeitos são minimizados e, conforme a contagem de defeitos decresce, a quantidade de retrabalho durante o projeto é também reduzida. Isso leva à diminuição do custo total do projeto.

CASASEGURA

Estabelecimento de uma Abordagem de Métricas

A cena: Escritório de Doug Miller quando o projeto de software do *CasaSegura* está prestes a começar.

Os personagens: Doug Miller (gerente da equipe de engenharia de software do *CasaSegura*) e Vinod Raman e Jamie Lazar, membros da equipe de produto de engenharia de software.

A conversa:

Doug: Antes de começarmos a trabalhar neste projeto, eu gostaria que vocês definissem e coletassem um conjunto de métricas simples. Para começar, vocês teriam que definir suas metas.

Vinod (carrancudo): Nós nunca fizemos isso antes, e...

Jamie (interrompendo): E com base no gerenciamento de cronograma sobre o qual temos conversado, nunca teremos tempo. De qualquer modo, para que servem métricas?

Doug (levantando a mão para interromper o ataque): Devagar e tomem fôlego, camaradas. O fato de que nunca fizemos isso antes é a maior razão para começarmos agora, e as métricas funcionam. E o trabalho de métricas de que estou falando não deve levar muito tempo... na verdade, ele pode nos poupar tempo.

Vinod: Como?

2 Neste livro, um *erro* é definido como alguma falha em um produto de trabalho de engenharia de software, que é encontrado antes de o software ser entregue ao usuário final. Um *defeito* é uma falha encontrada *depois* da entrega ao usuário final. Deve-se observar que outros não fazem essa distinção. Discussão adicional é apresentada no Capítulo 26.

Doug: Veja, estamos começando a fazer uma porção de coisas a mais de engenharia de software internamente, à medida que o nosso produto torna-se mais inteligente, disponível na Web, tudo isso... e precisamos entender o processo que usamos para construir software, e aperfeiçoá-lo de modo que possamos construir software melhor. O único modo de fazer isso é medir.

Jamie: Mas estamos pressionados pelo tempo, Doug. Eu não sou a favor de mais papelada... precisamos de tempo para fazer nosso trabalho, não para coletar dados.

Doug (calmamente): Jamie, um trabalho de engenheiro envolve coleta de dados, avaliação deles e uso dos resultados para melhorar o produto e o processo. Estou errado?

Jamie: Não, mas...

Doug: E se nós restringirmos o número de medições que coletarmos a não mais do que cinco ou seis, e enfocarmos a qualidade?

Vinod: Ninguém pode argumentar contra a alta qualidade...

Jamie: Verdade... mas, eu não sei, ainda penso que isso não é necessário.

Doug: Eu vou pedir a você para me esclarecer sobre isso. Quanto vocês sabem sobre métricas de software?

Jamie (olhando para Vinod): Não muito.

Doug: Aqui estão algumas referências da Web... gastem algumas poucas horas para começar a ganhar tempo.

Jamie (sorrindo): Eu pensei que você tivesse dito que isso não gastaria tempo nenhum.

Doug: Tempo que você gasta aprendendo nunca é desperdiçado... vá, faça-o e então estabeleceremos algumas metas, formularemos algumas questões e definiremos as métricas que precisamos coletar.

22.2 MEDIÇÃO DE SOFTWARE

No Capítulo 15, mencionamos que medição de software pode ser categorizada de dois modos: (1) *medidas diretas* do processo de engenharia de software (por exemplo, custo e esforço aplicados) e produto (por exemplo, linhas de código (*lines of code*, LOC) produzidas, velocidade de execução, tamanho de memória e defeitos relatados durante um certo período); e (2) *medidas indiretas* do produto, que incluem funcionalidade, qualidade, complexidade, eficiência, confiabilidade, manutenibilidade e muitas outras "-dades", discutidas no Capítulo 15.

> "Nem tudo que pode ser contado conta, e nem tudo que conta pode ser contado."
>
> **Albert Einstein**

Métricas de projeto podem ser consolidadas para criar métricas de processo que são públicas para todas as organizações de software. Mas como uma organização combina métricas provenientes de diferentes indivíduos ou projetos?

AVISO *Como muitos fatores influenciam o trabalho de software, não use métricas para comparar indivíduos ou equipes.*

Para ilustrar, consideremos um exemplo simples. Indivíduos em duas diferentes equipes de projeto registram e categorizam todos os erros que encontram durante o processo de software. Medidas individuais são então combinadas para desenvolver medidas da equipe. A equipe A encontrou 342 erros durante o processo de software, antes da entrega. A equipe B encontrou 184 erros. Sendo todas as outras coisas iguais, qual equipe é mais efetiva na descoberta de erros ao longo do processo? Como não sabemos o tamanho ou a complexidade dos projetos, não podemos responder a essa questão. Todavia, se as medidas são normalizadas, é possível criar métricas de software que permitam comparação com médias organizacionais mais amplas. Tanto métricas orientadas a tamanho quanto a função são normalizadas desse modo.

22.2.1 Métricas Orientadas a Tamanho

Métricas orientadas a tamanho são originadas pela normalização das medidas de qualidade e/ou produtividade, considerando o *tamanho* do software que foi produzido. Se uma organização de software mantém registros simples, uma tabela de medidas orientadas a tamanho, tal como a mostrada na Figura 22.2, pode ser criada. A tabela apresenta cada projeto de software desenvolvido, que foi completado nos últimos cinco anos, e as correspondentes medidas daqueles projetos.

Document shared on www.docsity.com

Com referência à entrada da tabela (Figura 22.2) para o projeto alfa: 12.100 linhas de código foram desenvolvidas, com 24 pessoas-mês de esforço, a um custo de 168.000 dólares. Deve-se notar que o esforço e o custo registrados na tabela referem-se a todas as atividades de engenharia de software (análise, projeto, código e teste), não apenas à codificação. Mais informações do projeto alfa indicam que 365 páginas de documentação foram desenvolvidas, 134 erros foram registrados antes que o software fosse entregue e 29 defeitos foram encontrados depois da entrega ao cliente, no primeiro ano de operação. Três pessoas trabalharam no desenvolvimento do software para o projeto alfa.

Para desenvolver métricas, que podem ser integradas com métricas semelhantes de outros projetos, escolhemos *linhas de código* como nosso valor de normalização. A partir dos dados rudimentares contidos na tabela, um conjunto de métricas simples orientadas a tamanho pode ser desenvolvido para cada projeto: erros por KLOC (milhares de linhas de código), defeitos por KLOC, $ por KLOC e páginas de documentação por KLOC. Adicionalmente, outras métricas interessantes podem ser calculadas: erros por pessoa-mês, KLOC por pessoa-mês, $ por página de documentação.

PONTO CHAVE

Métricas orientadas a tamanho são amplamente usadas, mas o debate sobre sua validade e aplicabilidade continua.

Métricas orientadas a tamanho não são universalmente aceitas como o melhor modo de medir o processo de desenvolvimento de software [JON86]. A maior parte da controvérsia gira em torno do uso de linhas de código como medidas-chave. Adeptos da medida LOC alegam que LOC é um "artefato" de todos os projetos de desenvolvimento de software que pode ser facilmente contado, que muitos modelos existentes de estimativa de software usam LOC ou KLOC como entrada-chave e que já existe um grande volume de literatura e de dados baseados em LOC. Por outro lado, os oponentes argumentam que as medidas de LOC são dependentes da linguagem de programação, que quando a produtividade é considerada, elas penalizam programas curtos, mas bem projetados, que não podem acomodar facilmente linguagens não procedimentais, e que seu uso na estimativa requer um nível de detalhes que pode ser difícil de alcançar (por exemplo, o planejador deve estimar o LOC a ser produzido, muito antes que a análise e o projeto tenham sido completados).

22.2.2 Métricas Orientadas à Função

Métricas de software orientadas à função usam uma medida da funcionalidade entregue pela aplicação como valor de normalização. A métrica orientada a função mais amplamente usada é a *pontos por função* (*function point* – FP). Cálculo dos pontos por função é baseado em características do domínio de informação e complexidade do software. O mecanismo de cálculo de FP foi discutido no Capítulo 15[3].

A medida pontos por função, como a medida LOC, é controversa. Proponentes afirmam que a FP é independente da linguagem de programação, tornando-a ideal para aplicações que usam linguagens convencionais e não procedimentais, e que é baseada em dados que são mais prováveis de ser conhecidos logo na evolução de um projeto, tornando a FP mais atrativa para uma abordagem

FIGURA 22.2

Métricas orientadas a tamanho

Projeto	LOC	Esforço	$(000)	Pág. doc.	Erros	Defeitos	Pessoas
alfa	12.100	24	168	365	134	29	3
beta	27.200	62	440	1224	321	86	5
gama	20.200	43	314	1050	256	64	6
•	•	•	•	•	•		
•	•	•	•	•	•		
•	•	•	•	•	•		

3 Veja a Seção 15.3.1 para uma discussão detalhada do cálculo de FP.

de estimativa. Oponentes alegam que o método requer alguma "mágica", pois o cálculo é baseado em dados subjetivos em vez de objetivos, que contagens no domínio da informação (e outras dimensões) podem ser difíceis de coletar posteriormente, e que a FP não tem significado físico direto – é apenas um número.

22.2.3 Reconciliação de Métricas LOC e FP

A relação entre linhas de código e pontos por função depende da linguagem de programação que é usada para implementar o software e da qualidade do projeto. Vários estudos tentaram relacionar as medidas FP e LOC. Para citar Albrecht e Gaffney [ALB83]:

> A tese deste trabalho é que a quantidade de funções a ser produzida por uma aplicação (programa) pode ser estimada pela itemização dos principais componentes[4] de dados a ser usados ou produzidos por ela. Além disso, essa estimativa de funções deve ser correlacionada com a quantidade de LOC a ser produzida e o esforço de desenvolvimento necessário.

A tabela seguinte[5] [JON98] fornece estimativas grosseiras do número médio de linhas de código necessárias para construir um ponto por função, em várias linguagens de programação:

Linguagem de Programação	LOC por Ponto por Função			
	Média	Mediana	Baixa	Alta
Access	35	38	15	47
Ada	154	—	104	205
APS	86	83	20	184
ASP 69	62	—	32	127
Assembler	337	315	91	694
C	162	109	33	704
C++	66	53	29	178
Clipper	38	39	27	70
COBOL	77	77	14	400
Cool:Gen/IEF	38	31	10	180
Culprit	51	—	—	—
DBase IV	52	—	—	—
Easytrieve+	33	34	25	41
Excel47	46	—	31	63
Focus	43	42	32	56
FORTRAN	—	—	—	—
FoxPro	32	35	25	35
Ideal	66	52	34	203
IEF/Cool:Gen	38	31	10	180
Informix	42	31	24	57
Java	63	53	77	—
JavaScript	58	63	42	75
JCL	91	123	26	150
JSP	59	—	—	—
Lotus Notes	21	22	15	25
Mantis	71	27	22	250
Mapper	118	81	16	245
Natural	60	52	22	141

4 É importante notar que "a itemização dos componentes principais" pode ser interpretada de diversos modos. Alguns engenheiros de software que trabalham em um ambiente de desenvolvimento orientado a objetos usam o número de classes ou objetos como a métrica de tamanho predominante. Uma organização de manutenção pode ver o tamanho do projeto em termos do número de ordens de alteração de engenharia (Capítulo 27). Uma organização de sistema de informação pode ver o número de processos do negócio afetados por uma aplicação.

5 Usada com permissão de Quantitative Software Management (www.qsm.com), copyright 2002.

Document shared on www.docsity.com

(continuação)

Linguagem de Programação	LOC por Ponto por Função			
	Média	Mediana	Baixa	Alta
Oracle	30	35	4	217
PeopleSoft	33	32	30	40
Perl	60	—	—	—
PL/1	78	67	22	263
Powerbuilder	32	31	11	105
REXX	67	—	—	—
RPG II/III	61	49	24	155
SAS	40	41	33	49
Smalltalk	26	19	10	55
SQL	40	37	7	110
VBScript36	34	27	50	—
Visual Basic	47	42	16	158

Uma revisão desses dados indica que uma LOC de C++ fornece aproximadamente 2,4 vezes (em média) a "funcionalidade" de uma LOC de C. Além disso, uma LOC de Smalltalk fornece no mínimo quatro vezes mais funcionalidade do que uma LOC de linguagem de programação convencional tal como Ada, COBOL ou C. Usando a informação contida na tabela, é possível "retroagir" [JON98] programas existentes para estimar o número de pontos por função, uma vez que o número total de comandos da linguagem de programação é conhecido.

Métricas baseadas em pontos por função e LOC têm sido consideradas relativamente precisas para prever o esforço e custo de desenvolvimento de software. Todavia, a fim de usar LOC e FP para estimativas (Capítulo 23), uma referência histórica de informação deve ser estabelecida.

No contexto de métricas de processo e projeto, estamos preocupados, principalmente, com produtividade e qualidade – medidas de "saída" do desenvolvimento de software como uma função de esforço e tempo aplicados e medidas da "aptidão para uso" dos produtos de trabalho que são produzidos. Para melhorar o processo e planejar os objetivos do projeto, nosso interesse é histórico. Qual foi a produtividade de desenvolvimento de software em projetos anteriores? Qual foi a qualidade do software produzido? Como os dados de produtividade e qualidade do passado podem ser extrapolados para o presente? Como isso pode nos ajudar a aperfeiçoar o processo e planejar novos projetos mais precisamente?

22.2.4 Métricas Orientadas a Objetos

Métricas de projeto de software convencional (LOC ou FP) podem ser usadas para estimar projetos de software orientados a objetos. No entanto, essas métricas não fornecem granularidade suficiente para os ajustes de cronograma e esforço que são necessários quando iteramos ao longo de um processo evolutivo ou incremental. Lorenz e Kidd [LOR94] sugerem o seguinte conjunto de métricas para projetos OO:

AVISO

Não é incomum vários scripts de cenários mencionarem a mesma funcionalidade ou objetos de dados. Assim, seja cuidadoso quando usar contadores de scripts.

Número de scripts de cenário (*number of scenario scripts*). Um script de cenário (análogo à discussão de caso de uso ao longo das Partes 2 e 3 deste livro) é uma seqüência de passos detalhados que descreve a interação entre o usuário e a aplicação. O número de scripts de cenário é diretamente proporcional ao tamanho da aplicação e ao número de casos de teste que precisa ser desenvolvido para exercitar o sistema depois que ele estiver construído.

Número de classes-chave (*number of key classes*). *Classes-chave* são os "componentes altamente independentes" [LOR94] que são definidos no início na análise orientada a objetos (Capítulo 8)[6]. Como as classes-chave são centrais ao domínio do problema, o número de tais classes

6 Classes-chave são referenciadas como *classes de análise* na Parte 2 deste livro.

AVISO

Classes podem variar em tamanho e complexidade. Assim, vale a pena considerar a classificação das contagens de classes por tamanho e complexidade.

é uma indicação da quantidade de esforço requerido para desenvolver o software e também uma indicação da quantidade potencial de reuso a ser aplicada durante o desenvolvimento do sistema.

Número de classes de apoio (*number of support classes*). *Classes de apoio* são necessárias para implementar o sistema, mas não são imediatamente relacionadas ao domínio do problema. Exemplos podem ser classes de IU, classes de acesso e manipulação a banco de dados, e classes de cálculo. Além disso, classes de apoio podem ser desenvolvidas para cada uma das classes-chave. O número de classes de apoio é um indicativo da quantidade de esforço requerida para desenvolver o software e uma indicação da quantidade potencial de reuso a ser aplicada durante o desenvolvimento do sistema.

Número médio de classes de apoio por classe-chave (*average number of support classes per key class*). Em geral, as classes-chave são conhecidas cedo no projeto. Classes de apoio são definidas durante o projeto. Se o número médio de classes de apoio por classe-chave for conhecido para um dado domínio de problema, estimar (baseado no número total de classes) seria muito simplificado. Lorenz e Kidd sugerem que aplicações com uma IGU têm entre duas e três vezes o número de classes de apoio em relação a classes-chave. Aplicações sem IGU têm entre uma e duas vezes o número de classes de apoio em relação a classes-chave.

Número de subsistemas (*number of subsystems*). Um *subsistema* é uma agregação de classes que apóia uma função que é visível ao usuário final de um sistema. Uma vez identificados os subsistemas, é mais fácil projetar um cronograma razoável no qual o trabalho dos subsistemas é subdividido entre a equipe de projeto.

Para serem usadas efetivamente em um ambiente de engenharia de software orientado a objetos, métricas análogas àquelas mencionadas anteriormente devem ser coletadas juntamente com medições de projeto tais como esforço gasto, erros e defeitos descobertos, e modelos ou páginas de documentação produzidos. À medida que o banco de dados cresce (depois que um certo número de projetos foi completado), relacionamentos entre medidas orientadas a objetos e medidas de projeto vão fornecer métricas que podem auxiliar na estimativa de projeto.

22.2.5 Métricas Orientadas a Casos de Uso

Poderia parecer razoável aplicar casos de uso[7] como uma medida de normalização análoga a LOC ou FP. Como FP, o caso de uso é definido logo no início do processo de software, permitindo que seja usado para estimativa antes que atividades significativas de modelagem e construção sejam iniciadas. Casos de uso descrevem (indiretamente, pelo menos) as funções e características visíveis ao usuário que são requisitos básicos de um sistema. O caso de uso é independente da linguagem de programação. Além disso, o número de casos de uso é diretamente proporcional ao tamanho da aplicação em LOC e ao número de casos de teste que terá de ser projetado para exercitar completamente a aplicação.

Como casos de uso podem ser criados em vários níveis de abstração, não há padronização quanto ao tamanho de um caso de uso. Sem uma medida padrão do que um caso de uso é, sua aplicação como uma medida de normalização (por exemplo, esforço gasto por caso de uso) é suspeita. Embora um certo número de pesquisadores (por exemplo, [SMI99]) tivesse tentado derivar métricas de casos de uso, muito trabalho permanece a ser feito.

22.2.6 Métricas de Projeto de Engenharia da Web

O objetivo de todos os projetos de engenharia da Web (Parte 3 deste livro) é construir uma aplicação de Web (WebApp) que entregue uma combinação de conteúdo e funcionalidade ao usuário final. Medidas e métricas usadas para projetos de engenharia de software tradicional são difíceis de traduzir diretamente para WebApps. No entanto, uma organização de engenharia da Web deve desenvolver um banco de dados que lhe permita avaliar sua produtividade e qualidade interna em relação a um certo número de projetos. Entre as medidas que podem ser coletadas estão:

Número de páginas estáticas da Web (*number of static Web pages*). Páginas da Web com conteúdo estático (isto é, o usuário final não tem controle sobre o conteúdo exibido na página) são

7 Casos de uso são discutidos ao longo das Partes 2 e 3 deste livro.

Document shared on www.docsity.com

as mais comuns de todas as características de WebApp. Essas páginas representam complexidade relativamente baixa e em geral necessitam de menor esforço para construção do que páginas dinâmicas. Essa medida fornece uma indicação do tamanho global da aplicação e do esforço necessário para desenvolvê-la.

Número de páginas dinâmicas da Web (*number of dymanic Web pages*). Páginas da Web com conteúdo dinâmico (isto é, as ações do usuário final resultam em exibição de conteúdo personalizado na página) são essenciais em todas as aplicações de e-commerce, motores de busca, aplicações financeiras, e muitas outras categorias de WebApp. Essas páginas representam complexidade relativamente alta e requerem mais esforço para a construção do que páginas estáticas. Essa medida fornece uma indicação do tamanho global da aplicação e do esforço necessário para desenvolvê-la.

Número de links internos da página (*number of internal page links*). Links internos da página são ponteiros que fornecem um hiperlink para alguma outra página da Web dentro da WebApp. Essa medida fornece uma indicação do grau de acoplamento arquitetural na WebApp. À medida que o número de links de páginas aumenta, o esforço gasto no projeto e na construção navegacional também aumenta.

Número de objetos de dados persistentes (*number of persistent data objects*). Uma WebApp pode ter acesso a um ou mais objetos de dados persistentes (por exemplo, um banco de dados ou arquivo de dados). Conforme o número de objetos de dados persistentes cresce, a complexidade da WebApp também cresce, e o esforço para implementá-la aumenta proporcionalmente.

Número de sistemas externos interfaceados (*number of external systems interfaced*). WebApps precisam freqüentemente ter interface com aplicações de negócio de "retaguarda". À medida que os requisitos de interfaceamento crescem, a complexidade e esforço de desenvolvimento do sistema também crescem.

Número de objetos de conteúdo estático (*number of static content objects*). Objetos de conteúdo estático englobam informação estática baseada em texto, gráfico, vídeo, animação e áudio e que é incorporada na WebApp. Vários objetos de conteúdo podem aparecer em uma única página da Web.

Número de objetos de conteúdo dinâmico (*number of dynamic content objects*). Objetos de conteúdo dinâmico são gerados com base em ações do usuário final e abrangem informação gerada internamente com base em texto, gráfica, vídeo, animação e áudio que são incorporadas na WebApp. Vários objetos de conteúdo podem aparecem em uma única página da Web.

Número de funções executáveis (*number of executable functions*). Uma função executável (por exemplo, um script ou *applet*) fornece algum serviço computacional ao usuário final. Conforme o número de funções executáveis aumenta, o esforço de modelagem e construção também aumenta.

Cada uma das medidas mencionadas deve poder ser determinada em um estágio relativamente inicial do processo de engenharia da Web.

Por exemplo, podemos definir uma métrica que reflete o grau de personalização ao usuário final que é requerido da WebApp e correlacioná-lo ao esforço gasto no projeto de WebE e/ou aos erros encontrados quando revisão e teste são conduzidos. Para conseguir isso definimos:

Nsp = número de páginas estáticas da Web
Ndp = número de páginas dinâmicas da Web

Então,

Índice de personalização, $C = N_{dp}/(N_{dp} + N_{sp})$

O valor de C varia de 0 a 1. À medida que C cresce, o nível de personalização da WebApp torna-se um tópico técnico significativo.

Métricas de aplicação da Web semelhantes podem ser calculadas e correlacionadas com medições de projeto tais como esforço gasto, erros e defeitos descobertos, e modelos ou páginas de documentação produzidos. Conforme o banco de dados cresce (depois que um certo número de projetos tiver sido completado), relacionamentos entre medições da WebApp e medições de projeto fornecerem indicadores que podem ajudar na estimativa do projeto.

FERRAMENTAS DE SOFTWARE

Métricas de Projeto e Processo

Objetivo: Para apoiar a definição, a coleta, a avaliação e o relato de medidas e métricas de software.

Mecânica: Cada ferramenta varia em sua aplicação, mas todas fornecem mecanismos para coletar e avaliar dados que levam ao cálculo de métricas de software.

Ferramentas Representativas[8]

Function Point WORKBENCH, desenvolvida por Charismatek (*www.charismatek.com.au*), oferece uma ampla gama de métricas orientadas a FP.

MetricCenter, desenvolvida por Distributive Software (www.distributive.com), apóia de forma automatizada a coleta, análise, formatação de gráfico, geração de relatório e outras tarefas de medição de dados.

PSM Insight, desenvolvida por Practical Software and Systems Measurement (*www.psmsc.com*), apóia a criação e subseqüente análise de um banco de dados de medições de projeto.

SLIM tool set, desenvolvida por QSM (*www.qsm.com*), fornece um conjunto abrangente de ferramentas de métricas e estimativas.

SPR tool set, desenvolvida por Software Productivity Research (*www.spr.com*), oferece uma coleção abrangente de ferramentas orientadas a FP.

TychoMetrics, desenvolvida por Predicate Logic, Inc. (*www.predicate.com*), é um conjunto de ferramentas para gestão de coleta e relato de métricas.

22.3 MÉTRICAS DE QUALIDADE DE SOFTWARE

A principal meta da engenharia de software é produzir um sistema, aplicação ou produto de alta qualidade em um espaço de tempo que satisfaça uma necessidade de mercado. Para atingir essa meta, engenheiros de software devem aplicar métodos efetivos combinados com modernas ferramentas no contexto de um processo de software amadurecido. Além disso, um bom engenheiro de software (e bons gerentes de engenharia de software) precisa medir se a alta qualidade deve ser alcançada.

Métricas privadas, colhidas por engenheiros de software individuais, são ajustadas para fornecer resultados no nível de projeto. Apesar de muitas medidas de qualidade poderem ser coletadas, a tendência principal no nível de projeto é medir erros e defeitos. Métricas derivadas dessas medidas fornecem indicação da efetividade da garantia de qualidade de software, individual e de grupo e das atividades de controle.

Métricas como erros de produto do trabalho (por exemplo, requisitos ou projeto) por ponto por função, erros descobertos por hora de revisão e erros descobertos por hora de teste são métricas que fornecem compreensão da eficácia de cada uma das atividades por elas determinadas. Dados de erros também podem ser usados para calcular a *eficiência de remoção de defeitos* (*defect removal efficiency*, DRE), para cada atividade de arcabouço do processo. A DRE é discutida na Seção 22.3.2.

Veja na Web

Uma excelente fonte de informação sobre qualidade de software e tópicos relacionados (inclusive métricas) pode ser encontrada em www.qualityworld.com.

22.3.1 Medição de Qualidade

Apesar de existirem várias medidas de qualidade de software[9], a correção, manutenibilidade, integridade e usabilidade fornecem indicadores úteis à equipe de software. Gilb [GIL88] sugere definições e medidas para cada uma.

Correção. Um programa precisa operar corretamente ou é de pouco valor para seus usuários. Correção é o grau em que o software desempenha sua necessária função. A medida mais comum de correção é defeitos por KLOC, em que um defeito é definido como uma falha verificada de obediência aos requisitos. Quando se considera a qualidade geral de um produto de software, defeitos

8 As ferramentas mencionadas aqui não representam uma recomendação, mas, em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

9 Uma discussão detalhada dos fatores que influenciam a qualidade de software e as métricas que podem ser usadas para avaliar qualidade de software é apresentada no Capítulo 15.

Document shared on www.docsity.com

são problemas relatados por um usuário do programa, após o programa ter sido entregue para uso geral. Para fins de avaliação de qualidade, defeitos são contados durante um período de tempo padrão, em geral um ano.

Manutenibilidade. Manutenção de software consome mais esforço do que qualquer atividade de engenharia de software. Manutenibilidade é a facilidade com que um programa pode ser corrigido, se um erro é encontrado, adaptado, se seu ambiente se modifica, ou é aperfeiçoado, se o cliente deseja uma modificação nos requisitos. Não há modo de medir manutenibilidade diretamente; assim, precisamos usar medidas indiretas. Uma métrica simples orientada a tempo é *tempo médio para modificação* (*mean-time-to-change*, MTTC), que é o tempo despendido para analisar o pedido de modificação, projetar uma modificação adequada, implementar a modificação, testá-la e distribuí-la para todos os usuários. Em média, programas manuteníveis terão um MTTC mais baixo (para tipos equivalentes de modificação) do que programas não-manuteníveis.

Integridade. A integridade de software tornou-se cada vez mais importante na era dos *hackers* e equipamentos *firewalls*. Este atributo mede a capacidade de o sistema resistir a ataques (tanto acidentais quanto intencionais) à sua segurança. Os ataques podem ser feitos aos três componentes do software: programas, dados e documentos.

Para medir a integridade, dois atributos adicionais precisam ser definidos: ameaça e segurança. *Ameaça* é a probabilidade (que pode ser estimada ou originada de evidência empírica) de um ataque de tipo específico ocorrer dentro de um certo período. *Segurança* é a probabilidade (que pode ser estimada ou originada de evidência empírica) de um ataque ser repelido. A integridade do sistema pode ser então definida como

$$\text{integridade} = \Sigma\,[1 - (\text{ameaça} \times (1 - \text{segurança}))]$$

Por exemplo, se a ameaça (a probabilidade que um ataque venha a ocorrer) é de 0,25 e a segurança (a probabilidade de repelir o ataque) é de 0,95, a integridade do sistema é de 0,99 (muito alta). Se por outro lado, a probabilidade de ameaça é de 0,50 e a probabilidade de repelir um ataque é de apenas 0,25, a integridade do sistema é de 0,63 (inaceitavelmente baixa).

Usabilidade. Se um programa não é fácil de usar, está freqüentemente fadado ao fracasso, mesmo se as funções que desempenha forem valiosas. Usabilidade é uma tentativa de quantificar facilidade de uso e pode ser medida em termos das características apresentadas no Capítulo 12. Os quatro fatores recém-descritos são apenas um exemplo dos que têm sido propostos como medida da qualidade de software. O Capítulo 15 considera este assunto em mais detalhes.

22.3.2 Eficiência na Remoção de Defeitos

Uma métrica de qualidade, que fornece benefício tanto no nível de projeto quanto de processo, é a eficiência na remoção de defeitos (*defect removal efficiency*, DRE). Em essência, DRE é uma medida da capacidade de filtragem das atividades de controle e garantia de qualidade de software, à medida que são aplicadas às atividades de arcabouço de processo.

Quando considerada para o projeto todo, a DRE é definida da seguinte maneira:

$$\text{DRE} = E/(E + D)$$

em que E é a quantidade de erros encontrados antes da entrega do software ao usuário final e D é a quantidade de defeitos encontrados após a entrega.

AVISO

Se a DRE é baixa durante a análise e o projeto, invista algum tempo aperfeiçoando o modo pelo qual você conduz revisões técnicas formais.

O valor ideal da DRE é 1, ou seja, nenhum defeito é encontrado no software. Na prática, D é maior que 0, mas o valor da DRE ainda pode ficar próximo de 1. À medida que E aumenta (para um dado valor de D), o valor global da DRE se aproxima de 1. De fato, à medida que E aumenta, é provável que o valor final de D diminua (os erros são filtrados antes que se transformem em defeitos). Se usada como métrica, que fornece indicador da capacidade de filtragem das atividades de garantia e controle de qualidade, a DRE motiva uma equipe de projeto de software a instituir técnicas para encontrar tantos erros quanto possível antes da entrega.

A DRE também pode ser usada no projeto para avaliar a capacidade de uma equipe encontrar erros antes que eles sejam passados para a próxima atividade de arcabouço ou tarefa de engenharia

Document shared on www.docsity.com

de software. Por exemplo, a tarefa de análise de requisitos produz um modelo de análise que pode ser revisado para encontrar e corrigir erros. Os erros que não são encontrados durante a revisão do modelo de análise são passados para a tarefa de projeto (em que eles podem ou não ser encontrados). Quando usada nesse contexto, redefinimos a DRE como:

$$\text{DRE}_i = E_i/(E_i + E_{i+1})$$

em que E_i é o número de erros encontrados durante a atividade i de engenharia de software, e E_{i+1} é a quantidade de erros encontrados durante a atividade de engenharia de software $i+1$, que são atribuíveis a erros que não foram descobertos na atividade de engenharia de software i.

Um objetivo de qualidade para uma equipe de software (ou para um engenheiro de software individualmente) é conseguir que a DRE_i fique próxima de 1. Isto é, os erros devem ser filtrados antes que passem para a próxima atividade.

CASASEGURA

Estabelecimento de uma Abordagem de Métricas

A cena: Escritório de Doug Miller dois dias depois da reunião inicial sobre métricas de software.

Os personagens: Doug Miller (gerente da equipe de engenharia de software do *CasaSegura*) e Vinod Raman e Jamie Lazar, membros da equipe de engenharia de software do produto.

A conversa:

Doug: Vocês dois tiveram a chance de aprender um pouco sobre métricas de processo e projeto?

Vinod e Jamie: [Os dois balançam a cabeça afirmativamente.]

Doug: É sempre uma boa idéia estabelecer metas quando você adota qualquer métrica. Quais são as suas?

Vinod: Nossa métrica deverá enfocar a qualidade. Na realidade, nossa meta global é manter o número de erros, que passamos de uma atividade de engenharia de software para a seguinte, a um mínimo absoluto.

Doug: E fique muito seguro de que você manteve o número de defeitos entregues com o produto tão perto de zero quanto possível.

Vinod (balançando a cabeça afirmativamente): Está certo.

Jamie: Eu gosto de DRE como métrica e acho que podemos usá-la para o projeto todo. Também podemos usá-la quando movermos de uma atividade de arcabouço para a seguinte. Isso vai nos encorajar a encontrar erros a cada passo.

Vinod: Eu também gostaria de coletar o número de horas que gastamos em revisões.

Jamie: E o esforço global gasto em cada tarefa de engenharia de software.

Doug: Você pode calcular a taxa de revisão por desenvolvimento... pode ser interessante.

Jamie: Eu gostaria também de rastrear alguns dados de caso de uso. Como a quantidade de esforço necessária para desenvolver um caso de uso, a quantidade de esforço necessária para construir software, para implementar um caso de uso, e...

Doug (rindo): Eu pensei que nós íamos manter isso simples.

Vinod: Deveríamos, mas quando se entra nesse negócio de métricas, há uma porção de coisas interessantes a observar.

Doug: Eu concordo, mas vamos caminhar antes de correr, e nos ater a nossa meta. Limite os dados a ser coletados a cinco ou seis itens, e estamos prontos para prosseguir.

22.4 INTEGRAÇÃO DE MÉTRICAS NO PROCESSO DE SOFTWARE

A maioria dos desenvolvedores de software ainda não mede e, infelizmente, muitos têm pouca vontade de começar. Como observamos anteriormente neste capítulo, o problema é cultural. Tentativas de coletar medidas, em que nenhuma era anteriormente coletada, freqüentemente causam resistência. "Por que precisamos fazer isso?", pergunta um atarefado gerente de projeto. "Não entendo a razão", reclama um profissional sobrecarregado.

Nesta seção, consideramos alguns argumentos a favor das métricas de software e apresentamos uma abordagem para instituir um programa para coletar métricas em uma organização de engenharia de software. Mas, antes de começarmos, algumas palavras sábias são sugeridas por Grady e Caswell [GRA87]:

> Algumas das coisas que descrevemos aqui parecerão muito fáceis. Todavia, na prática, estabelecer um programa de métricas bem-sucedido em toda a empresa é um trabalho duro. Quando dizemos que é necessário esperar pelo menos três anos até que tendências abrangentes da organização se tornem disponíveis, você pode ter uma idéia da grandeza desse esforço.

O conselho dado pelos autores merece ser seguido, mas os benefícios da medição são tão atraentes que o trabalho duro vale a pena.

22.4.1 Argumentos Favoráveis a Métricas de Software

Por que é tão importante medir o processo de engenharia de software e o produto (software) que ele produz? A resposta é relativamente óbvia. Se não medirmos, não haverá forma real de determinar se estamos melhorando. E se não estivermos melhorando, estaremos perdidos.

Pela solicitação e avaliação de medidas de produtividade e qualidade, uma equipe de software (e os seus gerentes) pode estabelecer metas significativas para aperfeiçoamento do processo de engenharia de software. No Capítulo 1, observamos que software é um assunto estratégico do negócio para muitas empresas. Se o processo pelo qual é desenvolvido puder ser aperfeiçoado, poderá resultar em impacto direto na base. Mas, para estabelecer metas para aperfeiçoamento, o estado atual do desenvolvimento de software deve ser entendido. Assim, a medição é usada para estabelecer um referencial do processo a partir do qual melhorias podem ser avaliadas.

> "Gerimos as coisas 'por números' em muitos aspectos de nossas vidas... Esses números nos dão compreensão e ajudam a dirigir nossas ações."
>
> **Michael Mah e Larry Putnam**

O rigor do dia-a-dia do trabalho de projeto de software deixa pouco tempo para raciocínio estratégico. Gerentes de projeto de software estão preocupados com assuntos mais comuns (mas igualmente importantes): desenvolver estimativas significativas de projeto, produzir sistemas de alta qualidade e entregar produtos dentro do prazo. Usando-se a medição para estabelecer um referencial do projeto, cada um desses aspectos torna-se mais gerenciável. Já observamos que um referencial serve como base para estimativas. Além disso, coletar métricas de qualidade permite à organização "sintonizar" seu processo de software para remover as "poucas, mas vitais" causas de defeitos, que têm maior impacto no desenvolvimento de software[10].

? O que é uma referência para métricas e qual o benefício que ela fornece a um engenheiro de software?

22.4.2 Estabelecimento de uma Referência

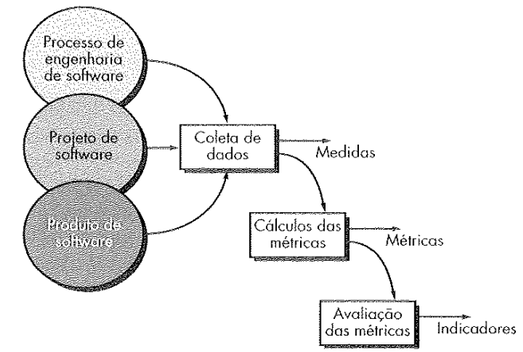
Pelo estabelecimento de uma referência para as métricas, os benefícios podem ser obtidos nos níveis (técnicos) de processo, projeto e produto. Mesmo assim, a informação que é coletada não precisa ser fundamentalmente diferente. As mesmas métricas podem servir a muitos mestres. A referência para as métricas consiste em dados coletados de projetos anteriores de desenvolvimento de software e pode ser tão simples como a tabela apresentada na Figura 22.2, ou tão complexa como um abrangente banco de dados contendo dezenas de medidas de projeto e as métricas delas derivadas.

Para constituir uma ajuda efetiva na melhoria do processo e/ou na estimativa de custo e esforço, os dados referenciais devem ter os seguintes atributos: (1) os dados precisam ser razoavelmente precisos — "estimativas", sem base sólida, relativas a projetos anteriores, devem ser evitadas; (2) dados devem ser coletados por tantos projetos quanto possível; (3) as medições, por exemplo, devem ser consistentes, uma linha de código deve ser interpretada uniformemente ao longo de todos os projetos para os quais os dados são coletados; (4) as aplicações devem ser análogas ao trabalho que precisa ser estimado — faz pouco sentido usar uma referência para um trabalho de sistemas de informação com processamento por lotes (*batch*), para estimar uma aplicação embutida de tempo real.

10 Essas idéias são formalizadas em uma abordagem chamada de *garantia estatística de qualidade de software* e são discutidas em detalhes no Capítulo 26.

FIGURA 22.3

Processo para coletar métricas de software



22.4.3 Coleta, Cálculo e Avaliação de Métricas

O processo para estabelecer uma referência está ilustrado na Figura 22.3. Seria ideal que os dados necessários tivessem sido coletados de modo progressivo. Infelizmente, esse caso é raro. Assim, coletar dados exige uma investigação histórica de projetos anteriores para reconstruir os dados necessários. Uma vez coletadas essas medidas (inquestionavelmente o passo mais difícil), o cálculo das métricas é possível. Dependendo da amplitude das medidas coletadas, as métricas podem abranger um amplo intervalo de métricas orientadas a aplicação (por exemplo, LOC, FP, orientadas a objetos, de WebApp), bem como outras métricas de qualidade e orientadas a projeto. Finalmente, as métricas precisam ser avaliadas e aplicadas durante a estimativa, o trabalho técnico, o controle do projeto e a melhoria do processo. A avaliação de métricas focaliza as razões subjacentes dos resultados obtidos e produz um conjunto de indicadores que dão diretrizes ao projeto ou processo.

PONTO CHAVE

Dados de métricas referenciais devem ser coletados para uma amostra grande, representativa de projetos de software anteriores.

22.5 MÉTRICAS PARA PEQUENAS ORGANIZAÇÕES

AVISO

Se você está apenas começando a coletar dados de métricas, lembre-se de mantê-las simples. Se você se enterra em dados, seus esforços de métricas falharão.

? Como deveríamos derivar um conjunto de métricas de software "simples"?

A grande maioria das organizações de desenvolvimento de software tem menos de 20 pessoas de software. É irracional, e na maior parte dos casos inviável, esperar que tais organizações desenvolvam programas abrangentes de métricas de software. Todavia, é razoável sugerir que organizações de software de todos os tamanhos meçam e depois usem as métricas resultantes para ajudar a aperfeiçoar seu processo local de software, e a qualidade e pontualidade dos produtos que produzem.

Uma abordagem de bom senso para a implementação de qualquer atividade relacionada a processo de software seria: torne simples, ajuste às necessidades locais e certifique-se de que tem valor. Nos parágrafos que se seguem, examinamos como essas diretrizes se relacionam a métricas para pequenas empresas[11].

"Torne simples" é uma diretriz que funciona razoavelmente bem em muitas atividades. Contudo, como originamos um conjunto de métricas de software "simples", que ainda tenha valor, e como podemos estar certos de que essas métricas simples vão satisfazer às necessidades de uma organização de software específica? Começamos focalizando não a medição, mas os resultados. O grupo de software é interrogado para definir um único objetivo que requer aperfeiçoamento. Por exemplo, "reduza o prazo para avaliar e implementar pedidos de modificação". Uma pequena organização pode selecionar o seguinte conjunto de medidas facilmente coletáveis:

- Tempo (horas ou dias) transcorrido entre o momento em que o pedido é feito até que a avaliação seja completada, t_{pta}.
- Esforço (pessoa-horas) para realizar a avaliação, W_{avaf}.

11 Essa discussão é igualmente relevante para as equipes de software que adotaram um processo ágil de desenvolvimento de software (Capítulo 4).

Document shared on www.docsity.com

- Tempo (horas ou dias) transcorrido desde o término da avaliação até a atribuição da ordem de modificação ao pessoal, t_{aval}.
- Esforço (pessoa-horas) necessário para fazer a modificação, $W_{modificação}$.
- Tempo necessário (horas ou dias) para fazer a modificação, $t_{modificação}$.
- Erros descobertos durante o trabalho para fazer a modificação, $E_{modificação}$.
- Defeitos descobertos depois que a modificação é entregue ao cliente, $D_{modificação}$.

Uma vez coletadas essas medidas, para um certo número de pedidos de modificação, é possível calcular o tempo total transcorrido desde o pedido até a implementação da modificação e a porcentagem do tempo transcorrido, absorvida pelo escalonamento inicial, avaliação, atribuição e implementação da modificação. Analogamente, a porcentagem do esforço necessário para avaliação e implementação pode ser determinada. Essas métricas podem ser avaliadas no contexto dos dados de qualidade, $E_{modificação}$ e $D_{modificação}$. As porcentagens permitem discernir onde o processo do pedido de modificação sofre demora, e podem levar a passos de melhoria do processo para reduzir t_{fila}, W_{aval}, t_{aval}, $W_{modificação}$ e/ou $E_{modificação}$. Além disso, a eficiência na remoção de defeitos pode ser calculada como:

$$DRE = Emodificação/(Emodificação + Dmodificação)$$

DRE pode ser comparada com o tempo transcorrido e o esforço total para determinar o impacto das atividades de garantia de qualidade pelo tempo e o esforço necessários para fazer uma modificação.

22.6 Estabelecimento de um Programa de Métricas de Software

O Software Engineering Institute desenvolveu um manual de diretrizes abrangente [PAR96] para estabelecer um programa de métricas de software "orientado a metas", que sugere os seguintes passos:

Veja na Web

O manual de diretrizes para medição de software orientada a metas (*Guidebook for Goal-Driven Software Measurement*) pode ser obtido no endereço www.sei.cmu.edu.

1. Identifique suas metas de negócio.
2. Identifique o que você deseja saber ou aprender.
3. Identifique suas submetas.
4. Identifique as entidades e atributos relacionados a suas submetas.
5. Formalize suas metas de medição.
6. Identifique questões quantificáveis e os indicadores correlacionados que você vai usar para ajudá-lo a atingir suas metas de medição.
7. Identifique os elementos de dados que você vai coletar para construir os indicadores que ajudam a responder a suas perguntas.
8. Defina as medidas a ser usadas e torne essas definições operacionais.
9. Identifique as ações que você executará para implementar as medidas.
10. Prepare um plano para implementar as medidas.

PONTO CHAVE

As métricas de software que você escolhe são dirigidas pelas metas de negócio ou técnicas que deseja atingir.

Deixamos a discussão detalhada desses passos para o livro de diretrizes do SEI. No entanto, uma visão geral dos pontos-chave é aconselhável.

Como o software apóia funções do negócio, diferencia sistemas ou produtos baseados em computador ou age como um produto em si mesmo, as metas definidas para o negócio podem, quase sempre, ser rastreadas até metas específicas de engenharia de software. Por exemplo, considere uma empresa que produz sistemas avançados de segurança residencial, que tem substancial conteúdo de software. Trabalhando em equipe, gerentes de engenharia de software e de negócio podem desenvolver uma lista priorizada de metas de negócio:

1. Aperfeiçoar a satisfação de nossos clientes com nossos produtos.
2. Tornar nossos produtos mais fáceis de usar.
3. Reduzir o tempo para colocação de um novo produto no mercado.

4. Facilitar o apoio aos nossos produtos.
5. Aperfeiçoar nossa lucratividade global.

A organização de software examina cada meta de negócio e pergunta: Que atividades nós gerimos ou executamos e o que desejamos aperfeiçoar nessas atividades? Para responder a essas questões o SEI recomenda a criação de uma "lista de questões-entidade" na qual todas as coisas (entidades) dentro do processo de software, que são geridas ou influenciadas pela organização de software, são anotadas. Exemplos de entidades incluem recursos de desenvolvimento, produtos de trabalho, código-fonte, casos de teste, pedidos de modificação, tarefas de engenharia de software e cronogramas. Para cada entidade relacionada, o pessoal de software desenvolve um conjunto de perguntas para avaliar as características quantitativas da entidade (por exemplo, tamanho, custo, prazo para desenvolver). As questões originadas em conseqüência da criação da lista de questões-entidades levam à origem de um conjunto de submetas, que se relacionam diretamente com as entidades criadas e as atividades desenvolvidas, como parte do processo de software.

Considere a quarta meta: "Facilitar o apoio aos nossos produtos". A seguinte lista de perguntas pode ser originada dessa meta [PAR96]:

- Os pedidos de modificação do cliente contêm a informação de que precisamos para avaliar adequadamente a modificação e depois implementá-la a tempo?
- Qual o tamanho da lista de pedidos de modificação pendentes?
- Nosso tempo de resposta para corrigir defeitos é aceitável em relação às necessidades do cliente?
- Nosso processo de controle de modificação (Capítulo 27) está sendo seguido?
- As modificações com alta prioridade são implementadas a tempo?

Com base nessas perguntas, a organização de software pode criar a seguinte submeta: *aperfeiçoar o desempenho da gestão do processo de modificação*. As entidades e atributos do processo de software, que são relevantes para a submeta, são identificados, e metas de medição associadas a eles são delineadas.

O SEI [PAR96] oferece diretrizes detalhadas para os passos 6 a 10 de sua abordagem de medição orientada a metas. Em essência, um processo de refinamento passo a passo é aplicado, no qual as metas são refinadas até chegar a perguntas, que são então refinadas até chegar a entidades e atributos, que são depois refinados até chegar a métricas.

INFO

Estabelecimento de um Programa de Métricas

O Centro de Produtividade de Software (*Software Productivity Center*) (www.spc.ca) sugere uma abordagem com oito passos para estabelecer um programa de métricas em uma organização de software, que pode ser usado como uma alternativa para a abordagem do SEI descrita na Seção 22.6. Sua abordagem é resumida nesta moldura.

1. Entenda o processo de software existente.
   - As atividades de arcabouço (Capítulo 2) são identificadas.
   - Informação de entrada para cada atividade é descrita.
   - Tarefas associadas com cada atividade são definidas.
   - Funções de garantia de qualidade são observadas.
   - Produtos de trabalho produzidos são listados.
2. Defina as metas para atingir o estabelecimento de um programa de métricas.
   - Exemplos: melhore a precisão de estimativa, melhore a qualidade do produto.
3. Identifique as métricas necessárias para atingir as metas.
   - Questões a ser respondidas são definidas, por exemplo, quantos erros encontrados em uma atividade de arcabouço podem ser rastreados até a atividade de arcabouço anterior?
   - Crie medidas e métricas a ser coletadas e calculadas.
4. Identifique as medidas e métricas a ser coletadas e calculadas.

Document shared on www.docsity.com

5. Estabeleça um processo de coleta de medidas respondendo a essas questões:
   - Qual é a fonte das medições?
   - Ferramentas podem ser usadas para coletar os dados?
   - Quem é responsável por coletar os dados?
   - Quando os dados são coletados e registrados?
   - Como os dados são armazenados?
   - Que mecanismos de validação são usados para garantir que os dados estejam corretos?
6. Adquira ferramentas adequadas para apoiar a coleta e avaliação.
7. Estabeleça um banco de dados de métricas.
   - A sofisticação relativa do banco de dados é estabelecida.
   - O uso de ferramentas relacionadas (por exemplo, um repositório SCM, Capítulo 27) é explorado.
   - Produtos de banco de dados existentes são avaliados.
8. Defina mecanismos de realimentação adequados.
   - Quem solicita informação contínua sobre métricas?
   - Como a informação deve ser entregue?
   - Qual é o formato da informação?

Uma descrição consideravelmente mais detalhada desses oito passos pode ser obtida em *http://www. spc.ca/ resources/metrics/*.

22.7 Resumo

Medição permite a gerentes e profissionais melhorar e aperfeiçoar o processo de software; colaborar no planejamento, acompanhamento e controle de um projeto de software; e avaliar a qualidade do produto (software) que é produzido. Medidas de atributos específicos do processo, projeto e produto são usadas para calcular métricas de software. Essas métricas podem ser analisadas para obter indicadores que orientam as ações gerenciais e técnicas.

Métricas de processo permitem que uma organização obtenha visão estratégica, fornecendo compreensão da efetividade do processo de software. Métricas de projeto são táticas. Elas permitem que um gerente de projeto adapte o fluxo de trabalho e a abordagem técnica do projeto em tempo real.

Tanto métricas orientadas a tamanho quanto a função são usadas por toda a indústria. Métricas orientadas a tamanho usam linha de código como fator de normalização para outras medidas, tais como pessoa-mês ou defeitos. Pontos por função é originada de medidas do domínio de informação e de uma avaliação subjetiva da complexidade do problema. Além disso, métricas orientadas a objetos e métricas de aplicação da Web podem ser usadas.

Métricas de qualidade de software, assim como métricas de produtividade, focalizam o processo, projeto e produto. Desenvolvendo e analisando referências para métricas de qualidade, uma organização pode corrigir as áreas do processo de software, que são a causa dos defeitos de software.

Medição resulta em mudança cultural. Coletar dados, calcular e analisar métricas são três passos que devem ser implementados para iniciar um programa de métricas. Em geral, uma abordagem orientada a metas ajuda a organização a focalizar as métricas adequadas para o negócio. Criando uma referência para métricas — um banco de dados que contém medições de produto e processo — engenheiros de software e seus gerentes podem ter melhor compreensão do trabalho e do produto.

Referências Bibliográficas

[ALB83] Albrecht, A. J. e Gaffney, J. E., "Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation", *IEEE Trans. Software Engineering*, nov. 1983, p. 639-648.

[BOE81] Boehm, B., *Software Engineering Economics*, Prentice-Hall, 1981.

[GRA87] Grady, R. B. e Caswell, D. L., *Software Metrics: Establishing a Company-Wide Program*, Prentice-Hall, 1987.

[GRA92] Grady, R. G., *Practical Software Metrics for Project Management and Process Improvement*, Prentice-Hall, 1992.

[GIL88] Gilb, T., *Principles of Software Project Management*, Addison-Wesley, 1988.

[HET93] Hetzel, W., *Making Software Measurement Work*, QED Publishing Group, 1993.

[HUM95] Humphrey, W., *A Discipline for Software Engineering*, Addison-Wesley, 1995.

[IEE93] *IEEE Software Engineering Standards*, Standard 610.12-1990, p. 47-48.

[JON86] Jones, C., *Programming Productivity*, McGraw-Hill, 1986.

[JON91] _____, *Applied Software Measurement*, McGraw-Hill, 1991.

[JON98] _____, *Estimating Software Costs*, McGraw-Hill, 1998.

[LOR94] Lorenz, M. e Kidd, J., *Object-Oriented Software Metrics*, Prentice-Hall, 1994.

[MCC78] McCall, J. A. e Cavano, J. R., "A Framework for the Measurement of Software.

[PAR96] Park, R. E., Goethert W. B. e Florac, W. A., *Goal Driven Software Measurement — A Guidebook*, CMU/SEI-96-BH-002, Software Engineering Institute, Carnegie Mellon University, ago. 1996.

[PAU94] Paulish, D. e Carleton, A., "Case Studies of Software Process Improvement Measurement", *Computer*, v. 27, n. 9, set. 1994, p. 50-57.

[QSM02] "QSM Function Point Language Gearing Factors", Versão 2.0, Quantitative Software Management, 2002, disponível em http://www.qsm.com/FPGearing.html.

[RAG95] Ragland, B., "Measure, Metric or Indicator: What's the Difference?", *Crosstalk*, v. 8, n. 3, mar. 1995, p. 29-30.

[SMI99] Smith, J., "The Estimation of Effort Based on Use-Cases", a white paper by Rational Corporation, 1999, disponível em http://www.rational.com/products/rup/whitepapers.jsp.

Problemas e Pontos a Considerar

22.1. Descreva a diferença entre métricas de processo e de projeto com suas próprias palavras.

22.2. Por que algumas métricas de software precisam ser mantidas "privadas"? Dê exemplos de três métricas que devem ser privadas. Dê exemplo de três métricas que devem ser públicas.

22.3. O que é uma medida indireta e por que tais medidas são comuns no trabalho com métricas de software?

22.4. Grady sugere uma etiqueta para métricas de software. Você conseguiria adicionar mais três regras àquelas mencionadas na Seção 22.1.1?

22.5. A equipe A encontrou 342 erros durante o processo de engenharia de software anterior à entrega. A equipe B encontrou 184 erros. Que medidas adicionais deveriam ser tomadas com relação aos projetos A e B para determinar qual das equipes eliminou erros mais eficientemente? Que métricas você proporia para ajudá-lo a fazer essa determinação? Que dados históricos poderiam ser úteis?

22.6. Apresente um argumento contra linhas de código como medida de produtividade de software. Esse argumento continuará válido quando dezenas ou centenas de projetos forem considerados?

22.7. Calcule o valor de pontos por função para um projeto com as seguintes características do domínio de informação:

Quantidade de entradas do usuário: 32
Quantidade de saídas do usuário: 60
Quantidade de consultas do usuário: 24
Quantidade de arquivos: 8
Quantidade de interfaces externas: 2

Considere que todos os valores de ajuste da complexidade são pela média. Use o algoritmo mencionado no Capítulo 15.

22.8. Usando a tabela apresentada na Seção 22.2.3, faça uma argumentação contra o uso da linguagem assembler com base na funcionalidade entregue por declaração de código. Outra vez referindo-se à tabela, discuta por que C++ apresentaria uma melhor alternativa do que C.

22.9. O software usado para controlar uma fotocopiadora requer 32.000 linhas de C e 4.200 linhas de Smalltalk. Estime o número de pontos por função do software da fotocopiadora.

22.10. Uma equipe de engenharia da Web construiu uma WebApp de e-comércio que contém 145 páginas individuais. Dessas páginas, 65 são dinâmicas, isto é, elas são geradas internamente com base em entrada do usuário final. Qual é o índice de personalização para essa aplicação?

Document shared on www.docsity.com

22.11. Uma WebApp e seu ambiente de apoio não estão plenamente fortificados contra ataques. Os engenheiros da Web estimam que a probabilidade de repelir um ataque é somente de 30%. O sistema não contém informação confidencial ou controversa, de modo que a probabilidade de ameaça é de 25%. Qual é a integridade da WebApp?

22.12. Na conclusão de um projeto que usou o Processo Unificado (Capítulo 3), verificou-se que 30 erros foram encontrados durante a fase de elaboração e 12 erros foram encontrados durante a fase de construção, quando foram rastreados até erros não descobertos na fase de elaboração. Qual é a DRE para essas duas fases?

22.13. Um incremento de software é entregue para os usuários finais por uma equipe de software. Os usuários descobrem oito defeitos durante o primeiro mês de uso. Antes da entrega, a equipe de software descobriu 242 erros durante revisões técnicas formais e todas as tarefas de teste. Qual é o DRE global para o projeto?

Leituras e Fontes de Informação Adicionais

Melhoria de processo de software (*Software Process Improvement*, SPI) recebeu bastante atenção ao longo das últimas duas décadas. Como medição e métricas de software são peças-chave para aperfeiçoar com sucesso o processo de software, muitos livros sobre SPI também discutem métricas. Fontes de informação que valem a pena sobre métricas de processo incluem:

Burr, A. e Owen M., *Statistical Methods for Software Quality*, International Thomson Publishing, 1996.

El Emam, K. e Madhavji, N. (eds.), *Elements of Software Process Assessment and Improvement*, IEEE Computer Society, 1999.

Florac, W. A. e Carleton A. D., *Measuring the Software Process: Statistical Process Control for Software Process Improvement*, Addison-Wesley, 1999.

Garmus, D. e Herron, D. *Measuring the Software Process: A Practical Guide to Functional Measurements*, Prentice-Hall, 1996.

Humphrey, W., *Introduction to the Team Software Process*, Addison-Wesley Longman, 2000.

Kan, S. H., *Metrics and Models in Software Quality Engineering*, Addison-Wesley, 1995.

McGarry e seus colegas (*Practical Software Measurement*, Addison-Wesley, 2001) apresentam uma sugestão detalhada para avaliação de processo de software. Uma coleção de artigos foi editada por Haug e seus colegas (*Software Process Improvement: Metrics, Measurement, and Process Modeling*, Springer-Verlag, 2001). Florac e Carlton (*Measuring the Software Process*, Addison-Wesley, 1999) e Fenton e Pfleeger (*Software Metrics: A Rigorous and Practical Approach*, Revised, Brooks/Cole Publishers, 1998) discutem como métricas de software podem ser usadas para fornecer indicadores necessários para melhorar o processo de software.

Putnam e Myers (*Five Core Metrics*, Dorset House, 2003) se apóiam em um banco de dados com mais de 6 mil projetos de software para demonstrar como cinco métricas principais – tempo, esforço, tamanho, confiabilidade e produtividade de processo — podem ser usadas para controlar projetos de software. Maxwell (*Applied Statistics for Software Managers*, Prentice-Hall, 2003) apresenta técnicas para analisar dados de projeto de software. Munson (*Software Engineering Measurement*, Auerbach, 2003) discute uma ampla variedade de tópicos de medição de engenharia de software. Jones (*Software Assessments, Benchmarks and Best Practices*, Addison-Wesley, 2000) descreve tanto fatores de medidas quantitativas quanto qualitativas que ajudam uma organização a avaliar seus processos e práticas de software. Garmus e Herron (*Function Point Analysis: Measurement Practices for Successful Software Projects*, Addison-Wesley, 2000) discutem métricas de processo com ênfase em análise de pontos por função.

Lorenze e Kidd [LOR94] e DeChampeax (*Object-Oriented Development Process and Metrics*, Prentice-Hall, 1996) consideram o processo OO e descrevem um conjunto de métricas para avaliá-lo. Whitmire (*Object-Oriented Design Measurement*, Wiley, 1997) e Henderson-Sellers (*Object-Oriented Metrics: Measures of Complexity*, Prentice-Hall, 1995) enfocam métricas técnicas para trabalho OO, mas também consideram medidas e métricas que podem ser usadas no nível de processo e produto.

Relativamente pouco tem sido publicado sobre métricas para trabalho de engenharia da Web. No entanto, Stern (*Web Metrics: Proven Methods for Measuring Web Site Success*, Wiley, 2002), Inan e Kean (*Measuring the Success of Your Website*, Longman, 2002), e Nobles e Grady (*Web Site Analysis and Reporting*, Premier Press, 2001) tratam de métricas para Web em uma perspectiva de negócio e marketing.

A última pesquisa na área de métricas é resumida pelo IEEE (*Symposium on Software Metrics*, publicado anualmente). Uma ampla variedade de fontes de informação sobre métricas de processo e projeto está disponível na Internet. Uma lista atualizada de referências da World Wide Web pode ser encontrada no site deste livro: **http://www.mhhe.com/pressman**.

Estimativa de Projetos de Software

CAPÍTULO 23

Conceitos-Chave

complexidade 528
estimativa
conceitos 524
baseada em FP 528
baseada em LOC 527
baseada em processo .. 529
de acomodação 532
de casos de uso 530
viabilidade 521
planejamento de projeto .. 521
recursos 522
escopo 522
dimensionamento do software 525

A gerência de projetos de software começa com um conjunto de atividades chamadas coletivamente de *planejamento do projeto*. Antes que o projeto possa começar, o gerente e a equipe de software precisam estimar o trabalho a ser feito, os recursos necessários e o tempo que vai decorrer do início ao fim. Uma vez que essas atividades são realizadas, a equipe de software deve estabelecer um cronograma de projeto que defina as tarefas e marcos da engenharia de software, identifique quem é o responsável por conduzir cada tarefa e especifique as dependências intertarefas que precisam ter um forte acompanhamento do progresso.

Em um excelente guia para "sobrevivência de projeto de software", Steve McConnell [MCC98] apresenta uma visão do mundo real de planejamento de projeto:

> Muitos trabalhadores técnicos preferem fazer o trabalho técnico do que gastar tempo planejando. Muitos gerentes técnicos não têm treinamento suficiente em gestão técnica para sentir confiança em que seu planejamento vai melhorar o resultado de um projeto. Como nenhuma parte deseja fazer planejamento, ele freqüentemente não é feito.
>
> Mas falha em planejar é um dos erros mais críticos que um projeto pode gerar... planejamento efetivo é necessário para resolver problemas na nascente [mais cedo no projeto] a baixo custo, em vez de na foz [mais tarde no projeto] a alto custo. O projeto gasta em média *80%* de seu tempo em retrabalho — consertando erros cometidos anteriormente no projeto.

McConnell alega que é possível encontrar tempo para planejar em relação a cada projeto (e para adaptar o plano ao longo do projeto) simplesmente tomando uma pequena porcentagem do tempo que seria gasto em retrabalho, o qual acaba ocorrendo em razão de o planejamento não ter sido conduzido.

Panorama

O que é? Uma necessidade real para software tem sido estabelecida; interessados estão atentos; engenheiros de software estão prontos para começar; e o projeto está para ser iniciado. Mas como você prossegue? Planejamento de projeto de software, na verdade, abrange as cinco principais atividades – estimativa, cronogramação, análise de risco, planejamento da gestão da qualidade e planejamento da gestão de modificação. No contexto deste capítulo, consideramos somente estimativa – a sua tentativa de determinar quanto dinheiro, esforço, recursos e tempo serão necessários para construir um sistema ou produto baseado em software específico.

Quem faz? Gerentes de software – usando informação solicitada aos interessados e engenheiros de software e dados de métricas de software coletados de projetos anteriores.

Por que é importante? Você construiria uma casa sem saber quanto iria gastar, as tarefas que precisaria realizar, e o tempo para o trabalho a ser conduzido? Certamente não, e como a maioria dos sistemas e produtos baseados em computador custa consideravelmente mais para construir que uma casa grande, parece razoável fazer uma estimativa antes de começar a criar o software.

Quais são os passos? A estimativa começa com uma descrição do escopo do produto. O problema é então decomposto em um conjunto de problemas menores e cada um desses é estimado usando dados históricos e experiência como diretriz. A complexidade e o risco do problema são considerados antes que uma estimativa final seja feita.

Qual é o produto do trabalho? Uma tabela simples, delineando as tarefas a serem realizadas, as funções a serem implementadas, e o custo, o esforço e o tempo envolvidos para cada uma delas, é gerada.

Como tenho certeza de que fiz corretamente? Isso é difícil, porque você realmente não saberá até que o projeto tenha sido completado. Todavia, se você tem experiência e segue uma abordagem sistemática, gera estimativas usando dados históricos sólidos e cria dados pontuais da estimativa, usando pelo menos dois métodos diferentes, estabelece uma cronogramação realística e continuamente se adapta à medida que o projeto avança, você pode ficar confiante de que fez o melhor.

Document shared on www.docsity.com

23.1 Observações sobre Estimativa

Planejamento requer que gerentes técnicos e membros da equipe de software selem um compromisso inicial, mesmo que seja provável que esse "compromisso" venha a provar-se errado.

Sempre que são feitas estimativas, olhamos para o futuro e aceitamos algum grau de incerteza como inevitável. Para citar Frederick Brooks [BRO75]:

> ... Nossas técnicas de estimativa são maldesenvolvidas. Mais seriamente, elas refletem um pressuposto não mencionado, que não é muito verdadeiro, isto é, de que tudo irá correr bem... como não estamos certos de nossas estimativas, freqüentemente falta aos gerentes de software a firmeza de fazer o pessoal esperar por um bom produto.

Apesar de estimar ser tanto arte como ciência, essa importante atividade não precisa ser conduzida de modo aleatório. Já existem técnicas úteis para estimativa de tempo e esforço. Métricas de processo e projeto podem fornecer perspectiva histórica e insumo poderoso para a geração de estimativas quantitativas. Experiência anterior (de todo o pessoal envolvido) pode ajudar imensamente, à medida que estimativas são desenvolvidas e revistas. Como a estimativa estabelece uma base para todas as outras atividades de planejamento de projeto e como este fornece um guia para a engenharia de software bem-sucedida, seria temerário começar sem ela.

> "Boas abordagens para estimativa e dados históricos sólidos oferecem a melhor perspectiva de que a realidade vai vencer as exigências impossíveis."
>
> **Capers Jones**

A estimativa de recursos, de custo e de cronograma para um esforço de engenharia de software exige experiência, acesso a boas informações históricas (métricas) e coragem de se empenhar em previsões quantitativas, quando informação qualitativa é tudo que existe. Estimativa tem risco[1] inerente e esse risco leva à incerteza.

A disponibilidade de informação histórica tem forte influência no risco da estimativa. Olhando para trás, podemos imitar coisas que funcionaram e aperfeiçoar áreas em que surgiram problemas. Quando métricas de software abrangentes (Capítulo 22), oriundas de projetos anteriores, estão disponíveis, as estimativas podem ser feitas com maior segurança, os cronogramas podem ser estabelecidos para evitar dificuldades enfrentadas antes e o risco global é reduzido.

> "A característica de uma mente instruída é ficar satisfeita com o grau de precisão que a natureza de um assunto admite, e não ficar buscando exatidão, quando apenas uma aproximação da verdade é possível."
>
> **Aristóteles**

O risco da estimativa é medido pelo grau de incerteza das estimativas quantitativas estabelecidas para recursos, custo e cronograma. Se o escopo do projeto é mal entendido ou se os requisitos estão sujeitos a mudanças, a incerteza e risco tornam-se perigosamente altos. O planejador e, principalmente, o cliente devem reconhecer que variabilidade nos requisitos de software significa instabilidade no custo e no cronograma.

Contudo, um gerente de projeto não deve ficar obcecado a respeito de estimativa. As abordagens modernas de engenharia de software (por exemplo, modelos incrementais de processos) assumem uma visão iterativa do desenvolvimento. Em tais abordagens, é possível – não significa que é sempre aceitável politicamente – revisitar a estimativa (à medida que mais informações são conhecidas) e revisá-la quando o cliente faz modificações nos requisitos.

1 Técnicas sistemáticas para análise de risco são apresentadas no Capítulo 25.

23.2 O Processo de Planejamento de Projeto

AVISO

Quanto mais você sabe, melhor você estima. Assim, atualize suas estimativas à medida que o projeto avança.

O objetivo do planejamento de projeto é fornecer um arcabouço que permita ao gerente fazer estimativas razoáveis de recursos, custo e cronograma. Além disso, as estimativas devem tentar definir cenários correspondentes tanto ao melhor quanto ao pior caso, de modo que o comportamento do projeto possa ser delimitado. Assim, o plano deve ser adaptado e atualizado à medida que o projeto avança. Nas seções seguintes, cada uma das atividades associadas com o planejamento de projeto de software é discutida.

Conjunto de Tarefas

Conjunto de Tarefas para Planejamento de Projeto

1. Estabeleça o escopo do projeto
2. Determine a viabilidade
3. Analise riscos (Capítulo 25)
4. Defina recursos necessários
   a. Determine recursos humanos necessários
   b. Defina recursos reusáveis de software
   c. Identifique recursos ambientais
5. Estime custo e esforço
   a. Decomponha o problema
   b. Desenvolva duas ou mais estimativas usando tamanho, pontos por função, tarefas de processo, ou casos de uso
   c. Harmonize as estimativas
6. Desenvolva um cronograma de projeto (Capítulo 24)
   a. Estabeleça um conjunto significativo de tarefas
   b. Defina uma rede de tarefas
   c. Use ferramentas de cronogramação para desenvolver um diagrama de tempo
   d. Defina mecanismos de rastreamento de cronograma

23.3 Escopo e Viabilidade do Software

PONTO CHAVE

Embora haja muitas razões para a incerteza, informação incompleta sobre os requisitos predomina.

O *escopo do software* descreve as funções e características a serem entregues aos usuários finais, os dados que entram e saem, o "conteúdo" que é apresentado aos usuários como conseqüência do uso do software e o desempenho, as restrições, as interfaces e a confiabilidade que *limitam* o sistema. Escopo é definido usando uma das duas técnicas.

1. A descrição narrativa do escopo do software é desenvolvida depois da comunicação com todos os interessados.
2. Um conjunto de casos de uso[2] é desenvolvido pelos usuários finais.

As funções descritas na declaração de escopo (ou nos casos de uso) são avaliadas, e em alguns casos refinadas, para fornecer mais detalhes antes do início da estimativa. Como tanto as estimativas de custo quanto o cronograma são orientados funcionalmente, algum grau de decomposição é freqüentemente útil. Considerações de desempenho abrangem requisitos de processamento e tempo de resposta. Restrições identificam limites colocados no software pelo hardware externo, disponibilidade de memória ou outros sistemas existentes.

Uma vez identificado o escopo (com a concordância do cliente), é razoável perguntar: "Podemos construir um software para satisfazer esse escopo? O projeto é exeqüível?". Freqüentemente, os engenheiros passam por cima dessas questões (ou são pressionados para desconsiderá-las por

2 Casos de uso foram discutidos em detalhe ao longo das Partes 2 e 3 deste livro. Um caso de uso é uma descrição baseada em um cenário da interação do usuário com o software do ponto de vista do usuário.

Document shared on www.docsity.com

AVISO

A viabilidade de um projeto é importante, mas uma consideração da necessidade do negócio é ainda mais importante. Não é vantagem alguma construir um sistema ou um produto de alta tecnologia que realmente ninguém quer.

gerentes ou clientes pacientes), para ficar envolvidos apenas em um projeto que está condenado desde o início. Putnam e Myers [PUT97a] tratam desse assunto quando escrevem:

> [N]em tudo o que é imaginável é exeqüível, nem mesmo em software, que pode ser imperceptível a estranhos. Pelo contrário, a exeqüibilidade de software tem quatro dimensões sólidas: *Tecnologia* — O projeto é exeqüível tecnicamente? Está dentro do estado da arte? Os defeitos podem ser reduzidos em um nível que satisfaça as necessidades de aplicação? *Finança* — O projeto é exeqüível financeiramente? O desenvolvimento pode ser completado a um custo que a organização de software, seu cliente ou o mercado possam pagar? *Tempo* — O prazo para o projeto chegar ao mercado vai vencer a concorrência? *Recursos* — A organização tem os recursos necessários para obter sucesso?

Putnam e Myers sugerem corretamente que o estabelecimento do escopo não é suficiente. Uma vez entendido o escopo, a equipe de software e os outros devem trabalhar para determinar se pode ser feito dentro das dimensões mencionadas. Essa é uma parte crucial, freqüentemente esquecida, do processo de estimativa.

23.4 Recursos

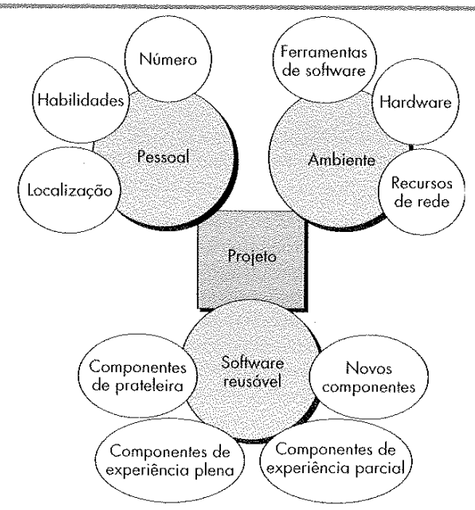
A segunda tarefa de planejamento de software é estimar os recursos necessários para realizar o esforço de desenvolvimento de software. A Figura 23.1 ilustra as três principais categorias de recursos de engenharia de software — pessoas, componentes de software reusável e o ambiente de desenvolvimento (ferramentas de hardware e de software). Cada recurso é especificado por quatro características: descrição do recurso; declaração de disponibilidade; época em que o recurso será necessário; prazo durante o qual o recurso será aplicado. As duas últimas características podem ser vistas como uma *janela de tempo*. A disponibilização do recurso para uma janela especificada deve ser estabelecida o mais cedo possível.

23.4.1 Recursos Humanos

O planejador começa pela avaliação do escopo e pela seleção das aptidões necessárias para completar o desenvolvimento. Tanto a posição na organização (por exemplo, gerente, engenheiro de software experiente) quanto a especialidade (por exemplo, telecomunicações, bases de dados, cliente/servidor) são especificadas. Para projetos relativamente pequenos (poucas pessoa-mês), um

FIGURA 23.1

Recursos de projeto



único indivíduo pode realizar todas as tarefas de engenharia de software, consultando especialistas quando necessário. Para projetos grandes, a equipe de software pode estar geograficamente dispersa em um número diferente de localizações. Assim, a localização de cada recurso humano é especificada.

A quantidade de pessoas necessárias para um projeto de software só pode ser determinada depois de ser feita uma estimativa do esforço de desenvolvimento (por exemplo, pessoa-mês). Técnicas para a estimativa de esforço são discutidas posteriormente neste capítulo.

23.4.2 Recursos de Software Reusáveis

Engenharia de software baseada em componentes (Capítulo 30) enfatiza reusabilidade — isto é, a criação e o reuso de blocos construtivos de software [HOO91]. Esses blocos construtivos, freqüentemente chamados *componentes*, devem ser catalogados para facilitar a referência, padronizados para facilitar a aplicação e validados para facilitar a integração.

Bennatan [BEN92] sugere quatro categorias de recurso de software, que devem ser consideradas à medida que o planejamento avança:

AVISO

Nunca esqueça que integrar uma variedade de componentes reusáveis pode ser um desafio significativo. O problema da integração freqüentemente volta à tona à medida que vários componentes são melhorados.

Componentes de prateleira. Software existente pode ser adquirido de terceiros ou ter sido desenvolvido internamente para um projeto anterior. Componentes de prateleira (*commercial of-the-shelf* — COTS) são adquiridos de terceiros, estão prontos para uso no projeto corrente e foram plenamente validados.

Componentes de experiência plena. Especificações, projetos, código ou dados de teste existentes desenvolvidos para projetos anteriores, que são similares ao software a ser construído para o presente projeto. Os membros da equipe atual de software têm plena experiência da área de aplicação representada por esses componentes. Assim, as modificações necessárias para os componentes de experiência plena serão de risco relativamente baixo.

Componentes de experiência parcial. Especificações, projetos, código ou dados de teste existentes desenvolvidos para projetos anteriores, que são relacionados ao software a ser construído para o presente projeto, mas que vão exigir substancial modificação. Os membros da atual equipe de software têm apenas experiência limitada na área de aplicação representada por esses componentes. Assim, as modificações necessárias nos componentes de experiência parcial têm um grau de risco considerável.

Componentes novos. Componentes de software que precisam ser construídos pela equipe de software especificamente para as necessidades do presente projeto.

Ironicamente, componentes de software reusáveis são freqüentemente negligenciados durante o planejamento, tornando-se foco de preocupação somente durante a fase de desenvolvimento do processo de software. É melhor especificar logo os requisitos dos recursos de software. Desse modo, a avaliação técnica das alternativas pode ser efetuada e a aquisição pode ocorrer a tempo.

23.4.3 Recursos de Ambiente

O ambiente que apóia o projeto de software, freqüentemente chamado *ambiente de engenharia de software* (*software engineering environment* — SEE), incorpora hardware e software. O hardware fornece a plataforma que apóia as ferramentas (software) necessárias para produzir os produtos de trabalho, que são resultado de boas práticas de engenharia de software[3]. Como a maioria das organizações de software tem múltiplas localizações que exigem acesso ao SEE, o planejador de um projeto deve fixar o período de tempo exigido para o hardware e para o software, e verificar se esses recursos estarão disponíveis.

Quando um sistema baseado em computador (que incorpora hardware e software especializados) tiver que passar por engenharia, a equipe de software precisará ter acesso a elementos de hardware em desenvolvimento por outras equipes de engenharia. Por exemplo, o software para

3 Outro hardware — o ambiente-alvo — é(são) o(s) computador(es) no(s) qual(is) o software será executado quando for entregue ao usuário final.

Document shared on www.docsity.com

controle numérico (*numerical control* — NC) usado em uma classe de máquinas-ferramentas pode exigir uma máquina-ferramenta específica (por exemplo, uma fresa de controle numérico) como parte do passo de teste de validação; um projeto de software avançado para leiaute de página pode precisar de um sistema de composição digital em algum período do desenvolvimento. Cada elemento de hardware deve ser especificado pelo planejador do projeto de software.

23.5 Estimativa do Projeto de Software

O software é o elemento virtualmente mais caro de todos os sistemas baseados em computador. Para sistemas complexos, feitos sob medida, um erro de estimativa grande pode fazer a diferença entre lucro e prejuízo. Excesso de custo pode ser desastroso para o desenvolvedor.

> "Em uma era de terceirização e concorrência acirrada, a capacidade de estimar mais precisamente... surgiu como fator crítico de sobrevivência para muitos grupos de tecnologia da informação."
>
> **Rob Thomsett**

AVISO

Embora o esforço de engenharia de software seja um elemento dominante do custo de projeto, é importante lembrar que outros custos (por exemplo, ambiente e ferramentas de desenvolvimento, viagem, treinamento, espaço de escritório, hardware) também devem ser considerados.

A estimativa de custo e de esforço de software nunca será uma ciência exata[4]. Um demasiado número de variáveis — humanas, técnicas, ambientais, políticas — pode afetar o custo final do software e o esforço aplicado para desenvolvê-lo. Todavia, a estimativa de projetos de software pode ser transformada de algo sobrenatural em uma série de passos sistemáticos que fornecem estimativas com risco aceitável.

Existem algumas opções para conseguir estimativas de custo e esforço confiáveis:

1. Adie a estimativa até que o projeto esteja mais adiantado (obviamente, podemos conseguir estimativas 100% precisas depois que o projeto estiver completo!).
2. Baseie as estimativas em projetos semelhantes, que já foram completados.
3. Use técnicas de decomposição relativamente simples para gerar estimativas de custo e esforço do projeto.
4. Use um ou mais modelos empíricos para estimativa de custo e esforço do software.

Infelizmente, a primeira opção, apesar de atraente, não é prática. Estimativas de custo precisam ser fornecidas "logo". Entretanto, é preciso reconhecer que quanto mais esperarmos, mais saberemos, e quanto mais soubermos, será menos provável cometermos erros sérios em nossas estimativas.

A segunda função pode funcionar razoavelmente bem se o presente projeto for bastante semelhante a esforços anteriores e as outras influências do projeto (por exemplo, o cliente, as condições do negócio, o SEE, os prazos) forem equivalentes. Infelizmente, a experiência anterior nem sempre é um bom indicador de resultados futuros.

As opções restantes são abordagens viáveis para a estimativa de projetos de software. O ideal é que as técnicas citadas para cada opção sejam aplicadas em conjunto; cada uma usada para verificar a outra. Técnicas de decomposição usam a abordagem "dividir e conquistar" para a estimativa de projetos de software. Pela decomposição de um projeto em suas funções principais e atividades relacionadas de engenharia de software, a estimativa de custo e esforço pode ser realizada passo a passo. Modelos empíricos de estimativa podem ser usados para complementar as técnicas de decomposição e oferecem por si mesmos uma valiosa abordagem de estimativa. Esses modelos são discutidos na Seção 23.7.

Cada uma das opções viáveis de estimativa de custo de software é boa na mesma medida em que o são os dados históricos usados para alimentar a estimativa. Se não existem dados históricos, o custo se baseia em uma fundação precária. No Capítulo 22 examinamos as características de algumas das métricas de software que fornecem o banco de dados históricos para estimativas.

4 Bennatan [BEN03] relata que 40% dos desenvolvedores de software continuam a fazer esforço com estimativa e que o tamanho do software e o tempo de desenvolvimento são muito difíceis de estimar precisamente.

23.6 Técnicas de Decomposição

A estimativa de projetos de software é uma forma de solução de problemas e, na maioria dos casos, o problema a ser resolvido (por exemplo, desenvolver uma estimativa de custo e esforço para um projeto de software) é muito complexo para ser considerado como um todo. Por essa razão, nós decompomos o problema, recaracterizando-o como um conjunto de problemas menores (e, espera-se, mais fácil de digerir).

No Capítulo 21, a abordagem de decomposição foi discutida de dois pontos de vista diferentes: decomposição do problema e decomposição do processo. A estimativa usa uma ou ambas as formas de particionamento. Mas, antes que uma estimativa possa ser feita, o planejador do projeto precisa entender o escopo do software a ser construído e gerar uma estimativa do seu "tamanho".

23.6.1 Dimensionamento do Software

PONTO CHAVE

O "tamanho" do software a ser construído pode ser estimado usando uma medida direta, LOC, ou uma medida indireta, FP.

A precisão da estimativa de um projeto de software depende de alguns fatores: (1) o grau com que o planejador estimou adequadamente o tamanho do produto a ser construído; (2) a aptidão para traduzir a estimativa de tamanho em esforço humano, tempo transcorrido e dinheiro (em função da disponibilidade de métricas de software confiáveis de projetos anteriores); (3) o grau com que o plano de projeto reflete a capacidade da equipe de software e (4) a estabilidade dos requisitos do produto e do ambiente que apóiam o esforço de engenharia de software.

Nesta seção, consideramos o problema de *dimensionamento de software*. Como uma estimativa de projeto é boa na mesma medida em que o é a estimativa do tamanho do trabalho a ser realizado, o dimensionamento representa o primeiro grande desafio do planejador de projeto. No contexto do planejamento de projeto, o tamanho refere-se ao resultado quantificável do projeto de software. Se uma abordagem direta é adotada, o tamanho pode ser medido em linhas de código (LOC). Se uma abordagem indireta é escolhida, o tamanho é representado como pontos por função (FP).

? Como dimensionamos o software que planejamos construir?

Putnam e Myers [PUT92] sugerem quatro abordagens diferentes para o problema do dimensionamento:

- *Dimensionamento de "lógica nebulosa".* Para aplicar essa abordagem, o planejador deve identificar o tipo de aplicação, estabelecer sua magnitude em uma escala qualitativa e depois refiná-la dentro do intervalo original.
- *Dimensionamento de pontos por função.* O planejador desenvolve estimativas das características do domínio da informação, discutidas no Capítulo 15.
- *Dimensionamento de componentes padrão.* O software é composto por vários "componentes padrão" diferentes, que são genéricos para uma área de aplicação específica. Por exemplo, os componentes padrão para um sistema de informação são subsistemas, módulos, telas, relatórios, programas interativos, programas com processamento por lotes, arquivos, LOC e instruções em nível de objeto. O planejador do projeto estima a quantidade de ocorrências de cada componente padrão e depois usa dados históricos de projeto para determinar o tamanho correspondente a cada componente padrão.
- *Dimensionamento de modificação.* Essa abordagem é usada quando um projeto abrange o uso de software existente, que precisa ser modificado de algum modo como parte do projeto. O planejador estima a quantidade e o tipo (por exemplo, reuso, adição de código, modificação de código, eliminação de código) das modificações que precisam ser efetuadas.

Putnam e Myers sugerem que os resultados de cada uma dessas abordagens de dimensionamento sejam combinados estatisticamente para criar uma estimativa de *três pontos* ou *valor esperado*. Isso é conseguido desenvolvendo um valor otimista (baixo), um mais provável e um pessimista (alto) para o tamanho, e combinando-os por meio da equação (23-1) descrita na próxima seção.

Document shared on www.docsity.com

23.6.2 Estimativa Baseada no Problema

No Capítulo 22, linhas de código e pontos por função foram descritos como medidas a partir das quais as métricas de produtividade podem ser calculadas. Dados de LOC e FP são usados de dois modos durante a estimativa de projetos de software: (1) como variável de estimativa para "dimensionar" cada elemento do software e (2) como métricas referenciais coletadas de projetos anteriores e usadas juntamente com variáveis de estimativa para desenvolver projeções de custo e esforço.

? O que as estimativas orientadas a LOC e FP têm em comum?

As estimativas LOC e FP são técnicas distintas. Todavia, ambas têm algumas características em comum. O planejador do projeto começa com uma declaração delimitada do escopo do software e a partir dela tenta decompor o software em funções do problema que podem ser estimadas individualmente. LOC ou FP (a variável de estimativa) é então estimada para cada função. Como alternativa, o planejador pode escolher outro componente para dimensionar, como classes ou objetos, modificações ou processos do negócio afetados.

AVISO

Quando coletar métricas de produtividade para projetos, estabeleça uma taxonomia de tipos de projeto. Isso vai possibilitar-lhe calcular médias específicas por domínio, tornando a estimativa mais precisa.

Métricas referenciais de produtividade (por exemplo, LOC/pm ou FP/pm[5]) são então aplicadas à variável de estimativa adequada e o custo ou esforço correspondente à função é derivado. As estimativas de função são combinadas para produzir uma estimativa global para todo o projeto.

É importante notar, entretanto, que freqüentemente há substancial espalhamento nas métricas de produtividade de uma organização, fazendo com que o uso de uma única métrica de produtividade referencial seja suspeito. Em geral, as médias de LOC/pm ou FP/pm devem ser calculadas por domínio de projeto. Isto é, os projetos devem ser agrupados por tamanho de equipe, área de aplicação, complexidade e outros parâmetros relevantes. Médias locais dos domínios devem então ser calculadas. Quando um novo projeto é estimado, deve ser primeiro classificado em um domínio e depois a média de produtividade do domínio adequado deve ser usada para gerar a estimativa.

As técnicas de estimativa LOC e FP diferem quanto ao nível de detalhe necessário para a decomposição e quanto ao objetivo do particionamento. Quando LOC é usada como variável de estimativa, a decomposição é absolutamente essencial e é freqüentemente levada a um considerável nível de detalhe. Quanto maior for o grau de particionamento, mais provável será que estimativas razoavelmente precisas de LOC possam ser desenvolvidas.

PONTO CHAVE

Para estimativas de FP, a decomposição focaliza as características do domínio da informação.

Para estimativas de FP, a decomposição funciona de modo diferente. Em vez de focalizar a função, cada uma das cinco características do domínio de informação bem como os 14 valores de ajuste de complexidade discutidos no Capítulo 15 são estimados. As estimativas resultantes podem, então, ser usadas para derivar um valor de FP, que pode ser relacionado a dados anteriores e usado para gerar uma estimativa.

Independentemente da variável de estimativa que é usada, o planejador do projeto começa estimando um intervalo de valores para cada função ou valor do domínio da informação. Usando dados históricos ou (quando tudo falha) intuição, o planejador estima um valor de tamanho otimista, um mais provável e um pessimista para cada função, ou contagem, para cada valor do domínio da informação. Uma indicação implícita do grau de incerteza é fornecida quando um intervalo de valores é especificado.

? Como calculo o "valor esperado" do tamanho do software?

Um valor esperado, ou de três pontos, pode então ser calculado. O *valor esperado* para a variável de estimativa (tamanho), S, pode ser calculado como média ponderada das estimativas otimista (S_{ot}) mais provável (S_m) e pessimista (S_{pess}). Por exemplo,

$$S = (S_{ot} + 4S_m + S_{pess})/6 \qquad (23\text{-}1)$$

dá maior peso à estimativa "mais provável" e segue uma distribuição de probabilidade beta. Consideramos que há uma probabilidade muito pequena de que o resultado correspondente ao tamanho real caia fora do intervalo entre os valores otimista e pessimista.

Uma vez determinado o valor esperado para a variável de estimativa, os dados históricos de produtividade LOC ou FP são aplicados. As estimativas estão corretas? A única resposta razoável para essa questão é: não podemos estar seguros. Qualquer técnica de estimativa, não importa quão sofisticada, deve ser comparada com outra abordagem. Ainda assim, devem prevalecer o bom senso e a experiência.

5 A sigla *pm* significa o esforço pessoa-mês (*person-month*).

AVISO

Muitas aplicações modernas residem em uma rede ou são parte de uma arquitetura cliente/servidor. Assim, certifique-se de que suas estimativas incluem o esforço necessário para o desenvolvimento de software de "infra-estrutura".

23.6.3 Um Exemplo de Estimativa Baseada em LOC

Como exemplo das técnicas de estimativa baseadas no problema LOC e FP, consideremos um pacote de software a ser desenvolvido para uma aplicação de projeto apoiado por computador de componentes mecânicos. O software deve ser executado em uma estação de trabalho de engenharia e vai fazer a interface com vários periféricos de computação gráfica, que incluem mouse, digitalizador, monitor de vídeo colorido de alta resolução e impressora a laser. Uma declaração preliminar do escopo do software pode ser desenvolvida:

> O software mecânico CAD vai aceitar dados geométricos bi e tridimensionais de um engenheiro. O engenheiro vai interagir e controlar o sistema CAD por meio de uma interface de usuário, que vai exibir características de bom projeto de interface homem/máquina. Todos os dados geométricos e outras informações de apoio vão ser mantidos em um banco de dados CAD. Módulos de análise de projeto vão ser desenvolvidos para produzir as saídas necessárias, que vão ser mostradas em diversos dispositivos gráficos. O software vai ser projetado para controlar e interagir com dispositivos periféricos que incluem mouse, digitalizador, impressora a laser e plotter.

Essa declaração de escopo é preliminar — não é delimitada. Cada sentença teria que ser expandida para fornecer detalhes concretos e limites quantitativos. Por exemplo, antes que a estimativa possa ter início, o planejador precisa determinar o que significa "características de bom projeto de interface homem/máquina", ou qual o tamanho e sofisticação necessários do "banco de dados CAD".

AVISO

Não caia na tentação de usar esse resultado como estimativa de seu projeto. Você deve derivar outra estimativa usando uma abordagem diferente.

Para o nosso propósito, consideramos que houve maior refinamento e que as seguintes funções principais do software listadas na Figura 23.2 foram identificadas. Seguindo a técnica de decomposição para LOC, é desenvolvida uma tabela de estimativa, mostrada na Figura 23.2. Um intervalo de estimativa de LOC é desenvolvido para cada função. Por exemplo, o intervalo de estimativas de LOC para a função de análise geométrica 3D é otimista — 4.600 LOC, mais provável — 6.900 LOC e pessimista — 8.600 LOC. Aplicando a Equação (23-1), o valor esperado para a função de análise geométrica 3D é 6.800 LOC. Outras estimativas são derivadas de modo análogo. Somando verticalmente os valores da coluna de estimativas de LOC, é estabelecida uma estimativa de 33.200 linhas de código para o sistema CAD.

Uma pesquisa dos dados históricos indica que a produtividade organizacional média para sistemas desse tipo é de 620 LOC/pm. Com base no valor bruto da mão-de-obra de 8.000 dólares por mês, o custo por linha de código é de aproximadamente 13 dólares. Com base na estimativa de LOC e nos dados históricos de produtividade, o custo total estimado para o projeto é de 431.000 dólares e o esforço estimado de 54 pessoas-mês[6].

FIGURA 23.2

Tabela de estimativa para o método LOC

Função	Estimativa de LOC
Recursos de controle e interface com o usuário (UICF)	2.300
Análise geométrica bidimensional (2DGA)	5.300
Análise geométrica tridimensional (3DGA)	6.800
Gestão de base de dados (DBM)	3.350
Recursos de visualização da computação gráfica (CGDF)	4.950
Função de controle de periféricos (PCF)	2.100
Módulos de análise do projeto (DAM)	8.400
Estimativa das linhas de código	*33.200*

6 As estimativas são arredondadas para o próximo milhar de dólares e pessoa-mês. Mais precisão é desnecessária e irreal, dada a limitação da precisão de estimativa.

Document shared on www.docsity.com

CASASEGURA

Estimativa

A cena: Escritório de Doug Miller quando o planejamento de projeto começa.

Os personagens: Doug Miller (gerente da equipe de engenharia de software do *CasaSegura*) e Vinod Raman, Jamie Lazar, e outros membros da equipe de engenharia de software do produto.

A conversa:

Doug: Precisamos desenvolver uma estimativa de esforço para o projeto, e então temos que definir um microcronograma para o primeiro incremento e um macrocronograma para os incrementos restantes.

Vinod (balançando a cabeça afirmativamente): Certo, mas ainda não definimos nenhum incremento.

Doug: Verdade, mas é por isso que precisamos estimar.

Jamie (carrancudo): Você quer saber quanto tempo isso vai levar?

Doug: Eis aqui o que eu preciso. Primeiro, precisamos decompor a funcionalidade do software *CasaSegura*... em alto nível... depois temos que estimar o número de linhas de código que cada função vai ter... depois...

Jamie: Ei! Como vamos fazer isso?

Vinod: Eu tenho feito isso em projetos anteriores. Você usa casos de uso, determina a funcionalidade necessária para implementar cada um, imagina a estimativa do número de LOC para cada parte da função. A melhor abordagem é ter alguém que faça isso independentemente e depois comparar os resultados.

Doug: Ou você pode fazer uma decomposição funcional do projeto todo.

Jamie: Mas isso não vai acabar nunca, e temos que começar.

Vinod: Não... isso pode ser feito em poucas horas... na realidade, nesta manhã.

Doug: Eu concordo... não podemos esperar exatidão, apenas uma idéia aproximada de qual será o tamanho do *CasaSegura*.

Jamie: Eu acho que deveríamos apenas estimar esforço... só isso.

Doug: Faremos isso também e muito mais. Depois usaremos ambas as estimativas como verificação cruzada.

Vinod: Vamos fazer...

23.6.4 Um Exemplo de Estimativa Baseada em FP

A decomposição para estimativa baseada em FP focaliza os valores do domínio da informação em vez das funções do software. Com referência à tabela de cálculo de pontos por função, apresentada na Figura 23.3, o planejador do projeto estima entradas, saídas, consultas, arquivos e interfaces externas para o software CAD. FP são calculados usando a técnica discutida no Capítulo 15. Para essa estimativa, o fator de ponderação da complexidade é considerado médio. A Figura 23.3 apresenta os resultados dessa estimativa.

Cada um dos fatores de ponderação da complexidade é estimado e o fator de ajuste da complexidade é calculado como descrito no Capítulo 15:

FIGURA 23.3

Estimativa dos valores do domínio da informação

Valor do domínio da informação	Otimista	Mais provável	Pessimista	Contagem	Peso	Contagem FP
Quantidade de entradas	20	24	30	24	4	97
Quantidade de saídas	12	15	22	16	5	78
Quantidade de consultas	16	22	28	22	5	88
Quantidade de arquivos	4	4	5	4	10	42
Quantidade de interfaces externas	2	2	3	2	7	15
Contagem total						*320*

Fator	Valor
1. Salvamento e recuperação	4
2. Comunicação de dados	2
3. Processamento distribuído	0
4. Desempenho crítico	4
5. Ambiente operacional existente	3
6. Entrada de dados on-line	4
7. Transações de entrada em diversas telas	5
8. Arquivos-mestre atualizados on-line	3
9. Valores do domínio da informação complexos	5
10. Processamento interno complexo	5
11. Código projetado para reuso	4
12. Conversão/instalação no projeto	3
13. Instalações múltiplas	5
14. Aplicação projetada para modificações	5
Fator de ajuste da complexidade	**1,17**

Finalmente, a quantidade estimada de FP é derivada:

$$FP_{estimado} = \text{cont.total} \times [0{,}65 + 0{,}01 \times S\,(F_i)]$$
$$FP_{estimado} = 375$$

A produtividade organizacional média para sistemas desse tipo é de 6,5 FP/pm. Baseado em um valor bruto salarial de 8.000 dólares por mês, o custo por FP é de aproximadamente 1.230 dólares. Baseado na estimativa LOC e nos dados históricos de produtividade, o custo total estimado do projeto é de 461.000 dólares e o esforço estimado de 58 pessoas-mês.

AVISO

Se o tempo permitir, use maior granularidade ao especificar as tarefas da Figura 23.4. Por exemplo, subdividir a análise em suas tarefas principais e estimar cada uma separadamente.

23.6.5 Estimativa Baseada em Processo

A técnica mais comum para estimar um projeto é basear a estimativa em um processo que será usado. Isto é, o processo é decomposto em um conjunto relativamente pequeno de tarefas e o esforço necessário para realizar cada tarefa é estimado.

Como outras técnicas baseadas no problema, a estimativa baseada no processo começa com um delineamento das funções de software obtidas do escopo do projeto. Uma série de atividades de processo de software precisa ser realizada para cada função. As funções e as atividades[7] de arcabouço relacionadas podem ser representadas como parte de uma tabela semelhante àquela apresentada na Figura 23.4.

Uma vez combinadas as funções do problema e as atividades do processo, o planejador estima o esforço (por exemplo, pessoas-mês) que será necessário para realizar cada atividade do processo de software, para cada função do software. Esses dados constituem a matriz central da tabela da Figura 23.4. Taxas médias de trabalho (por exemplo, custo/unidade de esforço) são então aplicadas ao esforço estimado para cada atividade do processo. É muito provável que a taxa de trabalho varie para cada tarefa. O pessoal mais experiente, que está mais envolvido nas primeiras atividades de arcabouço, é em geral mais bem remunerado do que o pessoal menos experiente envolvido nas tarefas de projeto posteriores, na geração de código e nos primeiros testes.

O custo e o esforço para cada função e atividade de arcabouço do processo de software são calculados como último passo. Se a estimativa baseada no processo é realizada, independentemente

7 As atividades de arcabouço escolhidas para esse projeto diferem um pouco das atividades genéricas discutidas no Capítulo 2. São: comunicação com o cliente (CC), planejamento, análise de risco, engenharia e construção/entrega.

Document shared on www.docsity.com

FIGURA 23.4

Tabela de estimativa baseada no processo

Atividade →	CC	Planejamento	Análise de risco	Engenharia		Liberação para construção		AC	Totais
Tarefa →				Análise	Projeto	Código	Teste		
Função ↓									
UICF				0,50	2,50	0,40	5,00	n/d	8,40
2DGA				0,75	4,00	0,60	2,00	n/d	7,35
3DGA				0,50	4,00	1,00	3,00	n/d	8,50
CGDF				0,50	3,00	1,00	1,50	n/d	6,00
DBM				0,50	3,00	0,75	1,50	n/d	5,75
PCF				0,25	2,00	0,50	1,50	n/d	4,25
DAM				0,50	2,00	0,50	2,00	n/d	5,00
Totais	0,25	0,25	0,25	3,50	20,50	4,50	16,50		46,00
% do esforço	1%	1%	1%	8%	45%	10%	36%		

CC = comunicação com o cliente AC = avaliação pelo cliente

da estimativa de LOC ou FP, dispomos agora de duas ou três estimativas para o custo e o esforço, que podem ser comparadas e conciliadas. Se os dois conjuntos de estimativas exibem razoável concordância, há forte razão para acreditar que as estimativas são confiáveis. Se, por outro lado, os resultados dessas técnicas de decomposição mostram pouca concordância, mais pesquisa e análise devem ser realizadas.

> "É melhor entender a base de uma estimativa antes que você a use."
>
> **Barry Boehm e Richard Fairley**

23.6.6 Um Exemplo de Estimativa Baseada em Processo

Para ilustrar o uso de estimativa baseada em processo, consideramos novamente o software CAD introduzido na Seção 23.6.3. A configuração do sistema e todas as funções do software permanecem inalteradas e são indicadas pelo escopo do projeto.

Com referência à tabela completa, baseada no processo, mostrada na Figura 23.4, as estimativas do esforço (em pessoas-mês), para cada atividade de engenharia de software, são fornecidas para cada função do software CAD (cujos nomes estão abreviados). As atividades de engenharia e construção são subdivididas nas principais tarefas de engenharia de software mostradas. São fornecidas estimativas grosseiras do esforço para comunicação com o cliente, planejamento e análise de risco. Elas estão anotadas na linha de totais, na parte inferior da tabela. Os totais horizontais e verticais fornecem uma indicação da estimativa do esforço necessário para análise, projeto, código e teste. Deve-se notar que 53% de todo o esforço é dispendido nas tarefas iniciais de engenharia (análise de requisitos e projeto), indicando a importância relativa desse trabalho.

Com base na taxa média bruta de mão-de-obra de 8.000 dólares por mês, o total estimado do custo do projeto é de 368.000 dólares e o esforço estimado é de 46 pessoas-mês. Se necessário, taxas de mão-de-obra podem ser associadas a cada atividade de arcabouço ou tarefa de engenharia de software, e calculadas separadamente.

23.6.7 Estimativas com Casos de Uso

Como mencionamos no docorrer das Partes 2 e 3 deste livro, casos de uso fornecem a uma equipe de software discernimento do escopo e requisitos do software. No entanto, desenvolver uma abordagem de estimativa com casos de uso é problemático pelas seguintes razões [SMI99]:

? Por que é difícil desenvolver uma técnica de estimativa usando casos de uso?

- Casos de uso são descritos usando muitos formatos e estilos diferentes — não há um formato padrão.
- Casos de uso representam uma visão externa (a visão do usuário) do software e são freqüentemente escritos em diferentes níveis de abstração.

- Casos de uso não tratam da complexidade e das características das funções que são descritas.
- Casos de uso não descrevem comportamento complexo (por exemplo, interações) que envolvem muitas funções e características.

Diferentemente de uma LOC ou ponto por função, um "caso de uso" de uma pessoa pode precisar meses de esforço, enquanto um caso de uso de outra pessoa pode ser implementado em um ou dois dias.

Embora um certo número de investigadores tivesse considerado casos de uso uma entrada de estimativa, nenhum método de estimativa comprovado surgiu até agora. Smith [SMI99] sugere que casos de uso podem ser usados para estimativa, mas somente se eles forem considerados no contexto da "hierarquia estrutural" que casos de uso descrevem.

Smith alega que qualquer nível dessa hierarquia estrutural pode ser descrito por não mais do que 10 casos de uso. Cada um desses casos de uso deveria englobar não mais do que 30 cenários distintos. Obviamente, casos de uso que descrevem um sistema grande são escritos em um nível muito mais alto de abstração (e representam consideravelmente mais esforço de desenvolvimento) do que casos de uso escritos para descrever um único subsistema. Assim, antes que casos de uso possam ser usados para estimativa, o nível dentro da hierarquia estrutural é estabelecido, o tamanho médio (em páginas) de cada caso de uso é determinado, o tipo de software (por exemplo, tempo real, negócio, engenharia/científico, embutido) é definido, e um esboço de arquitetura do sistema é considerado. Uma vez estabelecidas essas características, dados empíricos podem ser usados para estabelecer o número estimado de LOC ou FP por caso de uso (para cada nível da hierarquia). Dados históricos são então usados para calcular o esforço necessário para desenvolver o sistema.

Para ilustrar como esse cálculo pode ser feito, considere o seguinte relacionamento[8]:

$$\text{LOC estimado} = N \times \text{LOC}_{avg} + [(S_a/S_h - 1) + (P_a/P_h - 1)] \times \text{LOC}_{adjust} \qquad (23\text{-}2)$$

em que

N = número real (Actual Number) de casos de uso
LOC_{avg} = média histórica (Historical Average) de LOC por caso de uso para esse tipo de subsistema
LOC_{adjust} = representa um ajuste (adjustment) com base em n por cento de LOC_{avg} em que n é definido localmente e representa a diferença entre esse projeto e a "média" dos projetos.
S_a = cenários reais (Actual Scenarios) por caso de uso
S_h = média de cenários (Average Scenarios) por caso de uso para esse tipo de subsistema
P_a = páginas reais (Actual Pages) por caso de uso
P_h = média de páginas (Average Pages) por caso de uso para esse tipo de subsistema

A Equação (23-2) é usada para desenvolver um esboço de estimativa do número de LOC baseado no número real de casos de uso ajustado pelo número de cenários e tamanho em página dos casos de uso. O ajuste representa até n por cento da média histórica de LOC por caso de uso.

23.6.8 Um Exemplo de Estimativa Baseada em Casos de Uso

O software CAD introduzido na Seção 23.6.3 é composto de três grupos de subsistemas:

- Subsistema de interface com o usuário (inclui *user interface and control facilities* — UICF).
- Grupo de subsistemas de engenharia (inclui o susbsistema 2DGA (*two-dimensional geometric analysis*), subsistema 3DGA (*three-dimensional geometric analysis*) e subsistema DAM (*design analysis modules*).

8 É importante notar que a Equação (23-2) é usada apenas com finalidade ilustrativa. Como todos os modelos de estimativa, ela deve ser validada localmente antes que possa ser usada com confiança.

Document shared on www.docsity.com

- Grupo de subsistemas de infra-estrutura (inclui o subsistema *computer graphics display facilities* — CGDF e subsistema *peripheral control function* — PCF).

Seis casos de uso descrevem o subsistema de interface com o usuário. Cada caso de uso é descrito por não mais do que 10 cenários e tem tamanho médio de seis páginas. O grupo de subsistemas de engenharia é descrito por 10 casos de uso (esses são considerados como estando em um nível mais alto da hierarquia estrutural). Cada um desses casos de uso tem não mais do que 20 cenários associados a ele e tem um tamanho médio de oito páginas. Finalmente, o grupo de subsistemas de infra-estrutura é descrito por cinco casos de uso com uma média de apenas seis cenários e um tamanho médio de cinco páginas.

Usando o relacionamento mencionado na Equação (23-2) com $n = 30\%$, a tabela mostrada na Figura 23.5 é desenvolvida. Considerando a primeira linha da tabela, dados históricos indicam que software IU requer uma média de 800 LOC por caso de uso quando o caso de uso não tem mais do que 12 cenários e é descrito em menos que cinco páginas.

Esses dados estão razoavelmente de acordo com o sistema CAD. Assim, a estimativa de LOC para o subsistema de interface com o usuário é calculado usando a Equação (23-2). Usando a mesma abordagem, estimativas são feitas tanto para os grupos de subsistemas de engenharia quanto de infra-estrutura. A Figura 23.5 resume a estimativa e indica que o tamanho global do software CAD é estimado em 42.500 LOC.

Usando 620 LOC/pm como a média de produtividade para sistemas desse tipo e uma taxa de trabalho com encargos de 8.000 dólares por mês, o custo por linha de código é aproximadamente de 13 dólares. Baseado na estimativa de casos de uso e dados históricos de produtividade, o custo total estimado do projeto é de 552.000 dólares e o esforço estimado é de 68 pessoas-mês.

23.6.9 Acomodação das Estimativas

As técnicas de estimativa discutidas nas seções anteriores resultam em estimativas múltiplas que devem ser acomodadas para produzir uma única estimativa de esforço, duração de projeto ou custo. Para ilustrar esse procedimento de acomodação, vamos considerar, novamente, o software CAD introduzido na Seção 23.6.3.

> "Métodos complicados podem não produzir uma estimativa mais precisa, particularmente quando os desenvolvedores podem incorporar sua própria intuição na estimativa."
>
> **Philip Johnson et al.**

O esforço total estimado para o software CAD varia entre um mínimo de 46 pessoas-mês (derivado usando uma abordagem de estimativa baseada em processo) para um máximo de 68 pessoas-mês (derivada com estimativa de caso de uso). A estimativa média (usando todas as quatro abordagens) é de 56 pessoas-mês. A variação da estimativa média é aproximadamente 18% do lado mais baixo e 21% do lado mais alto.

O que acontece quando a concordância entre as estimativas é baixa? A resposta para essa questão requer uma reavaliação da informação usada para fazer as estimativas. Estimativas altamente divergentes podem com freqüência ser atribuídas a uma de duas causas:

1. O escopo do projeto não está entendido adequadamente ou tem sido mal interpretado pelo planejador.

FIGURA 23.5 Estimativa de caso de uso

	casos de uso	cenários	páginas	cenários	páginas	LOC	LOC estimado
Subsistema de interface com o usuário	6	10	6	12	5	560	3,366
Grupo de subsistemas de engenharia	10	20	8	16	8	3100	31,233
Grupo de subsistemas de infra-estrutura	5	6	5	10	6	1650	7,970
Total estimado de LOC							42,568

2. Dados de produtividade usados para as técnicas de estimativa baseadas no problema são inadequados para a aplicação, obsoletos (no sentido de que não mais refletem precisamente a organização de engenharia de software) ou foram mal aplicados.

O planejador deve determinar a causa da divergência e depois acomodar as estimativas.

INFO

Técnicas Automatizadas de Estimativa de Projetos de Software

Ferramentas automatizadas de estimativa permitem ao planejador estimar o custo e o esforço e realizar análises "e – se" para variáveis importantes de projetos tais como a data de entrega ou a equipe. Embora existam muitas ferramentas automatizadas (ver moldura mais adiante neste capítulo), todas exibem as mesmas características gerais, e realizam as seis funções genéricas seguintes [JON96]:

1. *Dimensionamento dos produtos de projeto.* O "tamanho" de um ou mais produtos de trabalho de software é estimado. Produtos de trabalho incluem a representação externa do software (por exemplo, telas, relatórios), o software em si (por exemplo, KLOC), a funcionalidade entregue (por exemplo, pontos por função) e informação descritiva (por exemplo, documentos).
2. *Seleção das atividades de projeto.* O arcabouço de processo adequado é selecionado e o conjunto de tarefas de engenharia de software é especificado.
3. *Previsão dos níveis da equipe.* O número de pessoas que vão estar disponíveis para fazer o trabalho é especificado. Como o relacionamento entre pessoas disponíveis e trabalho (esforço previsto) é altamente não-linear, essa é uma entrada importante.
4. *Previsão do esforço de software.* Ferramentas de estimativa usam um ou mais modelos (Seção 23.7) que relacionam o tamanho das coisas passíveis de entrega do projetos ao esforço necessário para produzi-las.
5. *Previsão do custo de software.* Dados os resultados do passo 4, custos podem ser calculados pela alocação de taxas de trabalho às atividades de projeto mencionadas no Passo 2.
6. *Previsão de cronogramas de software.* Quando esforço, nível de equipe e atividades de projeto são conhecidas, um rascunho de cronograma pode ser produzido alocando trabalho às atividades de engenharia de software com base nos modelos recomendados de distribuição de esforço, discutidos no Capítulo 24.

Quando diferentes ferramentas de estimativa são aplicadas aos mesmos dados de projeto, uma variação relativamente grande de resultados estimados pode ser encontrada. Mais importante, valores previstos algumas vezes são significativamente diferentes dos valores reais. Isso reforça a noção de que a saída das ferramentas de estimativa devem ser usadas como um "ponto de dados" a partir do qual estimativas são derivadas — não como a única fonte para a estimativa.

23.7 MODELOS DE ESTIMATIVA EMPÍRICOS

PONTO CHAVE

Um modelo de estimativa reflete a população de projetos da qual foi derivado. Assim, o modelo é sensível ao domínio.

O modelo de estimativa para software de computador usa fórmulas derivadas empiricamente para prever o esforço como função de LOC ou FP[9]. Valores de LOC ou FP são estimados usando a abordagem descrita nas Seções 23.6.3 e 23.6.4. Mas, em vez de usar as tabelas descritas naquelas seções, os valores resultantes de LOC ou FP são colocados em um modelo de estimativa.

Os dados empíricos que apóiam a maioria dos modelos de estimativa são derivados de uma amostra limitada de projetos. Por esse motivo, nenhum modelo de estimativa é adequado a todas as classes de software e a todos os ambientes de desenvolvimento. Assim, os resultados obtidos de tais modelos devem ser usados cuidadosamente.

Um modelo de estimativa deve ser calibrado para refletir as condições locais. O modelo deve ser testado aplicando os dados coletados de projetos já finalizados, ligando os dados ao modelo, e depois comparando os resultados reais com os previstos. Se a concordância for baixa, o modelo deverá ser sintonizado e retestado antes que possa ser usado.

9 Um modelo empírico usando casos de uso como variável independente é sugerido na Seção 23.6.7. No entanto, relativamente poucos têm aparecido na literatura até hoje.

Document shared on www.docsity.com

23.7.1 Estrutura dos Modelos de Estimativa

Um modelo de estimativa típico é derivado usando análise de regressão de dados coletados em projetos de software anteriores. A estrutura geral de tais modelos toma a forma [MAT94]

$$E = A + B \times (ev)^C \qquad (23\text{-}3)$$

em que A, B e C são constantes derivadas empiricamente, E é o esforço em pessoas-mês e e_v é a variável de estimativa (LOC ou PF). Além da relação mencionada na Equação (23-2), a maioria dos modelos de estimativa tem alguma forma de componente de ajuste ao projeto, que permite que E seja adequado por outras características do projeto (por exemplo, complexidade do problema, experiência da equipe, ambiente de desenvolvimento). Entre os vários modelos de estimativa orientados a LOC propostos na literatura estão

AVISO

Nenhum desses modelos deve ser usado sem calibração cuidadosa para o seu ambiente.

$E = 5{,}2 \times (KLOC)^{0,91}$	modelo de Walston-Felix
$E = 5{,}5 + 0{,}73 \times (KLOC)^{1,16}$	modelo de Bailey-Basili
$E = 3{,}2 \times (KLOC)^{1,05}$	modelo de Boehm simples
$E = 5{,}288 \times (KLOC)^{1,047}$	modelo de Doty para KLOC > 9

Modelos orientados a FP também foram propostos. Esses incluem

$E = -91{,}4 + 0{,}355\ FP$	modelo de Albrecht e Gaffney
$E = -37 + 0{,}96\ FP$	modelo de Kemerer
$E = -12{,}88 + 0{,}405\ FP$	modelo de regressão para pequenos projetos

Um breve exame desses modelos indica que cada um vai produzir resultado diferente para valores iguais de LOC ou FP. A implicação é clara. Os modelos de estimativa devem ser calibrados para as necessidades locais!

Veja na Web

Informação detalhada sobre COCOMO II, inclusive software que pode ser baixado da Web, pode ser obtida em sunset.usc.edu/research/COCOMOII/cocomo_main.html.

23.7.2 O Modelo COCOMO II

Em seu livro clássico sobre a "economia da engenharia de software", Barry Boehm [BOE81] introduziu uma hierarquia de modelos de estimativa de software denominada COCOMO (*COnstructive COst MOdel*), que significa modelo construtivo de custo. O modelo COCOMO original tornou-se um dos modelos de estimativa de custo de software mais amplamente usados e discutidos na indústria. Evoluiu para um modelo de estimativa mais abrangente, chamado COCOMO II [BOE96, BOE00]. Como seu predecessor, o COCOMO II é na verdade uma hierarquia de modelos de estimativa que tratam das seguintes áreas:

- *Modelo de composição da aplicação.* Usado durante os primeiros estágios da engenharia de software, quando a prototipagem das interfaces com o usuário, a consideração da interação do software com o sistema, a avaliação do desempenho e o julgamento da maturidade tecnológica são de extrema importância.
- *Modelo do primeiro estágio de projeto.* Usado após os requisitos terem sido estabilizados e a arquitetura básica do software ter sido estabelecida.
- *Modelo para o estágio após a arquitetura.* Usado durante a construção do software.

Como todos os modelos para estimativa de software, os modelos COCOMO II requerem informação de tamanho. Três diferentes opções de dimensionamento estão disponíveis como parte da hierarquia de modelos: pontos por objeto, pontos por função e linhas de código-fonte.

O modelo de composição da aplicação do COCOMO II usa pontos por objeto — uma medida indireta de software, que é calculada usando-se a contagem da quantidade de (1) telas (na interface com o usuário), de (2) relatórios e de (3) componentes, que serão provavelmente necessários para construir a aplicação. Cada instância de objeto (por exemplo, tela ou relatório) é classificada em um de três níveis de complexidade (por exemplo, simples, médio ou difícil), usando critérios sugeridos por Boehm [BOE96]. Essencialmente, a complexidade é função da quantidade e fonte das tabelas de

? O que é um "ponto por objeto"?

dados do cliente e do servidor, que são necessárias para gerar a tela ou relatório, e da quantidade de visões ou seções apresentadas como parte da tela ou relatório.

Uma vez determinada a complexidade, a quantidade de telas, relatórios e componentes é ponderada de acordo com a tabela ilustrada na Figura 23.6. A contagem de pontos por objeto é então determinada multiplicando a quantidade original de instâncias do objeto pelo fator de ponderação da figura e somando para obter a contagem total de pontos por objeto. Quando o desenvolvimento baseado em componentes ou reuso geral de software tiver que ser aplicado, a porcentagem de reuso (% de reuso) é estimada e a contagem de pontos por objeto é ajustada:

NOP = (pontos por objeto) × [(100 – % de reuso)/100]

em que NOP é definido como novos pontos por objeto.

Para derivar uma estimativa de esforço, com base no valor NOP calculado, uma "taxa de produtividade" precisa ser derivada. A Figura 23.7 apresenta a taxa de produtividade.

PROD = NOP/pessoa-mês

para diferentes níveis de experiência do desenvolvedor e de maturidade do ambiente de desenvolvimento. Uma vez determinada a taxa de produtividade, a estimativa do esforço do projeto pode ser derivada como

esforço estimado = NOP/PROD

Em modelos COCOMO II mais avançados[10], uma variedade de fatores de escala, direcionadores de custo e procedimentos de ajuste são necessários. O leitor interessado deve ver [BOE00] ou visitar o site Web COCOMO II.

23.7.3 A Equação de Software

A equação de software [PUT92] é um modelo multivariado, que considera uma distribuição de esforço específica, ao longo da vida de um projeto de desenvolvimento de software. O modelo foi derivado de dados de produtividade coletados em mais de 4 mil projetos de software contemporâneos. Baseado nesses dados, o modelo de estimativa é da forma

$$E = [LOC \times B^{0,333}/P]^3 \times (1/t^4) \qquad (23\text{-}4)$$

FIGURA 23.6

Um sistema de classificação por esteira rolante (BOE96)

Tipo de Objeto	Peso da complexidade		
	Simples	Médio	Difícil
Tela	1	2	3
Relatório	2	5	8
Componente 3GL			10

FIGURA 23.7 Taxas de produtividade para pontos por objeto (BOE96)

Experiência/capacidade do desenvolvedor	Muito baixa	Baixa	Normal	Alta	Muito alta
Maturidade/capacidade do ambiente	Muito baixa	Baixa	Normal	Alta	Muito alta
PROD	4	7	13	25	50

10 Como mencionado anteriormente, esses modelos usam a contagem FP e KLOC para a variável de tamanho.

Document shared on www.docsity.com

em que

E = esforço em pessoas-mês ou pessoas-ano
t = duração do projeto em meses ou anos
B = "fator de aptidões especiais"[11]
P = "parâmetro de produtividade" que reflete: maturidade global do processo e práticas de gestão, a medida em que boas práticas de engenharia de software são usadas, o nível das linguagens de programação usadas, o estado do ambiente de software, as aptidões e a experiência da equipe de software e a complexidade da aplicação.

Valores típicos poderiam ser $P = 2.000$, para o desenvolvimento de software embutido de tempo real; $P = 10.000$, para software de telecomunicações e de sistemas; $P = 28.000$, para aplicações de sistemas comerciais. O parâmetro de produtividade pode ser derivado para condições locais, usando dados históricos coletados de esforços de desenvolvimento anteriores.

Veja na Web

Informação sobre as ferramentas de estimativa de custo de software, que evoluíram a partir da equação de software, pode ser obtida no endereço www.qsm.com.

É importante notar que a equação do software tem dois parâmetros independentes: (1) uma estimativa de tamanho (em LOC) e (2) uma indicação da duração do projeto em meses ou anos transcorridos.

Para simplificar o processo de estimativa e usar uma forma mais comum para seu modelo de estimativa, Putnam e Myers [PUT92] sugerem um conjunto de equações derivado da equação de software. O tempo de desenvolvimento mínimo é definido como

$$t_{min} = 8{,}14\ (LOC/P)^{0{,}43} \text{ em meses para } t_{min} > 6 \text{ meses} \quad (23\text{-}5a)$$

$$E = 180\ Bt^3 \text{ em pessoas-mês para } E \geq 20 \text{ pessoas-mês} \quad (23\text{-}5b)$$

Note que t na equação (23-5b) é representado em anos.

Usando as equações (23-5), com $P = 12.000$ (valor recomendado para software científico) para o software CAD discutido anteriormente neste capítulo,

$$t_{min} = 8{,}14\ (33.200/12.000)^{0{,}43}$$
$$t_{min} = 12{,}6 \text{ meses corridos}$$
$$E = 180 \times 0{,}28 \times (1{,}05)^3$$
$$E = 58 \text{ pessoas-mês}$$

Os resultados da equação de software comparam-se favoravelmente com as estimativas desenvolvidas na Seção 23.6. Como o modelo COCOMO mencionado na seção anterior, a equação de software evoluiu durante a última década. Uma discussão mais ampla de uma versão estendida dessa abordagem de estimativa pode ser encontrada em [PUT97b].

23.8 Estimativa de Projetos Orientados a Objetos

Vale a pena complementar métodos convencionais de estimativa de custo de software com uma abordagem que foi projetada explicitamente para software OO. Lorenz e Kidd [LOR94] sugerem a seguinte abordagem:

1. Desenvolva estimativas usando decomposição de esforço, análise FP e qualquer outro método que seja aplicável a aplicações convencionais.
2. Usando modelagem de análise orientada a objetos (Capítulo 8), desenvolva casos de uso e determine uma contagem. Reconheça que o número de casos de uso pode modificar-se à medida que o projeto progride.
3. A partir do modelo de análise, determine o número de classes-chave (chamadas *classes de análise* no Capítulo 8).

11 B aumenta lentamente à medida que "cresce a necessidade de integração, teste, garantia de qualidade, documentação e aptidões gerenciais" [PUT92]. Para programas menores (KLOC = 5 a 15), $B = 0{,}16$. Para programas maiores que 70 KLOC, $B = 0{,}39$.

4. Categorize o tipo de interface para a aplicação e desenvolva um multiplicador para as classes de apoio:
   Multiplique o número de classes-chave (passo 3) pelo multiplicador para obter uma estimativa para o número de classes de apoio.

Tipo de Interface	Multiplicador
Não IGI	2,0
Interface com o usuário baseada em texto	2,25
IGI	2,5
IGU complexa	3,0

5. Multiplique o número total de classes (chave+apoio) pelo número médio de unidades de trabalho por classe. Lorenz e Kidd sugerem 15 a 29 pessoas-dia por classe.
6. Faça verificação cruzada da estimativa baseada em classe multiplicando o número médio de unidades de trabalho por caso de uso.

23.9 Técnicas Especializadas de Estimativa

As técnicas de estimativa discutidas nas Seções 23.6, 23.7 e 23.8 podem ser usadas em qualquer projeto de software. No entanto, quando uma equipe de software encontra um projeto de duração extremamente curta (semanas em vez de meses), em que é provável ocorrer uma corrente contínua de modificações, planejamento de projeto em geral e estimativa em particular devem ser abreviados[12]. Nas seções seguintes, examinamos duas técnicas especializadas de estimativa.

23.9.1 Estimativa para Desenvolvimento Ágil

Como os requisitos para um projeto ágil (Capítulo 4) são definidos como um conjunto de cenários de usuário (por exemplo, "histórias" em Programação Extrema), é possível desenvolver uma abordagem de estimativa que seja informal, mas ainda razoavelmente disciplinada e significativa no contexto de planejamento de projeto para cada incremento de software.

Estimativa para projetos ágeis usa uma abordagem de decomposição que engloba os seguintes passos:

? Como as estimativas são desenvolvidas quando um processo ágil é aplicado?

1. Cada cenário de usuário (o equivalente a um minicaso de uso criado no início de um projeto por usuários finais ou outros interessados) é considerado separadamente para finalidades de estimativa.
2. O cenário é decomposto em um conjunto de funções e tarefas de engenharia de software que vão ser necessárias para desenvolvê-lo.

3a. Cada tarefa é estimada separadamente. Nota: estimativa pode ser baseada em dados históricos, em um modelo empírico ou em "experiência".
3b. Alternativamente, o "volume" (tamanho) do cenário pode ser estimado em LOC, FP, ou alguma outra medida orientada a volume (por exemplo, pontos por objeto).
4a. Estimativas para cada tarefa são somadas para criar uma estimativa para o cenário.
4b. Alternativamente, o volume estimado para o cenário é traduzido em esforço usando dados históricos.
5. As estimativas de esforço para todos os cenários que devem ser implementados em um dado incremento de software são somadas para desenvolver a estimativa de esforço para o incremento.

AVISO

No contexto de estimativa para projetos ágeis, "volume" é uma estimativa do tamanho global de um cenário de usuário em LOC ou FP.

Como a duração de projeto necessária para o desenvolvimento de um incremento de software é bastante curta (tipicamente 3-6 semanas), essa abordagem de estimativa serve a duas finalidades: (1) para garantir que o número de cenários a ser incluído no incremento está de acordo com

12 "Abreviados" *não* significa eliminados. Mesmo projetos de curta duração devem ser planejados e estimativa é a base de sólido planejamento.

Document shared on www.docsity.com

os recursos disponíveis, e (2) para estabelecer uma base para alocação de esforço enquanto o incremento é desenvolvido.

23.9.2 Estimativa para Projetos de Engenharia Web

Como mencionamos no Capítulo 16, projetos de engenharia Web freqüentemente adotam o modelo ágil de processo. Uma medida de ponto por função modificada, acoplada aos passos delineados na Seção 23.9.1, pode ser usada para desenvolver uma estimativa para WebApp.

Roetzheim [ROE00] sugere os seguintes valores de domínio da informação quando se adaptam pontos por função (Capítulos 15 e 22) para estimativa de WebApp:

- *Entradas* são cada tela ou formulário de entrada (por exemplo, CGI ou Java), cada tela de manutenção e se você usa uma metáfora de classificador de orelhas (tab notebook) em algum lugar, cada orelha (*tab*).
- *Saídas* são cada página estática da Web, cada script de página dinâmica da Web (por exemplo, ASP, ISAPI ou outro script DHTML), e cada relatório (quer baseado na Web, quer de natureza administrativa).
- *Tabelas* são cada tabela lógica do banco de dados, mas, se você estiver usando XML para armazenar dados em arquivo, cada objeto XML (ou coleção de atributos XML).
- *Interfaces* definem-se como arquivos lógicos (por exemplo, formatos de registros exclusivos) nos nossos limites de saída do sistema.
- *Consultas* são cada interface externamente publicada ou que usa orientação a mensagem. Um exemplo típico são referências externas DCOM ou COM.

Pontos por função (calculados usando os valores do domínio da informação mencionado) são um razoável indicador de volume para uma WebApp.

Mendes e seus colegas [MEN01] sugerem que o volume de uma WebApp é mais bem determinado pela coleta de medidas (chamadas "variáveis de previsão") associadas com a aplicação (por exemplo, contagem de página, contagem de mídia, contagem de funções), suas características de página da Web (por exemplo, complexidade de página, complexidade de ligação, complexidade gráfica), características de mídia (por exemplo, duração de mídia), e características funcionais (por exemplo, tamanho de código, tamanho de código reusado). Essas medidas podem ser usadas para

FERRAMENTAS DE SOFTWARE

Estimativa de Esforço e Custo

Objetivo: O objetivo das ferramentas de estimativa de esforço e custo é fornecer a uma equipe de projeto estimativas do esforço necessário, duração do projeto e custo de modo que atendam às características específicas do projeto em mãos e do ambiente no qual o projeto vai ser construído.

Mecânica: Em geral, ferramentas de estimativa de custo fazem uso de um banco de dados histórico derivado de projetos locais, dados coletados na indústria e um modelo empírico (por exemplo, COCOMO II) que é usado para derivar estimativas de esforço, duração e custo. Características do projeto e do ambiente de desenvolvimento são inseridas, e a ferramenta fornece uma gama de saídas de estimativa.

Ferramentas Representativas[13]

Costar, desenvolvida por Softstar Systems (www.softstarsystems.com), usa o modelo COCOMO II para desenvolver estimativas de software.

Cost Xpert, desenvolvida por Cost Xpert Group, Inc. (www.costxpert.com), integra múltiplos modelos de estimativa e um banco de dados histórico de projetos.

Estimate Professional, desenvolvida por Software Productivity Centre, Inc. (www.spc.com), é baseada em COCOMO II e no modelo SLIM.

Knowledge Plan, desenvolvida por Software Productivity Research (*www.spr.com*), usa entrada de pontos por função como guia principal para um pacote de estimativa completo.

Price S, desenvolvida por Price Systems (*www.pricesystems.com*), é uma das mais antigas e mais amplamente usadas ferramentas de estimativa para projeto de desenvolvimento de software de larga escala.

SEER/SEM, desenvolvida por Galorath Inc., (*www.galorath.com*), fornece capacidade de estimativa abrangente, análise de sensibilidade, avaliação de risco e outras características.

SLIM-Estimate, desenvolvida por QSM (*www.qsm.com*), baseia-se em abrangentes "bases de conhecimento industrial" para fornecer uma "verificação de plausabilidade" para estimativas derivadas usando dados locais.

13 As ferramentas mencionadas aqui não representam uma recomendação, mas em vez disso uma exemplificação de ferramentas dessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

desenvolver modelos empíricos de estimativa para esforço total de projeto, esforço para autoria de página, esforço para autoria de mídia e esforço para scripts. No entando, trabalho adicional precisa ser feito antes que tais modelos sejam usados com confiança.

23.10 A Decisão de Fazer/Comprar

Em muitas áreas de aplicação de software, freqüentemente é mais eficaz em termos de custo adquirir do que desenvolver software de computador. Gerentes de engenharia de software defrontam-se com a decisão de fazer/comprar, que pode ser ainda mais complicada por algumas opções de compra: (1) software de prateleira pode ser adquirido (ou licenciado), (2) componentes de software de "experiência plena" ou "experiência parcial" (ver Seção 23.4.2) podem ser adquiridos e depois modificados e integrados para satisfazer necessidades específicas, ou (3) software pode ser construído sob medida por um fornecedor externo, para satisfazer as especificações do comprador.

Os passos envolvidos na aquisição de software são definidos pela criticalidade do software a ser adquirido e pelo custo final. Na análise final, a decisão fazer/comprar é tomada com base nas seguintes condições: (1) O produto de software vai ficar disponível antes do software desenvolvido internamente? (2) O custo de aquisição mais o custo de personalização serão menores do que o custo de desenvolvimento interno do software? (3) O custo do apoio externo (por exemplo, de um contrato de manutenção) será menor do que o custo de apoio interno? Essas condições aplicam-se a cada uma das opções de aquisição.

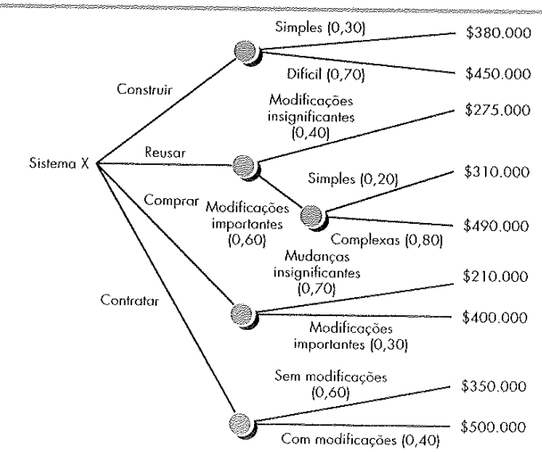
? Há um modo sistemático de organizar as opções associadas com a decisão de fazer/comprar?

23.10.1 Criação de uma Árvore de Decisão

Os passos descritos podem ser ampliados usando técnicas estatísticas tais como a *análise por árvore de decisão* [BOE89]. Por exemplo, a Figura 23.8 mostra uma árvore de decisão para um sistema baseado em software, *X*. Nesse caso, a organização de engenharia de software pode (1) cons-

FIGURA 23.8

Árvore de decisão para apoiar a decisão fazer/comprar



Document shared on www.docsity.com

truir o sistema *X* a partir do zero, (2) reusar componentes existentes de "experiência-parcial" para construir o sistema, (3) comprar um produto de software disponível e modificá-lo para satisfazer as necessidades locais ou (4) contratar o desenvolvimento do software com um fornecedor externo.

Se o sistema tiver que ser construído a partir do zero, então haverá 70% de probabilidade de que o trabalho vai ser difícil. Usando as técnicas de estimativa discutidas antes neste capítulo, o planejador do projeto estima que um esforço de desenvolvimento difícil vai custar 450.000 dólares. Um esforço de desenvolvimento "simples" tem um custo estimado de 380.000 dólares. O valor esperado do custo, calculado ao longo de qualquer ramo da árvore de decisão, é

$$\text{custo esperado} = \Sigma\ (\text{probabilidade do caminho})_i \times (\text{custo estimado do caminho})_i$$

em que *i* é o caminho na árvore de decisão. Para o caminho correspondente a construir,

$$\text{custo esperado}_{\text{de construção}} = 0{,}30\ (\text{U\$380K}) + 0{,}70\ (\text{U\$450K}) = \text{U\$429K}$$

Seguindo outros caminhos na árvore de decisão, os custos projetados para reuso, aquisição e contrato, sob diferentes circunstâncias, são também mostrados. Os custos esperados para esses caminhos são

$$\text{custo esperado}_{\text{reuso}} = 0{,}40\ (\text{U\$275K}) + 0{,}60\ [0{,}20(\text{U\$310K}) + 0{,}80\ (\text{U\$490K})] = \text{U\$382K}$$
$$\text{custo esperado}_{\text{aquisição}} = 0{,}70\ (\text{U\$210K}) + 0{,}30\ (\text{U\$400K})] = \text{U\$267K}$$
$$\text{custo esperado}_{\text{contrato}} = 0{,}60\ (\text{U\$350K}) + 0{,}40\ (\text{U\$500K})] = \text{U\$410K}$$

Com base na probabilidade e nos custos projetados que foram anotados na Figura 23.8, o menor custo esperado é o da opção "compra".

É importante notar, entretanto, que muitos critérios — não apenas custo — devem ser considerados durante o processo de tomada de decisão. A disponibilidade, a experiência do desenvolvedor/vendedor/fornecedor, o atendimento aos requisitos, a "política" local e a probabilidade de modificação são apenas alguns dos critérios que podem afetar a decisão final de construir, reusar, comprar ou contratar.

23.10.2 Terceirização

Mais cedo ou mais tarde, toda empresa que desenvolve software de computador formula uma questão fundamental: "Há algum modo de obter o software e os sistemas de que necessitamos a um preço menor?". A resposta a essa questão não é simples, e a discussão emocional conseqüente à resposta à questão sempre leva a uma única palavra: terceirização.

Conceitualmente, terceirização é bastante simples. Atividades de engenharia de software são contratadas com terceiros, que fazem o trabalho a um custo menor e, espera-se, com qualidade maior. O trabalho de software conduzido dentro da empresa é reduzido a uma atividade[14] de gestão de contrato.

> "Como regra, a terceirização requer gestão ainda mais competente que o desenvolvimento interno."
>
> Steve McConnell

A decisão de terceirizar pode ser estratégica ou tática. No nível estratégico, os gerentes do negócio consideram se uma parcela significativa de todo o trabalho de software pode ser contratada com terceiros. Em nível tático, um gerente de projeto determina se parte ou todo o projeto pode ser mais bem executado subcontratando o trabalho de software.

Independentemente da amplitude do foco, a decisão de terceirizar é com freqüência uma decisão financeira. Uma discussão detalhada da análise financeira da terceirização está além do escopo deste livro e é melhor deixá-la a outros (por exemplo, [MIN95]). Todavia, um breve resumo dos prós e contras da decisão vale a pena.

14 Terceirização pode ser vista mais geralmente como qualquer atividade que leve à aquisição de software ou componentes de software de uma fonte fora da organização de engenharia de software.

O aspecto positivo é que economias de custo podem usualmente ser conseguidas pela redução da quantidade de pessoal de software e das facilidades (por exemplo, computadores, infra-estrutura) que os apóiam. O aspecto negativo é que a empresa perde algum controle sobre o software de que necessita. Como o software é uma tecnologia que faz a diferença em sistemas, serviços e produtos, a empresa corre o risco de colocar o destino da sua competitividade nas mãos de terceiros.

CASASEGURA

Terceirização

A cena: Sala de reunião da Empresa CPI.

Os personagens: Mal Golden, gerente sênior de desenvolvimento de produto; Lee Warren, gerente de engenharia; Joe Camalleri, vice-presidente executivo de desenvolvimento de negócio; Doug Miller, gerente de projeto de engenharia de software.

A conversa:

Joe: Estamos considerando terceirizar parte do produto de engenharia de software *CasaSegura*.

Doug (chocado): Quando isso aconteceu?

Lee: Tivemos uma proposta de um desenvolvedor do exterior. Ela está 30% abaixo do que o seu grupo acredita que vai custar. Aqui está [entrega a proposta a Doug que a lê].

Mal: Como você sabe, Doug, nós estamos tentando manter os custos baixos, e 30% são 30%. Além disso, esse pessoal vem altamente recomendado.

Doug (tomando fôlego e tentanto permanecer calmo): Vocês me pegaram de surpresa nisso, mas antes de tomarem uma decisão final, aceitam alguns comentários?

Joe (concordando com a cabeça): Lógico, vá em frente.

Doug: Nós não trabalhamos com essa empresa de terceirização anteriormente, certo?

Mal: Certo, mas...

Doug: E eles dizem que qualquer mudança na especificação vai ser cobrada a uma taxa adicional, certo?

Joe (franzindo a testa): Certo, mas esperamos que as coisas fiquem razoavelmente estáveis.

Doug: Má suposição, Joe.

Joe: Bem...

Doug: É provável que lancemos novas versões desse produto nos próximos anos e é razoável assumir que o software vai fornecer muitas das novas características, certo?

[Todos concordam com a cabeça.]

Doug: Nós já coordenamos um projeto internacional antes?

Lee (parecendo preocupado): Não, mas me disseram...

Doug (tentando conter sua raiva): Então o que você está me dizendo é: (1) nós estamos prestes a trabalhar com um fornecedor desconhecido, (2) os custos para fazer isso não são tão baixos quanto parecem, (3) nós estamos de fato nos comprometendo a trabalhar com eles durante muitas versões do produto, não importa o que façam na primeira, e (4) vamos ter que aprender no serviço o que se refere a um projeto internacional.

[Todos permanecem em silêncio.]

Doug: Gente... Eu acho que isso é um erro, e eu gostaria que vocês tirassem um dia para reconsiderar. Nós vamos ter muito mais controle se fizermos o trabalho internamente. Nós temos competência e posso garantir que não vai custar muito mais... o risco será menor e eu sei que vocês são todos tão adversos a risco quanto eu.

Joe (franzindo a testa): Você levantou alguns pontos importantes, mas você tem um interesse oculto em manter esse projeto na empresa.

Doug: Isso é verdade, mas não muda os fatos.

Joe (com um suspiro): Está bem, vamos engavetar isso por um ou dois dias, dedicar-lhe um pouco mais de ponderação e nos reunir novamente para uma decisão final. Doug, posso falar com você em particular?

Doug: Lógico... Eu realmente quero estar certo de que vamos fazer a coisa direito.

23.11 Resumo

O planejador do projeto de software deve estimar três coisas antes de iniciar um projeto: quanto tempo ele vai levar, quanto esforço exigirá e quantas pessoas estarão envolvidas. Além disso, o planejador deve estimar os recursos (hardware e software) necessários e o risco envolvido.

A declaração de escopo ajuda o planejador a desenvolver estimativas usando uma ou mais técnicas situadas em duas categorias amplas: decomposição e modelagem empíricas. Técnicas de decomposição exigem um delineamento das principais funções do software, seguido por estimativas de (1) o número de LOC, ou (2) valores selecionados do domínio da informação, ou (3) o número de casos de uso, ou (4) número de pessoas-mês necessário para implementar cada função, ou (5)

Document shared on www.docsity.com

o número de pessoas-mês necessário para cada atividade de engenharia de software. Técnicas empíricas usam expressões derivadas de resultados práticos para tempo e esforço a fim de prever essas quantidades relativamente ao projeto. Ferramentas automatizadas podem ser usadas para implementar um modelo empírico específico.

Estimativas de projeto precisas usam, geralmente, pelo menos duas dessas três técnicas. Pela comparação e conciliação das estimativas derivadas usando diferentes técnicas, é mais provável que o planejador consiga derivar uma estimativa precisa. A estimativa de projetos de software nunca será uma ciência exata, mas uma combinação de bons dados históricos e técnicas sistemáticas pode melhorar a precisão da estimativa.

Referências Bibliográficas

[BEN92] Bennatan, E. M., *Software Project Management: A Practitioner's Approach*, McGraw-Hill, 1992.

[BEN03] Bennatan, E. M., "So What Is the State of Software Estimation?", *The Cutter Edge* (um newsletter on-line), 11 fev. 2003, disponível em http:// www.cutter.com.

[BOE81] Boehm, B., *Software Engineering Economics*, Prentice-Hall, 1981.

[BOE89] Boehm, B., *Risk Management*, IEEE Computer Society Press, 1989.

[BOE96] Boehm, B., "Anchoring the Software Process", IEEE *Software*, v. 13, n. 4, jul. de 1996, p. 73-82.

[BOE00] Boehm, B. et al., *Software Cost Estimation in COCOMO II*, Prentice-Hall, 2000.

[BRO75] Brooks, F., *The Mythical Man-Month*, Addison-Wesley, 1975.

[GAU89] Gause, D. C. e Weinberg, G. M., *Exploring Requirements: Quality Before Design*, Dorset House, 1989.

[HOO91] Hooper, J. e Chester, R. O., *Software Reuse: Guidelines and Methods*, Plenum Press, 1991.

[JON96] Jones, C., "How Software Estimation Tools Work", *American Programmer*, v. 9, n. 7, jul. 1996, p. 19-27.

[LOR94] Lorenz, M. e Kidd, J. *Object-Oriented Software Metrics*, Prentice-Hall, 1994.

[MAT94] Matson, J., Barrett B. e Mellichamp, J., "Software Development Cost Estimation Using Function Points", *IEEE Trans. Software Engineering*, v. SE-20, n. 4, abr. de 1994, p. 275-287.

[MCC98] McConnell, S., *Software Project Survival Guide*, Microsoft Press, 1998.

[MEN01] Mendes, E., Mosley, N. e Counsell, S., "Web Metrics — Estimating Design and Authoring Effort", *IEEE Multimedia*, jan./mar. de 2001, p. 50-57.

[MIN95] Minoli, D., *Analyzing Outsourcing*, McGraw-Hill, 1995.

[PHI98] Phillips, D., *The Software Project Manager's Handbook*, IEEE Computer Society Press, 1998.

[PUT78] Putnam, L., "A General Empirical Solution to the Macro Software Sizing and Estimation Problem", *IEEE Trans. Software Engineering*, v SE-4, n. 4, jul. 1978, p. 345-361.

[PUT92] Putnam, L. e Myers, W., *Measures for Excellence*, Yourdon Press, 1992.

[PUT97a] Putnam, L. e Myers, W., "How Solved Is the Cost Estimation Problem?", *IEEE Software*, nov. 1997, p. 105-107.

[PUT97b] Putnam, L. e Myers, W., *Industrial Strength Software: Effective Management Using Measurement*, IEEE Computer Society Press, 1997.

[ROE00] Roetzheim, W., "Estimating Internet Development", *Software Development*, ago. 2000, disponível em http://www.sdmagazine.com/documents/s_741/ sdm0008d/0008d.htm.

[SMI99] Smith, J., "The Estimation of Effort Based on Use Cases", Rational Software Corp., 1999, disponível em http://www.rational.com/media/whitepapers/ finalTP171.PDF.

Problemas e Pontos a Considerar

23.1. Imagine que você é o gerente de projeto de uma empresa que constrói software para robôs domésticos. Você foi contratado para construir o software de um robô que corta a grama para o dono da casa. Redija uma declaração de escopo que descreva o software. Certifique-se de que sua declaração de escopo é delimitada. Se você não está familiarizado com robôs faça um pouco de pesquisa antes de começar a escrever. Também declare seus pressupostos sobre o hardware que vai ser necessário. *Alternativa:* substitua o robô de cortar grama por outro problema de robótica que seja de seu interesse.

23.2. A complexidade de projetos de software influencia a precisão da estimativa. Desenvolva uma lista de características de software (por exemplo, operação concorrente, saída gráfica) que afetam a complexidade de um projeto. Estabeleça prioridades na lista.

23.3. Desempenho é uma consideração importante durante o planejamento. Discuta como o desempenho pode ser interpretado diferentemente, dependendo da área de aplicação do software.

23.4. Faça uma decomposição funcional do software robô que você descreveu no Problema 23.1. Estime o tamanho de cada função em LOC. Considerando que sua organização produz 450 LOC/pm, com uma taxa bruta de mão-de-obra de 7.000 dólares por pessoa-mês, estime o esforço e o custo necessários para construir o software usando a técnica de estimativa baseada em LOC, descrita neste capítulo.

23.5. Use o modelo COCOMO II para estimar o esforço necessário para construir software para um terminal de atendimento bancário (*automatic teller machine* — ATM) simples que produz 12 telas, 10 relatórios e que vai exigir aproximadamente 80 componentes de software. Considere complexidade média e maturidade média do desenvolvedor/ambiente. Use o modelo de composição da aplicação com pontos por objeto.

23.6. Use a equação de software para estimar o software do robô de cortar grama do Problema 23.1. Considere que as Equações (23-5) são aplicáveis e que $P = 8.000$.

23.7. Compare as estimativas de esforço derivadas nos Problemas 23.4 e 23.6. Qual é o desvio-padrão e como ele afeta seu grau de certeza a respeito da estimativa?

23.8. Usando os resultados obtidos no Problema 23.7, determine se é razoável esperar que o software possa ser construído dentro dos próximos seis meses e quantas pessoas seriam necessárias para executar o trabalho.

23.9. Desenvolva um modelo de planilha que implemente uma ou mais técnicas de estimativa descritas neste capítulo. Alternativamente, obtenha um ou mais modelos on-line para estimativa de projeto de software em fontes baseadas na Web.

23.10. Para uma equipe de projeto: desenvolva uma ferramenta de software que implemente cada uma das técnicas de estimativa desenvolvidas neste capítulo.

23.11. Parece estranho que estimativas de custo e cronograma sejam desenvolvidas durante o planejamento de projetos de software — antes que a análise detalhada dos requisitos de software ou projeto tenha sido conduzida. Por que acha que isso é feito? Há circunstâncias em que isso não deveria ser feito?

23.12. Recalcule os valores esperados, anotados na árvore de decisão da Figura 23.8, considerando que cada ramo tem uma probabilidade de 50-50. Isso iria modificar sua decisão final?

Leituras e Fontes de Informação Adicionais

A maioria dos livros sobre gerência de projetos de software contém discussões de estimativa de projetos. The Project Management Institute (*PMBOK Guide*, PMI, 2001), Wysoki e seus colegas (*Effective Project Management*, Wiley, 2000), Lewis (*Project Planning Scheduling and Control*, 3. ed., McGraw-Hill, 2000), Bennatan (*On Time, Within Budget: Software Project Management Practices and Techniques*, 3. ed., Wiley, 2000) e Phillips [PHI98] fornecem diretrizes úteis de estimativas.

Jones (*Estimating Software Costs*, McGraw-Hill, 1998) escreveu uma das mais abrangentes abordagens do assunto publicadas até hoje. Seu livro contém modelos e dados que são aplicáveis à estimativa de software em todo domínio de aplicação. Coombs (*IT Project Estimation*, Cambridge University Press, 2002), Roetzheim e Beasley (*Software Project Cost and Schedule Estimating: Best Practices*, Prentice-Hall, 1997) e Wellman (*Software Costing*, Prentice-Hall, 1992) apresentam muitos modelos úteis e sugerem diretrizes passo a passo para gerar as melhores estimativas possíveis.

O detalhado tratamento de estimativa de custo de software de Putnam e Myer ([PUT92] e [PUT97b]) e os livros de Boehm sobre economia da engenharia de software ([BOE81] e COCOMO II [BOE00]) descrevem modelos empíricos de estimativa. Esses livros fornecem análise de dados detalhada derivada de centenas de projetos de software. Um excelente livro de DeMarco (*Controlling Software Projects*, Yourdon Press, 1982) fornece conhecimento valioso sobre a gestão, medição e estimativa de projetos de software. Lorenz e Kidd (*Object-Oriented Software Metrics*, Prentice-Hall, 1994) e Cockburn (*Surviving Object-Oriented Projects*, Addison-Wesley, 1998) consideram estimativa de sistemas orientados a objetos.

Uma ampla variedade de fontes de informação sobre estimativa de software está disponível na Internet. Uma lista atualizada de referências da World Wide Web pode ser encontrada no site deste livro: **http://www.mhhe.com/pressman**.

Document shared on www.docsity.com

CAPÍTULO 24

CRONOGRAMAÇÃO DE PROJETO DE SOFTWARE

CONCEITOS-CHAVE

acompanhamento 555
atrasos 545
curva PNR 548
distribuição de esforço 549
encaixotamento do tempo . 556
gráficos de tempo 554
pessoal e esforço 556
princípios básicos 542
rede de tarefas 552
refinamento de tarefas ... 551
subdivisão de trabalho 553
valor agregado 557

No fim dos anos 60 um jovem engenheiro de olhar brilhante foi escolhido para "escrever" um programa de computador para uma aplicação de fabricação automatizada. O motivo dessa seleção era simples. Ele era a única pessoa do seu grupo técnico que assistiu a um seminário sobre programação de computador. Sabia detalhes da linguagem de máquina e de FORTRAN, mas não sabia nada sobre engenharia de software e ainda menos sobre cronogramação e acompanhamento de projeto.

Seu chefe deu-lhe os manuais adequados e uma descrição verbal do que tinha de ser feito. Foi informado de que o projeto deveria ser completado em dois meses.

Leu os manuais, considerou sua abordagem e começou a escrever códigos. Depois de duas semanas, o chefe chamou-o a sua sala e perguntou como as coisas estavam caminhando.

"Muito bem", disse o jovem engenheiro com o entusiasmo da juventude. "É muito mais simples do que pensei, já fiz praticamente 75% do trabalho."

O chefe sorriu e encorajou o jovem engenheiro a continuar o seu bom trabalho. Planejaram encontrar-se novamente em uma semana.

Uma semana depois o chefe chamou o engenheiro a sua sala e perguntou: "Em que pé estamos?".

"Está tudo indo bem", disse o jovem, "mas tive algumas pequenas dificuldades, que pretendo superar e voltar aos trilhos brevemente."

"E a data de entrega como fica?", o chefe perguntou.

"Nenhum problema", disse o engenheiro. "Estou perto de completar 90% do trabalho."

PANORAMA

O que é? Você selecionou um modelo de processo adequado, identificou as tarefas de engenharia de software que têm que ser realizadas, estimou a quantidade de trabalho e o número de pessoas; conhece o prazo, até considerou os riscos. Agora é hora de fazer as ligações. Isto é, você tem que criar uma rede de tarefas de engenharia de software que vai lhe permitir fazer o serviço no prazo. Uma vez criada a rede, você tem que designar um responsável para cada tarefa, certificar-se de que seja executada e adaptar a rede à medida que os riscos se tornam reais. Resumidamente, isso é a cronogramação e o acompanhamento de projeto.

Quem faz? No nível de projeto, os gerentes de projeto de software, usando informação obtida dos engenheiros de software. No nível individual, os próprios engenheiros de software.

Por que é importante? Para construir um sistema complexo, muitas tarefas de engenharia de software ocorrem em paralelo e o resultado do trabalho realizado ao longo de uma tarefa pode ter profundo efeito no que está sendo realizado em outra tarefa. Essas interdependências são muito difíceis de entender sem um cronograma. É virtualmente impossível, também, avaliar o progresso em um projeto de software de tamanho moderado a grande sem um cronograma detalhado.

Quais são os passos? As tarefas de engenharia de software ditadas pelo modelo do processo de software são refinadas de acordo com a funcionalidade a ser construída. A cada tarefa são atribuídos um esforço e uma duração e uma rede de tarefas (também chamada de "rede de atividades") é criada de modo que permita que a equipe de software satisfaça a data de entrega estabelecida.

Qual é o produto do trabalho? São produzidos o cronograma do projeto e a informação correlacionada.

Como tenho certeza de que fiz corretamente? Cronogramação adequada exige que (1) todas as tarefas figurem na rede, (2) esforço e duração sejam inteligentemente atribuídos a cada tarefa, (3) as interdependências entre tarefas sejam adequadamente indicadas, (4) recursos sejam alocados ao trabalho a ser feito e (5) marcos de referência pouco espaçados sejam estabelecidos de modo que o progresso possa ser acompanhado.

Se você trabalha no mundo do software há alguns anos, pode terminar a história. Não será surpresa para você que o jovem engenheiro[1] tenha ficado nos 90% durante o resto do prazo e terminou apenas um mês depois (com ajuda de outros).

Essa história tem se repetido dezenas de milhares de vezes por desenvolvedores de software durante as últimas quatro décadas. A grande questão é por quê?

24.1 CONCEITOS BÁSICOS

Apesar de haver muitas razões pelas quais o software é entregue atrasado, a maioria pode ser rastreada para uma ou mais das seguintes causas básicas:

- Data de entrega irrealística estabelecida por alguém de fora do grupo de engenharia de software e imposta a gerentes e profissionais do grupo.
- Mudanças nos requisitos do cliente não refletidas em mudanças do cronograma.
- Subestimativa honesta da quantidade de esforço e/ou quantidade de recursos que serão necessários para fazer o serviço.
- Riscos previsíveis e/ou imprevisíveis que não foram considerados quando o projeto teve início.
- Dificuldades técnicas que não puderam ser previstas a tempo.
- Dificuldades humanas que não puderam ser previstas a tempo.
- Falta de comunicação entre a equipe de projeto, que resultou em atrasos.
- Falha da gerência do projeto em reconhecer que o projeto estava sofrendo atraso e falta de ação para corrigir o problema.

> "Cronogramas excessivos ou irracionais são provavelmente a influência individual mais destrutiva em todo o software."
>
> Capers Jones

Datas de entrega agressivas (leia-se "impraticáveis") são o cotidiano nos negócios de software. Algumas vezes essas datas de entrega são exigidas por motivos legítimos, do ponto de vista da pessoa que as estabelece. Mas, o bom senso diz que a legitimidade também deve ser percebida pelo pessoal que faz o trabalho.

Napoleão disse certa vez: "Qualquer comandante-em-chefe que aceita cumprir um plano que considera falho incorre em erro; ele deve apresentar suas razões, insistir para que o plano seja modificado e finalmente solicitar seu afastamento em vez de ser instrumento da derrota de seu exército". Essas são palavras fortes que muitos gerentes de projeto de software deveriam ponderar.

As atividades de estimativa discutidas no Capítulo 23 e as técnicas de cronogramação descritas neste capítulo são freqüentemente implementadas dentro da restrição de uma data de entrega preestabelecida. Se estimativas mais cuidadosas indicam que a data de entrega é impraticável, um gerente de projeto competente deve "proteger sua equipe de pressão indevida [de cronograma]... [e] devolver a pressão aos que a originaram" [PAG85].

Para ilustrar, considere que um grupo de desenvolvimento de software foi solicitado a construir um sistema de controle em tempo real, para um instrumento de diagnóstico médico, que deve ser introduzido no mercado em nove meses. Depois de cuidadosa estimativa e análise de risco (Capítulo 25), o gerente do projeto de software chega à conclusão de que o software, tal como solicitado, vai exigir 14 meses corridos para ser criado com o pessoal disponível. Como deve proceder o gerente do projeto?

1 Se você está pensando que essa história é autobiográfica, acertou!

Document shared on www.docsity.com

> "Adoro datas de entrega. Gosto do zunido que elas fazem quando passam voando."
>
> **Douglas Adams**

É impraticável ir até o escritório do cliente (nesse caso o cliente provável é o pessoal de marketing/vendas) e exigir que a data de entrega seja modificada. Pressões externas de mercado determinaram a data e o produto precisa ser entregue. É igualmente ruim recusar fazer o trabalho (do ponto de vista de fazer carreira). Assim, o que fazer?

Os seguintes passos são recomendados nessa situação:

? O que devemos fazer quando a gerência exige que cumpramos uma data de entrega impraticável?

1. Faça uma estimativa detalhada usando dados históricos de projetos anteriores. Determine o esforço e a duração estimados para o projeto.
2. Usando um modelo incremental de processo (Capítulo 3), desenvolva uma estratégia de engenharia de software que entregue a funcionalidade crítica na data de entrega imposta, mas adie a funcionalidade restante para depois. Documente o plano.
3. Reúna-se com o cliente e (usando a estimativa detalhada) explique por que a data de entrega imposta é impraticável. Certifique-se de frisar que todas as estimativas estão baseadas no desempenho de projetos anteriores. Certifique-se também de indicar o aperfeiçoamento percentual que seria necessário para conseguir cumprir a data de entrega atualmente fixada[2]. O seguinte comentário é apropriado:
   "Acho que tenho um problema com a data de entrega do software de controle XYZ. Dei a cada um de vocês um resumo das taxas de produção obtidas em projetos anteriores e uma estimativa que fizemos de diferentes modos. Vocês perceberão que eu considerei um aperfeiçoamento de 20% nas taxas de produção anteriores, mas ainda assim chegamos a uma data de entrega para daqui a 14 meses, em vez dos 9 meses estabelecidos".
4. Ofereça uma estratégia de desenvolvimento incremental como alternativa:
   "Temos algumas opções e gostaria que vocês tomassem uma decisão baseada nelas. Primeiro, podemos aumentar o orçamento e trazer recursos adicionais em uma tentativa de fazer o serviço em nove meses. Mas entendam que isso vai aumentar o risco de má qualidade, por causa do prazo exíguo[3]. Em segundo lugar, podemos remover algumas funções e capacidades do software exigidas. Isso vai tornar a primeira versão do produto um pouco menos funcional, mas podemos anunciar toda a funcionalidade e depois entregá-la dentro de 14 meses. Em terceiro lugar, podemos ignorar a realidade e esperar completar o projeto em 9 meses. Vamos acabar entregando nada ao cliente. A terceira opção, espero que concordem, é inaceitável. O histórico passado e as nossas melhores estimativas dizem que, além de uma receita para o desastre, é irrealística".

Vai haver resmungos, mas se são apresentadas estimativas sólidas, com base em bons dados históricos, é provável que versões negociadas das opções 1 ou 2 sejam escolhidas. A data de entrega irrealística se evapora.

24.2 Cronogramação de Projeto

Fred Brooks, o conhecido autor de *The Mythical Man-Month* [BRO95], foi perguntado certa vez sobre como os projetos de software sofriam atrasos. Sua resposta foi tão simples quanto profunda: "Um dia de cada vez".

A realidade de um projeto técnico (quer envolva construir uma usina hidrelétrica ou desenvolver um sistema operacional) é que centenas de pequenas tarefas devem ser executadas para conseguir uma meta maior. Algumas dessas tarefas situam-se fora do fluxo principal e podem ser completadas sem preocupação com o impacto na data de finalização do projeto. Outras tarefas situam-se no

2 Se a porcentagem de aperfeiçoamento for de 10 a 25%, pode ser possível executar o serviço. Todavia, mais provavelmente, a porcentagem de aperfeiçoamento do desempenho da equipe precisa ser maior que 50%. Essa é uma expectativa irrealística.

3 Você pode também acrescentar que a incorporação de mais pessoal não reduz proporcionalmente o prazo.

AVISO

As tarefas necessárias para atingir o objetivo do gerente de projeto não devem ser executadas manualmente. Há excelentes ferramentas de cronogramação de projetos. Use-as.

caminho "crítico". Se essas tarefas "críticas" sofrerem atraso, a data de entrega do projeto, como um todo, é ameaçada.

O objetivo do gerente de projeto é definir todas as tarefas do projeto, construir uma rede que mostre suas interdependências, identificar as tarefas que são críticas nessa rede e depois acompanhar o seu progresso para certificar-se de que atrasos sejam reconhecidos "um dia de cada vez". Para conseguir isso, o gerente deve dispor de um cronograma que tenha sido definido em um grau de resolução que permita monitorar o progresso e controlar o projeto.

A cronogramação de projeto de software é uma atividade que distribui o esforço estimado pela duração planejada do projeto, partilhando esse esforço por tarefas específicas de engenharia de software. É importante notar, todavia, que o cronograma evolui com o tempo. Durante os primeiros estágios de planejamento do projeto, um cronograma macroscópico é desenvolvido. Esse tipo de cronograma identifica todas as principais atividades de arcabouço do processo e as funções do produto a que se aplicam. À medida que o projeto deslancha, cada entrada no cronograma macroscópico é refinada em um cronograma detalhado. Nele, tarefas de software específicas (necessárias para executar uma atividade) são identificadas e cronogramadas.

> "Cronogramação superotimista não resulta em cronogramas reais mais curtos, e sim mais longos."
>
> **Steve McConnell**

A cronogramação de projetos de engenharia de software pode ser vista de duas perspectivas bem diferentes. Na primeira, uma data final para entrega do sistema baseado em computador já foi estabelecida (irrevogavelmente). A organização de software está obrigada a distribuir o esforço dentro do espaço de tempo prescrito. A segunda situação de cronogramação de software considera que limites cronológicos aproximados foram discutidos, mas que a data final é estabelecida pela organização de engenharia de software. O esforço é distribuído para fazer o melhor uso dos recursos e uma data final é definida depois de cuidadosa análise do software. Infelizmente, a primeira situação é encontrada muito mais freqüentemente que a segunda.

24.2.1 Princípios Básicos

Como outras áreas de engenharia de software, alguns princípios básicos orientam a cronogramação de projetos de software:

Compartimentalização. O projeto deve ser compartimentalizado em um certo número de atividades e tarefas gerenciáveis. Para conseguir a compartimentalização, tanto o produto quanto o processo são decompostos.

Interdependência. A interdependência de cada atividade, ação ou tarefa compartimentalizada deve ser determinada. Algumas tarefas devem ocorrer em seqüência, enquanto outras podem ocorrer em paralelo. Algumas atividades ou ações não podem começar até que o trabalho produzido por outra esteja disponível. Outras atividades ou ações podem ocorrer independentemente.

PONTO CHAVE

Quando você desenvolver um cronograma, compartimentalize o trabalho, represente as interdependências entre as tarefas, atribua esforço e tempo a cada tarefa, defina as responsabilidades, resultados e marcos de referência.

Atribuição de tempo. A cada tarefa a ser cronogramada deve ser atribuído um certo número de unidades de trabalho (por exemplo, pessoas-dia de esforço). Além disso, a cada tarefa devem ser atribuídas datas de início e término, que são funções das interdependências e de o trabalho ser conduzido em tempo integral ou parcial.

Validação do esforço. Cada projeto tem um número definido de membros da equipe de software. À medida que a atribuição de tempo ocorre, o gerente de projeto deve garantir que não mais do que o número alocado de pessoas seja cronogramado em um determinado momento. Por exemplo, considere um projeto que tem três engenheiros de software na equipe (por exemplo, três pessoas-dia estão disponíveis por dia de esforço atribuído)[4]. Em um certo dia, sete tarefas simultâneas devem ser executadas. Cada tarefa requer 0,5 pessoa-dia de esforço. Foi alocado mais esforço do que pessoas disponíveis para fazer o trabalho.

4 Na realidade, menos do que três pessoas-dia de esforço estão disponíveis por causa de reuniões não relacionadas ao projeto, doenças, férias e várias outras razões. Para nossa finalidade, todavia, consideramos 100% de disponibilidade.

Document shared on www.docsity.com

Responsabilidades definidas. Cada tarefa cronogramada deve ser atribuída a um membro específico da equipe.

Resultados definidos. Cada tarefa cronogramada deve ter um resultado definido. Para projetos de software, o resultado é normalmente um produto de trabalho (por exemplo, o projeto de um módulo) ou parte dele. Produtos de trabalho são freqüentemente combinados em produtos prontos para entrega.

Marcos de referência definidos. Cada tarefa ou grupo de tarefas deve ser associado a um marco de referência do projeto. Um marco de referência é atingido quando um ou mais produtos do trabalho tiverem sido revisados quanto à qualidade (Capítulo 26) e tiverem sido aprovados.

Cada um desses princípios é aplicado à medida que o cronograma do projeto evolui.

24.2.2 O Relacionamento entre Pessoal e Esforço

AVISO

Se você precisa agregar pessoal a um projeto atrasado, certifique-se de atribuir-lhes trabalho altamente compartimentalizado.

Em um projeto pequeno de desenvolvimento de software, uma única pessoa pode analisar os requisitos, fazer o projeto, gerar o código e conduzir os testes. À medida que o tamanho do projeto aumenta, mais pessoas precisam ser envolvidas. (Raramente podemos nos dar ao luxo de atacar um esforço de dez pessoas-ano com uma pessoa trabalhando durante dez anos!)

Há um mito comum que ainda se perpetua entre muitos gerentes responsáveis por esforço de desenvolvimento de software: "Se ficarmos atrasados no cronograma, poderemos sempre agregar mais programadores e recuperar depois o atraso no projeto". Infelizmente, agregar pessoal nas últimas fases de um projeto tem freqüentemente um efeito danoso, causando atrasos ainda maiores no cronograma. O pessoal agregado precisa aprender o sistema, e os que irão treiná-los são os que estavam fazendo o trabalho. Enquanto ensinam não podem fazer o trabalho e o projeto atrasa ainda mais.

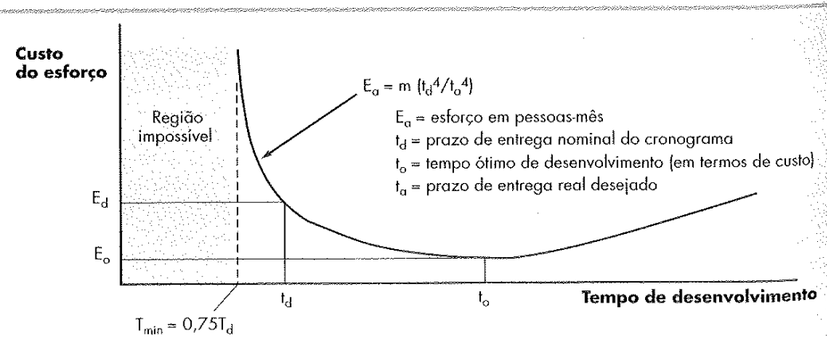
Além do tempo que leva para aprender o sistema, o pessoal adicional aumenta a quantidade de canais de comunicação e a complexidade da comunicação em todo o projeto. Apesar de a comunicação ser absolutamente essencial para o desenvolvimento de software bem-sucedido, cada novo canal de comunicação requer esforço adicional e conseqüentemente tempo adicional.

Ao longo dos anos, dados empíricos e análise teórica demonstram que cronogramas de projeto são elásticos. Isto é, é possível comprimir a data de entrega, de um determinado projeto (acrescentando recursos adicionais) em alguma quantidade. É também possível estender uma data de entrega (reduzindo o número de recursos).

A *Curva Putnam-Norden-Rayleigh (PNR)*[5] fornece uma indicação do relacionamento entre o esforço aplicado e o prazo de entrega em um projeto de software. Uma versão da curva, representando o esforço do projeto como função do prazo de entrega, é mostrada na Figura 24.1. A curva indica um valor mínimo, t_o, que indica o mínimo custo de tempo para entrega (isto é, o prazo de entrega que vai resultar no mínimo de esforço empregado). À medida que nos movemos à esquerda de t_o (isto é, à medida que tentamos acelerar a entrega), a curva sobe não linearmente.

FIGURA 24.1

O relacionamento entre esforço e prazo de entrega



5 A pesquisa original pode ser encontrada em [NOR70] e [PUT78].

PONTO CHAVE

Se a entrega pode ser adiada, a curva PNR indica que os custos de projeto podem ser substancialmente reduzidos.

AVISO

À medida que o prazo de entrega do projeto torna-se mais e mais apertado, você atinge um ponto no qual o trabalho não pode ser completado a tempo independentemente do número de pessoas que está fazendo o trabalho. Encare a realidade e defina uma nova data de entrega.

Como exemplo, consideramos que uma equipe de projeto tenha estimado que um nível de esforço, E_a, será necessário para conseguir um prazo de entrega nominal t_d que é ótimo em termos de cronograma e recursos disponíveis. Embora seja possível acelerar a entrega, a curva sobe mais abruptamente à esquerda de t_d. De fato, a curva PNR indica que o prazo de entrega do projeto não pode ser comprimido muito além de 0,75 de t_d. Se tentamos compressão adicional, o projeto entra na "região impossível" e o risco de falha torna-se muito alto. A curva PNR também indica que a opção de custo de entrega mais baixa $t_o = 2\ t_d$. A implicação aqui é que o adiamento da entrega do projeto pode reduzir os custos significativamente. Naturalmente, isso precisa ser ponderado contra os custos de negócios associados com o adiamento.

A equação de software [PUT92] introduzida no Capítulo 23 é derivada da curva PNR e demonstra o relacionamento altamente não linear entre o tempo cronológico para completar um projeto e o esforço humano aplicado ao projeto. A quantidade de linhas de código entregues (comandos fontes), L, é relacionada ao esforço e tempo de desenvolvimento pela equação:

$$L = P \times E^{1/3}\ t^{4/3}$$

em que E é o esforço de desenvolvimento em pessoas-mês, P é um parâmetro de produtividade que reflete uma variedade de fatores que levam a trabalho de engenharia de software de alta qualidade (valores típicos de P variam entre 2.000 e 12.000) e t é a duração do projeto em meses de calendário.

Rearranjando essa equação de software, podemos chegar a uma expressão para o esforço de desenvolvimento E:

$$E = L^3/(P^3t^4) \qquad (24\text{-}1)$$

em que E é o esforço despendido (em pessoas-ano) ao longo de todo o ciclo de vida para o desenvolvimento e manutenção de software e t é o tempo de desenvolvimento em anos. A equação para o esforço de desenvolvimento pode ser relacionada ao custo do desenvolvimento pela inclusão de um fator bruto de taxa de trabalho ($/pessoa-ano).

Isso leva a alguns resultados interessantes. Considere um projeto de software complexo de tempo real estimado em 33.000 LOC e 12 pessoas-ano de esforço. Se oito pessoas são designadas para a equipe de projeto, o projeto pode ser completado em aproximadamente 1,3 ano. Se, todavia, estendermos o prazo para 1,75 ano, a natureza altamente não-linear do modelo descrito na Equação (24-1) nos dá:

$$E = L^3/(P^3t^4) \sim 3{,}8 \text{ pessoas-ano.}$$

Estendendo o prazo de entrega em 6 meses, isso implica que, estendendo a data final por 6 meses, podemos reduzir a quantidade de pessoal de oito para quatro! A validade de tais resultados está aberta a discussão, mas a implicação é clara: benefícios podem ser obtidos usando menos pessoas durante um período maior para alcançar o mesmo objetivo.

24.2.3 Distribuição de Esforço

? *Como o esforço deve ser distribuído ao longo do fluxo de trabalho do processo de software?*

Cada uma das técnicas de estimativa de projetos de software, discutidas no Capítulo 23, leva a estimativas das unidades de trabalho (por exemplo, pessoas-mês) necessárias para completar um desenvolvimento de software. Uma distribuição de esforço recomendada ao longo do processo de software é freqüentemente referida como regra *40-20-40.* Quarenta por cento de todo o esforço é reservado para a análise e projeto iniciais. Uma porcentagem análoga é aplicada ao teste final. Você pode inferir corretamente que a codificação (20% do esforço) é desenfatizada.

Essa distribuição de esforço deve ser usada apenas como diretriz[6]. As características de cada projeto devem determinar a distribuição de esforço. O trabalho despendido no planejamento do projeto raramente chega a mais do que 2% a 3% do esforço, a menos que o plano comprometa a

6 Hoje em dia a regra 40-20-40 está sendo atacada. Alguns acreditam que mais de 40% de todo o esforço deve ser gasto durante análise e projeto. Por outro lado, alguns proponentes de desenvolvimento ágil (Capítulo 4) alegam que menos tempo deve gasto "inicialmente" e que uma equipe deve passar rapidamente para a construção.

Document shared on www.docsity.com

organização a grandes despesas com alto risco. A análise de requisitos pode abranger de 10 a 25% do esforço do projeto. O esforço despendido na análise ou prototipagem deve crescer em proporção direta com o tamanho e complexidade do projeto. Uma faixa de 20 a 25% do esforço é normalmente aplicada ao projeto de software. O tempo despendido na revisão do projeto e interação subseqüente também deve ser considerado.

Tendo em vista o esforço aplicado ao projeto de software, a codificação deve vir depois com relativamente pouca dificuldade. Uma faixa de 15 a 20% do esforço total pode ser suficiente. O teste e depuração subseqüentes podem usar de 30 a 40% do esforço de desenvolvimento de software. A criticalidade do software freqüentemente determina a quantidade de teste necessária. Se o software envolve pessoas (por exemplo, falha do software pode resultar em perda de vida), percentagens mais altas são até comuns.

24.3 Definição de um Conjunto de Tarefas para o Projeto de Software

Diversos modelos de processo foram descritos na Parte I deste livro. Independentemente de a equipe de software escolher um paradigma seqüencial linear, um modelo incremental, um modelo evolutivo ou alguma permutação deles, o modelo de processo é preenchido por um conjunto de tarefas que permitem à equipe de software definir, desenvolver e finalmente apoiar o software de computador.

Nenhum conjunto único de tarefas é adequado para todos os projetos. O conjunto de tarefas que seria adequado para um sistema grande e complexo seria considerado excessivo para um produto de software pequeno e relativamente simples. Assim, um processo de software efetivo deve definir uma coleção de conjuntos de tarefas, cada um projetado para satisfazer as necessidades de diferentes tipos de projeto.

Como mencionamos no Capítulo 2, um conjunto de tarefas é uma coleção de tarefas de trabalho de engenharia de software, marcos de referência e produtos de trabalho que precisam ser realizados para completar um projeto específico. O conjunto de tarefas deve fornecer a disciplina necessária para alcançar alta qualidade de software. Mas, ao mesmo tempo, não deve sobrecarregar a equipe de projeto com trabalho desnecessário.

Para desenvolver um cronograma de projeto, um conjunto de tarefas precisa ser distribuído pelo prazo do projeto. O conjunto de tarefas vai variar dependendo do tipo de projeto e do grau de rigor com o qual a equipe de software decide fazer o seu trabalho. Apesar de ser difícil desenvolver uma taxonomia abrangente de tipos de projeto de software, a maioria das organizações de software encontra os seguintes projetos:

1. *Projetos de desenvolvimento conceitual* que são iniciados para explorar algum conceito novo de negócio ou a aplicação de alguma tecnologia nova.
2. *Projetos de desenvolvimento de novas aplicações* que são efetuados como conseqüência de um pedido específico de cliente.
3. *Projetos de aperfeiçoamento de aplicações* que ocorrem quando o software existente passa por modificações importantes na função, desempenho ou interfaces que são observáveis pelo usuário final.
4. *Projetos de manutenção de aplicações* que corrigem, adaptam ou ampliam o software existente em aspectos que não são imediatamente óbvios ao usuário final.
5. *Projetos de reengenharia* que são efetuados com o objetivo de reconstruir um sistema existente (legado) no seu todo ou em parte.

Veja na Web

Um modelo adaptável de projeto (adaptable process model — APM) foi desenvolvido para ajudar a definição de conjuntos de tarefa para diferentes projetos de software. Uma descrição completa do APM pode ser encontrada em www.rspa.com/apm.

Mesmo em um único tipo de projeto, muitos fatores influenciam o conjunto de tarefas a ser escolhido. Entre eles estão [PRE99]: tamanho do projeto, número de usuários em potencial, criticalidade da missão, longevidade da aplicação, estabilidade dos requisitos, facilidade de comunicação cliente/desenvolvedor, maturidade da tecnologia aplicável, restrições de desempenho, características embutidas e não embutidas, equipe de projeto e fatores de reengenharia. Quando considerados em combinação, esses fatores fornecem uma indicação do grau de rigor com o qual o processo de software deve ser aplicado.

24.3.1 Um Exemplo de Conjunto de Tarefas

Cada um dos tipos de projeto descritos pode ser abordado usando um modelo de processo que é seqüencial linear, iterativo (por exemplo, modelos de prototipação ou incremental), ou evolutivos (por exemplo, o modelo espiral). Em alguns casos, um tipo de projeto flui suavemente para o seguinte. Por exemplo, projetos de desenvolvimento conceitual que são bem-sucedidos, freqüentemente, evoluem para novos projetos de desenvolvimento de aplicação. Quando um novo projeto de desenvolvimento de aplicação termina, algumas vezes se inicia um projeto de aperfeiçoamento da aplicação. Essa progressão é tanto natural quanto previsível e vai ocorrer independentemente do modelo de processo que é adotado por uma organização. Assim, as principais tarefas de engenharia de software descritas nas seções seguintes são aplicáveis a todos os fluxos de modelo de processo. Como exemplo, consideramos as tarefas de engenharia de software para um projeto de desenvolvimento conceitual. Projetos de desenvolvimento conceitual são iniciados quando o potencial para alguma tecnologia nova precisa ser explorado. Não há certeza de que a tecnologia vai ser aplicada, mas um cliente (por exemplo, marketing) acredita que existe benefício potencial. Projetos de desenvolvimento conceitual são abordados pela aplicação das seguintes tarefas principais:

1.1 Determinação de escopo conceitual determina o escopo global do projeto.
1.2 Planejamento conceitual preliminar estabelece a capacidade de a organização assumir o trabalho implicado pelo escopo do projeto.
1.3 Avaliação do risco da tecnologia avalia o risco associado à tecnologia a ser implementada como parte do escopo do projeto.
1.4 Prova de conceito demonstra a viabilidade de uma nova tecnologia no contexto de software.
1.5 Implementação conceitual implementa a representação conceitual de um modo que pode ser revisado por um cliente e é usada para finalidade de "marketing" quando um conceito precisa ser vendido a outros clientes ou gerentes.
1.6 Reação do cliente ao conceito exige realimentação sobre um novo conceito de tecnologia e visa aplicações específicas do cliente.

Uma análise rápida dessas tarefas deve produzir poucas surpresas. De fato, o fluxo de engenharia de software em projetos de desenvolvimento conceitual (e também para todos os outros tipos de projeto) é pouco mais do que bom senso.

24.3.2 Refinamento das Tarefas Principais

As principais tarefas descritas na seção anterior podem ser usadas para definir um cronograma macroscópico de um projeto. No entanto, o cronograma macroscópico precisa ser refinado para criar um cronograma detalhado de projeto. O refinamento começa tomando cada tarefa principal e decompondo-a em um conjunto de subtarefas (com produtos de trabalho relacionados e marcos de referência).

Como exemplo da decomposição de tarefas, considere a Tarefa 1.1, Escopo Conceitual. O refinamento da tarefa pode ser realizado usando um formato de esboço, mas, neste livro, uma abordagem de linguagem de projeto de processo é usada para ilustrar o fluxo da atividade de estabelecimento de escopo conceitual.

Definição de Tarefa: Tarefa 1.1 Determinação de escopo conceitual
1.1.1 Identificar necessidades, benefícios e potenciais clientes;
1.1.2 Definir saída/controle desejados e eventos de entrada que guiam a aplicação;
Início Tarefa 1.1.2
1.1.2.1 FTR: Revisar a descrição de necessidades escrita[7]
1.1.2.2 Derivar uma lista de saídas/entradas visíveis ao cliente
1.1.2.3 FTR: Revisar saídas/entradas com o cliente e revisar quando necessário;

7 FTR indica que uma revisão técnica formal (*formal technical review*) (Capítulo 26) deve ser conduzida.

Document shared on www.docsity.com

fim de tarefa Tarefa 1.1.2

1.1.3 Definir a funcionalidade/comportamento de cada função principal;

Início Tarefa 1.1.3

1.1.3.1 FTR: Revisar objetos de dados de saída e entrada derivados na tarefa 1.1.2;

1.1.3.2 Derivar um modelo de funções/comportamentos;

1.1.3.3 FTR: Revisar funções/comportamentos com cliente e revisar quando necessário;

fim de tarefa Tarefa 1.1.3

1.1.4 Isolar os elementos da tecnologia a ser implementados em software;

1.1.5 Pesquisar a disponibilidade de software existente;

1.1.6 Definir viabilidade técnica;

1.1.7 Fazer estimativa rápida de tamanho;

1.1.8 Criar uma Definição de Escopo;

fim da definição de tarefa: Tarefa 1.1

As tarefas e subtarefas mencionadas no refinamento em linguagem de projeto de processo formam a base para um cronograma detalhado para a atividade de estabelecimento de escopo conceitual.

24.4 Definição de uma Rede de Tarefas

PONTO CHAVE

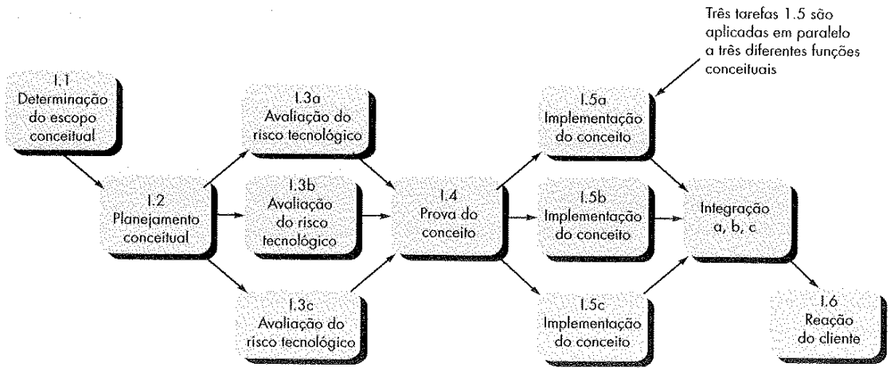
A rede de tarefas é um mecanismo útil para mostrar as dependências entre tarefas e determinar o caminho crítico.

Tarefas e subtarefas individuais têm interdependências baseadas na sua seqüência. Além disso, quando mais de uma pessoa está envolvida em um projeto de engenharia de software, é provável que atividades e tarefas do desenvolvimento sejam realizadas em paralelo. Quando isso ocorre, as tarefas simultâneas devem ser coordenadas, de modo que sejam completadas quando tarefas posteriores precisarem do(s) produto(s) do seu trabalho.

Uma *rede de tarefas*, também chamada de *rede de atividades*, é uma representação gráfica do fluxo de tarefas de um projeto. É algumas vezes usada como um mecanismo por meio do qual a seqüência e dependências de tarefas são alimentadas em uma ferramenta de cronogramação automática de projeto. Em sua forma mais simples (usada quando se cria um cronograma macroscópico), a rede de tarefas mostra as principais tarefas de engenharia de software. A Figura 24.2 mostra uma rede de tarefas esquemática, para um projeto de desenvolvimento conceitual.

A natureza concorrente das atividades de engenharia de software leva a vários requisitos de cronogramação importantes. Como as tarefas paralelas ocorrem de maneira assíncrona, o planejador

FIGURA 24.2 Uma rede de tarefas para desenvolvimento conceitual



deve determinar as dependências entre tarefas para garantir o progresso contínuo em direção ao término. Além disso, o gerente do projeto deve estar ciente das tarefas que ficam no caminho crítico. Isto é, tarefas que precisam ser completadas a tempo se todo o projeto tiver que ser completado a tempo. Esses tópicos são discutidos em mais detalhes posteriormente neste capítulo.

É importante notar que a rede de tarefas mostrada na Figura 24.2 é macroscópica. Em uma rede de tarefas detalhada (precursora de um cronograma detalhado), cada atividade mostrada na figura deveria ser expandida. Por exemplo, a Tarefa 1.1 deve ser expandida para mostrar todas as tarefas detalhadas no refinamento da Tarefa 1.1, mostrada na Seção 24.3.2.

24.5 Cronogramação

A cronogramação de um projeto de software não difere muito da cronogramação de qualquer esforço de engenharia multitarefa. Assim, ferramentas e técnicas gerais de cronogramação de projetos podem ser aplicadas com pouca modificação a projetos de software.

Técnica de avaliação e revisão de programação (Program Evaluation and Review Technique — PERT) e *método do caminho crítico* (*Critical Path Method* — CPM) são dois métodos de cronogramação de projetos que podem ser aplicados ao desenvolvimento de software. Ambas as técnicas são guiadas por informação já desenvolvida em atividades iniciais de planejamento de projetos:

- Estimativas de esforço.
- Uma decomposição da função do produto.
- Seleção de modelo de processo e conjunto de tarefas adequado.
- Decomposição de tarefas.

As interdependências entre tarefas podem ser definidas usando uma rede de tarefas. As tarefas, algumas vezes chamadas *estrutura de subdivisão do trabalho* (*work breakdown structure* — WBS) do projeto, são definidas para todo o produto ou para funções individuais.

> "Tudo o que temos que decidir é o que fazer com o tempo que nos é dado."
>
> Gandalf em *The Lord of the Rings: Fellowship of the Ring*

Tanto PERT quanto CPM fornecem ferramentas quantitativas que permitem ao planejador do software (1) determinar o *caminho crítico* — cadeia de tarefas que determina a duração do projeto; (2) estabelecer estimativas de tempo "mais prováveis", para tarefas individuais, aplicando modelos estatísticos; e (3) calcular "limites de tempo" que definem uma "janela" de tempo para uma tarefa específica.

FERRAMENTAS DE SOFTWARE

Cronogramação de Projeto

Objetivo: O objetivo de ferramentas de cronogramação de projeto é permitir ao gerente do projeto definir tarefas de trabalho, estabelecer suas dependências, associar recursos humanos a tarefas e desenvolver uma variedade de gráficos, diagramas e tabelas que ajudam a acompanhar e controlar o projeto de software.

Mecânica: Em geral, ferramentas de cronogramação de projeto requerem a especificação de uma estrutura de subdivisão de trabalho ou a geração de uma rede de tarefas. Uma vez definida a subdivisão de tarefas (um esboço) ou a rede, as datas de início e fim, recursos humanos, prazos de entrega fixados e outros dados são atribuídos a cada tarefa. A ferramenta gera então uma variedade de diagramas de tempo e outras tabelas que permitem ao gerente avaliar o fluxo de tarefas de um projeto. Esses dados podem ser atualizados continuamente à medida que o projeto é conduzido.

Document shared on www.docsity.com

Ferramentas Representativas[8]

AMS Realtime, desenvolvida por Advanced Management Systems (www.amsusa.com), fornece capacidade de cronogramação de projetos de todos os tamanhos e tipo.

Microsoft Project, desenvolvida por Microsoft (*www.microsoft.com*), é a ferramenta de cronogramação de projeto baseada em PC mais amplamente usada.

Viewpoint, desenvolvida por Artemis Internation Solutions Corp. (www.atemispm.com), apóia todos os aspectos do planejamento de projeto inclusive cronogramação.

Uma lista abrangente de fornecedores de software de gestão de projeto e produtos pode ser encontrada em *www.infogoal.com/pmc/pmcswr.htm*.

24.5.1 Gráficos de Tempo

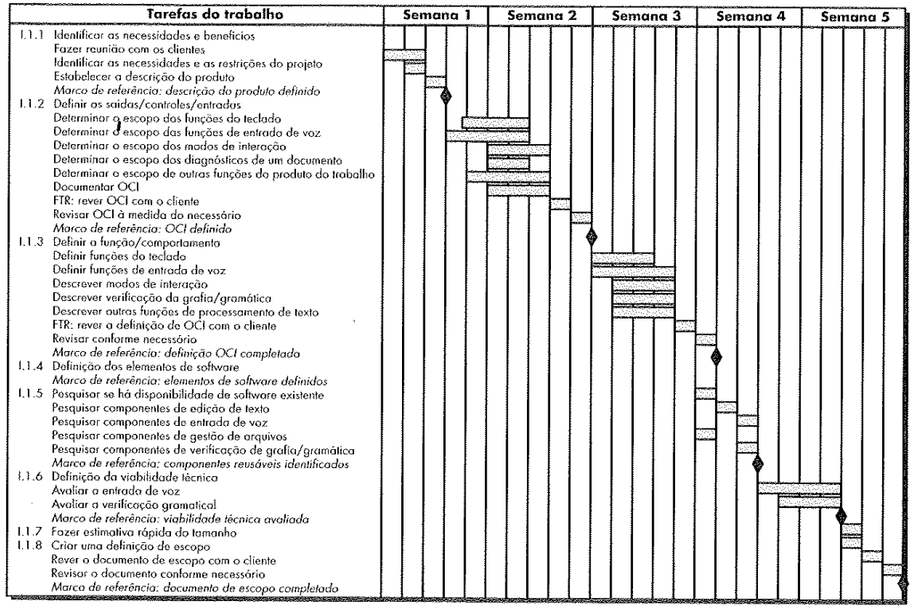
PONTO CHAVE

Um gráfico de tempo permite a você determinar que tarefas serão conduzidas em um determinado momento.

Ao criar um cronograma de um projeto de software, o planejador começa com um conjunto de tarefas (a estrutura de subdivisão do trabalho). Se ferramentas automáticas forem usadas, a subdivisão do trabalho é introduzida como uma rede de tarefas ou como um esboço das tarefas. Dá-se entrada então no esforço, duração e data de início para cada tarefa. Além disso, as tarefas podem ser atribuídas a indivíduos específicos.

Como conseqüência dessa entrada, um *gráfico de tempo*, também chamado *gráfico de Gantt*, é gerado. Um gráfico de tempo pode ser desenvolvido para todo o projeto. Alternativamente, gráficos separados podem ser desenvolvidos para cada função do projeto ou para cada indivíduo que trabalha no projeto. A Figura 24.3 ilustra o formato de um gráfico de tempo. Ela mostra uma parte do cronograma do projeto de software, que enfatiza a tarefa de determinação do escopo conceitual para um produto de software de processamento de texto (*Word Processing — WP*). Todas as tarefas

FIGURA 24.3 Um exemplo de gráfico de tempo



8 As ferramentas mencionadas aqui não representam uma recomendação, mas em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

FIGURA 24.4 Um exemplo de tabela de projeto

Tarefas de trabalho	Início planejado	Início real	Término planejado	Término real	Pessoal designado	Esforço alocado	Notas
1.1.1 Identificar necessidades e benefícios							A determinação do escopo vai exigir mais esforço/tempo
Reunir com os clientes	semana 1, dia 1	semana 1, dia 1	semana 1, dia 2	semana 1, dia 2	BLS	2 p-d	
Identificar as necessidades e restrições do projeto	semana 1, dia 2	semana 1, dia 2	semana 1, dia 2	semana 1, dia 2	JPP	1 p-d	
Estabelecer a descrição do produto	semana 1, dia 3	semana 1, dia 3	semana 1, dia 3	semana 1, dia 3	BLS/JPP	1 p-d	
Marco de referência: descrição do produto definido	semana 1, dia 3	semana 1, dia 3	semana 1, dia 3	semana 1, dia 3			
1.1.2 Defina a OCI desejada							
Determinar o escopo das funções de teclado	semana 1, dia 4	semana 1, dia 4	semana 2, dia 2		BLS	1,5 p-d	
Determinar o escopo das funções de entrada de voz	semana 1, dia 3	semana 1, dia 3	semana 2, dia 2		JPP	2 p-d	
Determinar o escopo dos modos de interação	semana 2, dia 1		semana 2, dia 3		MLL	1 p-d	
Determinar o escopo dos diagnósticos de um documento	semana 2, dia 1		semana 2, dia 2		BLS	1,5 p-d	
Determinar o escopo de outras funções de processamento de texto	semana 1, dia 4	semana 1, dia 4	semana 2, dia 3		JPP	2 p-d	
Documentar OCI	semana 2, dia 1		semana 2, dia 3		MLL	3 p-d	
FTR: Rever OCI com o cliente	semana 2, dia 3		semana 2, dia 3		todos	3 p-d	
Revisar OCI se necessário	semana 2, dia 4		semana 2, dia 4		todos	3 p-d	
Marco de referência: OCI definido	semana 2, dia 5		semana 2, dia 5				
1.1.3 Definir a função/comportamento							

de projeto (para a determinação de escopo conceitual) são listadas na coluna mais à esquerda. As barras horizontais indicam a duração de cada tarefa. Quando várias barras ocorrem ao mesmo tempo, a concorrência de tarefas fica implícita. Os losangos indicam marcos de referência.

Quando tiver sido dada entrada na informação necessária para a geração de um gráfico de tempo, a maioria das ferramentas de cronogramação de projetos de software produz *tabelas de projeto* — uma listagem tabular de todas as tarefas do projeto, de suas datas iniciais e finais, planejadas e reais, e várias outras informações relacionadas (Figura 24.4). Usadas em conjunto com os gráficos de tempo, as tabelas de projeto permitem ao gerente do projeto acompanhar o progresso.

24.5.2 Acompanhamento do Cronograma

O cronograma do projeto fornece um roteiro para um gerente de projeto de software. Se tiver sido adequadamente desenvolvido, o cronograma do projeto define as tarefas e os marcos de referências que devem ser acompanhados e controlados à medida que o projeto prossegue. O acompanhamento pode ser conseguido de vários modos diferentes:

- Conduzir reuniões periódicas sobre o estado do projeto nas quais cada membro da equipe relata o progresso e os problemas.
- Avaliar os resultados de todas as revisões conduzidas ao longo do processo de engenharia de software.
- Determinar se os marcos de referência formais do projeto (os losangos mostrados na Figura 24.3) foram atingidos na data prevista.
- Comparar a data de início real com a data de início planejada para cada tarefa do projeto listada na tabela de recursos (Figura 24.4).
- Reunir informalmente os profissionais para obter sua avaliação subjetiva do progresso alcançado e dos problemas que estão aparecendo.
- Usando a análise de valor agregado (Seção 24.6), avaliar o progresso quantitativamente.

Na realidade, todas essas técnicas de acompanhamento são usadas por gerentes de projeto experientes.

> "A regra básica do relato do estado de um software pode ser resumida em uma única frase: 'Sem surpresas!'."
>
> Capers Jones

Document shared on www.docsity.com

AVISO

O melhor indicador do progresso é o término e a revisão bem-sucedida de um produto definido do trabalho de software.

Controle é empregado por um gerente de projeto de software para administrar os recursos do projeto, lidar com os problemas e dirigir a equipe de projeto. Se as coisas estão indo bem (por exemplo, o projeto está em dia e dentro do orçamento, as revisões indicam que um progresso real está sendo conseguido e os marcos de referência alcançados), o controle é leve. Mas quando ocorrem problemas, o gerente de projeto deve exercer controle para resolvê-los o mais cedo possível. Depois que um problema tiver sido diagnosticado, recursos adicionais podem ser concentrados na área do problema: o pessoal pode ser redistribuído ou o cronograma do projeto pode ser redefinido.

Quando deparam com pressão excessiva para cumprimento de prazo, gerentes de projeto experientes usam algumas vezes uma técnica de cronogramação e controle de projetos chamada *encaixotamento do tempo* [ZAH95]. A estratégia de encaixotamento do tempo reconhece que o produto completo pode não ser entregue no prazo de entrega predefinido. Assim, um paradigma incremental de software (Capítulo 3) é adotado e um cronograma é derivado para cada entrega incremental.

As tarefas associadas com cada incremento são então encaixotadas em termos de tempo. Isso significa que o cronograma para cada tarefa é ajustado retroagindo a partir da data de entrega para o incremento. Uma "caixa" é colocada em volta de cada tarefa. Quando uma tarefa atinge o limite da caixa de tempo (mais ou menos 10%), o trabalho é interrompido e a tarefa seguinte começa.

A reação inicial à abordagem de encaixotamento do tempo é freqüentemente negativa: "Se o trabalho não está finalizado, como podemos prosseguir?". A resposta está no modo pelo qual o trabalho é efetuado. Na época em que o limite da caixa de tempo é alcançado, é provável que 90% da tarefa tenha sido completada[9]. Os restantes 10%, ainda que importantes, podem (1) ser adiados até o próximo incremento ou (2) ser completados depois, se necessário. Ao invés de ficar "empacado" em uma tarefa, o projeto prossegue em direção à data de entrega.

24.5.3 Acompanhamento do Progresso de um Projeto OO

Embora um modelo iterativo seja o melhor arcabouço para um projeto OO, paralelismo de tarefas torna o acompanhamento do projeto difícil. O gerente de projeto pode ter dificuldade em estabelecer marcos de referência significativos para um projeto OO porque um certo número de coisas diferentes está acontecendo simultaneamente. Em geral, os seguintes marcos de referências principais podem ser considerados "completados" quando o critério mencionado tiver sido satisfeito.

Marco de referência técnico: análise OO completada

- Todas as classes e a hierarquia de classes foram definidas e revisadas.
- Atributos de classe e operações associadas à uma classe foram definidos e revisados.
- Relacionamento de classes (Capítulo 8) foram estabelecidos e revisados.
- Um modelo comportamental (Capítulo 8) foi criado e revisado.
- Classes reusáveis foram anotadas.

Marco de referência técnico: projeto OO completado

- O conjunto de subsistemas (Capítulo 9) foi definido e revisado.
- Classes foram alocadas a subsistemas e revisadas.
- A alocação de tarefas foi estabelecida e revisada.
- Responsabilidades e colaborações (Capítulos 8 e 9) foram identificadas.
- Classes de projeto foram criadas e revisadas.
- O modelo de comunicação foi criado e revisado.

9 Um cínico poderia lembrar o provérbio: "Os primeiros 90% de um sistema tomam 90% do tempo. Os 10% restantes do sistema tomam 90% do tempo".

AVISO

Depuração e teste ocorrem de acordo um com o outro. O estado da depuração é freqüentemente avaliado, considerando o tipo e quantidade de erros "em aberto" (bugs).

Marco de referência técnico: programação OO completada

- Cada nova classe foi implementada em código a partir do modelo de projeto.
- Classes extraídas (de uma biblioteca de reuso) foram implementadas.
- Protótipo ou incremento foi construído.

Marco de referência técnico: teste OO

- A correção e completez dos modelos de análise e projeto OO foram revisadas.
- A rede de responsabilidade-colaboração de classes (Capítulo 8) foi desenvolvida e revisada.
- Casos de teste foram projetados e os testes intraclasse (Capítulo 14) foram conduzidos para cada classe.
- Casos de teste foram projetados, teste agregado (Capítulo 14) foi completado e as classes estão integradas.
- Testes de sistema foram completados.

Lembrando que o modelo de processo OO é iterativo, cada um desses marcos de referência pode ser revisitado à medida que diferentes incrementos são entregues ao cliente.

CASASEGURA

Acompanhamento do Cronograma

A cena: Escritório de Doug Miller, antes do início do projeto de software do *CasaSegura*.

Os personagens: Doug Miller (gerente da equipe de engenharia de sofware do *CasaSegura*) e Vinod Raman, Jamie Lazar, e outros membros da equipe de engenharia de software do produto.

A conversa:

Doug (olhando um slide de Powerpoint): O cronograma para o primeiro incremento do *CasaSegura* parece razoável, mas vamos ter problema para acompanhar o progresso.

Vinod (um olhar preocupado em seu rosto): Por quê? Nós temos as tarefas cronogramadas dia a dia, bastante produtos de trabalho e estamos seguros de que não alocamos recursos demais.

Doug: Tudo certo, mas como sabemos quando o modelo de análise do primeiro incremento fica pronto?

Jamie: As coisas são iterativas, assim isso é difícil.

Doug: Entendo isso, mas... bem, por exemplo, veja as *classes de análise definidas*.Você indicou isso como um marco de referência.

Vinod: É verdade.

Doug: Quem faz essa determinação?

Jamie (irritada): Elas ficam prontas quando ficam prontas.

Doug: Isso não é suficiente, Jamie. Nós temos que cronogramar FTRs [*formal technical reviews*, Capítulo 26], e você não fez isso. O término bem-sucedido de uma revisão do modelo de análise, por exemplo, é um marco de referência razoável. Entendeu?

Jamie (franzindo a testa): Certo, de volta à prancheta.

Doug: Não deve levar mais do que uma hora para fazer as correções... todos os outros podem começar agora.

24.6 ANÁLISE DO VALOR AGREGADO

PONTO CHAVE

O valor agregado fornece uma indicação quantitativa do progresso.

Na Seção 24.5 discutimos várias abordagens qualitativas para o acompanhamento de projeto. Cada uma fornece ao gerente de projeto uma indicação do progresso, mas a avaliação da informação fornecida é um tanto subjetiva. É razoável perguntar se há uma técnica quantitativa para avaliar o progresso à medida que a equipe de software progride nas tarefas de trabalho atribuídas ao cronograma do projeto. Na realidade, existe realmente uma técnica para realizar a análise quantitativa do progresso. É chamada *análise de valor agregado* (*Earned Value Analysis* — EVA). Humphrey [HUM95] discute o valor agregado da seguinte maneira:

Document shared on www.docsity.com

> O sistema de valor agregado fornece uma escala comum de valor para cada tarefa [do projeto de software], independentemente do tipo de trabalho que está sendo realizado. O total de horas para fazer todo o projeto é estimado e a cada tarefa é atribuído um valor agregado, com base na sua porcentagem estimada do total.

Dito ainda mais simplesmente, valor agregado é uma medida de progresso. Ela nos permite avaliar a "porcentagem de execução" de um projeto usando análise quantitativa, em vez de confiar em um sentimento. De fato, Fleming e Koppleman [FLE98] alegam que a análise de valor agregado "fornece medidas de desempenho precisas e confiáveis a partir do momento em que 15% do projeto já foi completado".

Como devo calcular o valor agregado para avaliar o progresso?

Para determinar o valor agregado, os seguintes passos são realizados:

1. O *custo orçado do trabalho cronogramado* (*Budgeted Cost of Work Scheduled* — BCWS) é determinado, para cada tarefa de trabalho representada no cronograma. Durante a atividade de estimativa, o trabalho (em pessoas-hora ou pessoas-dia) de cada tarefa de engenharia de software é planejado. Assim, $BCWS_i$ é o esforço planejado para a tarefa de trabalho *i*. Para determinar o progresso em um determinado ponto do cronograma do projeto, o valor de BCWS é a soma dos valores $BCSW_i$ de todas as tarefas de trabalho, que deveriam ter sido completadas até aquele momento no cronograma do projeto.
2. Os valores BCWS de todas as tarefas de trabalho são somados para derivar o orçamento na conclusão (*Budget At Completion* — BAC). Assim,

   $BAC = \Sigma (BCWS_k)$ para todas as tarefas *k*

3. Em seguida, o valor do *custo orçado do trabalho realizado* (*Budgeted Cost of Work Performed* — BCWP) é calculado. O valor de BCWP é a soma dos valores BCWS de todas as tarefas de trabalho, que foram efetivamente completadas em um determinado momento do cronograma do projeto.

Wilkens [WIL99] nota que "a distinção entre BCWS e BCWP é que a primeira representa o orçamento das atividades que tinham sido planejadas para estarem completadas e o último representa o orçamento das atividades que realmente tinham sido completadas". Uma vez determinados os valores de BCWS, BAC e BCWP, importantes indicadores de progresso podem ser calculados:

> Índice de desempenho do cronograma (*Schedule Performance Index* — SPI),
> SPI = BCWP/BCWS
> Variância do cronograma (*Schedule Variance* — SV), SV = BCWP – BCWS

Veja na Web

Uma ampla gama de recursos de análise de valor agregado pode ser encontrada em www.acq.osd.mil/pm/.

O SPI é uma indicação da eficiência com a qual o projeto está utilizando os recursos cronogramados. Um valor de SPI perto de 1,0 indica execução eficiente do cronograma do projeto. SV é simplesmente uma indicação absoluta da variância do cronograma planejado.

> Porcentagem programada para conclusão = BCWS/BAC

fornece indicação da porcentagem do trabalho que deveria ter sido completada no momento de tempo *t*.

> Porcentagem completada = BCWP/BAC

fornece uma indicação quantitativa da porcentagem de conclusão do projeto em um determinado momento de tempo, *t*.

É também possível calcular o *custo real do trabalho realizado* (*Actual Cost of Work Performed* — ACWP). O valor do ACWP é a soma do esforço despendido realmente nas tarefas de trabalho que foram completadas em um certo momento do cronograma do projeto. É possível, então, calcular

> Índice de desempenho do custo (*Cost Performance Index* — CPI), CPI = BCWP/ACWP
> Variância do custo (*Cost Variance* — CV), CV = BCWP – ACWP

Um valor de CPI próximo de 1,0 dá uma forte indicação de que o projeto está dentro do orçamento definido. CV é uma indicação absoluta das economias em um custo (em relação aos custos planejados) ou dos excessos de custo em um estágio específico de um projeto.

Da mesma forma que um radar além do horizonte, a análise do valor agregado ilumina as dificuldades de cronogramação antes que elas possam ser notadas de outra forma. Isso permite ao gerente de projeto de software adotar ação corretiva, antes que uma crise de projetos se desenvolva.

24.7 Resumo

Cronogramação é o resultado da atividade de planejamento, que é o componente principal da gestão de projeto de software. Quando combinada com métodos de estimativa e análise de risco, a cronogramação estabelece um roteiro para o gerente de projeto.

A cronogramação começa com decomposição do processo. As características do projeto são usadas para adaptar um conjunto de tarefas adequado ao trabalho a ser efetuado. Uma rede de tarefas representa cada tarefa de engenharia, sua dependência de outras tarefas e sua duração projetada. A rede de tarefas é usada para calcular o caminho crítico, um gráfico de tempo e uma grande variedade de informação de projeto. Usando o cronograma como guia, o gerente de projeto pode acompanhar e controlar cada passo do processo de software.

Referências Bibliográficas

[BRO95] Brooks, M., *The Mythical Man-Month*, edição de aniversário, Addison-Wesley, 1995.

[FLE98] Fleming, Q. W. e Koppelman, J. M. "Earned Value Project Management", *Crosstalk*, v. 11, n. 7, jul. 1998, p. 19.

[HUM95] Humphrey, W., *A Discipline for Software Engineering*, Addison-Wesley, 1995.

[NOR70] Norden, P., "Useful Tools for Project Management," in *Management of Production*, M. K. Starr, ed., Penguin Books, 1970.

Problemas e Pontos a Considerar

24.1. Datas de entrega "não razoáveis" são fatos corriqueiros nos negócios de software. Como você deve agir no caso de deparar com uma?

24.2. Qual a diferença entre um cronograma macroscópico e um cronograma detalhado? É possível gerir um projeto se apenas um cronograma macroscópico é desenvolvido? Por quê?

24.3. Há algum caso em que um marco de referência de projeto de software não está ligado a uma revisão? Em caso afirmativo, forneça um ou mais exemplos.

24.4. "Excesso de comunicação" pode ocorrer quando muitas pessoas trabalham em um projeto de software. O tempo gasto em comunicação com os outros reduz a produtividade individual (LOC/pessoa-mês), e o resultado é menor produtividade da equipe. Ilustre (quantitativamente) como engenheiros que são versados em boas práticas de engenharia de software e usam revisões técnicas formais podem aumentar a taxa de produção de uma equipe (quando comparada com a soma de taxas de produção individuais). Dica: você pode considerar que as revisões reduzem o retrabalho e que retrabalho pode tomar 20 a 40% do tempo de uma pessoa.

24.5. Apesar de a adição de pessoal a um projeto atrasado poder atrasá-lo mais, há circunstâncias nas quais isso não é verdade. Descreva-as.

24.6. O relacionamento entre pessoal e tempo é altamente não-linear. Usando a equação de software de Putnam (descrita na Seção 24.2.2), desenvolva uma tabela que relacione o número de pessoas à duração do projeto, para um projeto de software que exige 50.000 LOC e 15 pessoas-ano de esforço (o parâmetro de produtividade é 5.000). Considere que o software deve ser entregue em mais de 24 meses ou menos de 12 meses.

24.7. Considere que você foi contratado por uma universidade para desenvolver um sistema on-line para matrícula em cursos (SOLMC). Primeiro, aja como cliente (se você é aluno isso deve ser fácil!) e especifique as características de um bom sistema. (Como alternativa, seu instrutor vai fornecer-lhe um conjunto de requisitos preliminares

Document shared on www.docsity.com

de um sistema.) Usando os métodos de estimativa discutidos no Capítulo 23, desenvolva uma estimativa de esforço e duração para o SOLMC. Sugira como você:

a. Definiria atividades de trabalho paralelas durante o projeto do SOLMC.

b. Distribuiria o esforço ao longo do projeto.

c. Estabeleceria marcos de referência para o projeto.

24.8. Selecione um conjunto de tarefas adequado para o projeto SOLMC.

24.9. Defina uma rede de tarefas para o SOLMC, ou, como alternativa, para outro projeto de software que lhe interesse. Não deixe de mostrar as tarefas e marcos de referência e de juntar estimativas de esforço e duração para cada tarefa. Se possível, use uma ferramenta automática de cronogramação para realizar esse trabalho.

24.10. Se uma ferramenta de cronogramação automática estiver disponível, determine o caminho crítico da rede definida no Problema 24.9.

24.11. Usando uma ferramenta de cronogramação (se disponível) ou papel e lápis (se necessário), desenvolva um gráfico de tempo para o projeto do SOLMC.

24.12. Considere que você é um gerente de projeto de software e que lhe foi solicitado calcular estatísticas de valor agregado para um pequeno projeto de software. O projeto tem 56 tarefas de trabalho planejadas, para as quais foram estimadas 582 pessoas-dia para sua finalização. Na ocasião em que a análise de valor agregado lhe é solicitada, 12 tarefas já foram completadas. Todavia, o cronograma do projeto indica que 15 tarefas deveriam ter sido completadas. Os seguintes dados de cronogramação (em pessoas-dia) estão disponíveis:

Tarefa	Esforço planejado	Esforço real
1	12,0	12,5
2	15,0	11,0
3	13,0	17,0
4	8,0	9,5
5	9,5	9,0
6	18,0	19,0
7	10,0	10,0
8	4,0	4,5
9	12,0	10,0
10	6,0	6,5
11	5,0	4,0
12	14,0	14,5
13	16,0	—
14	6,0	—
15	8,0	—

Calcule o SPI, a variância do cronograma, a porcentagem para conclusão planejada, a porcentagem completada, o CPI e a variância de custo do projeto.

Leituras e Fontes de Informação Adicionais

Virtualmente todo livro escrito sobre gestão de projeto de software contém uma discussão de cronogramação. Project Management Institute (*PMBOK Guide*, PMI, 2001), Wysoki e seus colegas (*Effective Project Management*, Wiley, 2000), Lewis (*Project Planning Scheduling and Control*, 3. ed., McGraw-Hill, 2000), Bennatan (*On Time, Within Budget: Software Project Management Practices and Techniques*, 3. ed., Wiley, 2000), McConnell (*Software Project Survival Guide*, Microsoft Press, 1998) e Roetzheim e Beasley (*Software Project Cost and Schedule Estimating: Best Practices*, Prentice-Hall, 1997) contêm uma discussão que vale a pena. Boddie (Crunch Mode, Prentice-Hall, 1987) escreveu um livro para todos os gerentes que "têm 90 dias para fazer um projeto de seis meses".

McConnell (*Rapid Development*, Microsoft Press, 1996) apresenta uma excelente discussão das razões que levam a cronogramas de projetos de software excessivamente otimistas e do que você pode fazer a respeito. O'Connell (*How to Run Successful Projetcs II: The Silver Bullet*, Prentice-Hall, 1997) apresenta uma abordagem passo a passo para a gerência de projeto, que vai ajudá-lo a desenvolver um cronograma realístico para seus projetos.

Webb e Wake (*Using Earned Value: A Project Manager's Guide*, Ashgate Publishing, 2003) e Fleming e Koppelman (*Earned Value Project Management*, Project Management Institute Publications, 1996) discutem o uso de técnicas de valor agregado para planejamento, acompanhamento e controle de projeto em considerável detalhe.

Uma ampla variedade de fontes de informação sobre cronogramação de projeto de software está disponível na Internet. Uma lista atualizada de referências da World Wide Web pode ser encontrada no site deste livro: **http://www.mhhe.com/pressman**.

Document shared on www.docsity.com

CAPÍTULO

Gestão de Risco

Conceitos-Chave

avaliação 567
categorias de risco 563
estratégias proativas 563
estratégias reativas 563
exposição ao risco 569
identificação 564
previsão 566
princípios 564
refinamento 570
RMMM 573
segurança e imprevistos... 572
tabela de risco 568

Em seu livro sobre análise e gestão de riscos, Robert Charette [CHA89] apresenta uma definição conceitual de risco:

> Em primeiro lugar, risco afeta acontecimentos futuros. Presente e passado não preocupam, pois o que colhemos hoje já foi semeado por nossas ações anteriores. A questão é, mudando nossas ações hoje, podemos criar oportunidade para uma situação diferente e possivelmente melhor para nós amanhã? Isso significa, em segundo lugar, que risco envolve modificação, como, por exemplo, modificação de pensamento, opinião, ações ou lugares... [Em terceiro lugar], o risco envolve a escolha e a incerteza que a própria escolha envolve. Assim, paradoxalmente, o risco, como a morte e os impostos, é uma das poucas certezas da vida.

Quando o risco é considerado no contexto da engenharia de software, as três fundamentações conceituais de Charette estão sempre em evidência. O futuro é nossa preocupação — que riscos podem causar o insucesso do projeto de software? A modificação é nossa preocupação — como as modificações nos requisitos do cliente, nas tecnologias de desenvolvimento, nos computadores-alvo e em todas as outras entidades ligadas ao projeto afetam a pontualidade e o sucesso geral? Por último, devemos cuidar das escolhas — que métodos e ferramentas devemos usar, quantas pessoas devem ser envolvidas, quanta ênfase em qualidade é "suficiente"?

Peter Drucker [DRU75] disse certa vez, "já que é fútil tentar eliminar riscos e questionável tentar minimizá-los, o essencial é que os riscos considerados sejam os certos". Antes que possamos identificar os "riscos certos" que acontecerão durante um projeto de software, é importante identificar todos os demais que são óbvios, tanto para gerentes quanto para profissionais.

Panorama

O que é? Análise e gestão de riscos é uma série de passos que ajudam uma equipe de software a entender e administrar a incerteza. Muitos problemas podem perturbar um projeto de software. Um risco é um problema em potencial — pode ou não ocorrer. Mas, independentemente do resultado, é realmente uma boa idéia identificá-lo, avaliar a sua probabilidade de ocorrência, estimar seu impacto e estabelecer um plano de contingência no caso de o problema efetivamente ocorrer.

Quem faz? Todos os envolvidos no processo de software — gerentes, engenheiros de software e interessados — participam da análise e gestão de risco.

Por que é importante? Pense sobre o lema dos escoteiros: Sempre alerta. Software é um empreendimento difícil. Muitas coisas podem dar errado e, francamente, com freqüência dão. É por essa razão que estar alerta — entender os riscos e tomar medidas proativas para evitá-los ou administrá-los — é um elemento-chave da boa administração de projeto de software.

Quais são os passos? Reconhecer o que pode dar errado é o primeiro passo, chamado de "identificação de risco". Depois, cada risco é analisado para determinar a possibilidade de que venha ocorrer e o dano que vai causar, se efetivamente ocorrer. Uma vez estabelecida essa informação, os riscos são classificados por probabilidade e impacto. Finalmente, é desenvolvido um plano para administrar aqueles riscos com alta probabilidade e alto impacto.

Qual é o produto do trabalho? É produzido um plano de atenuação, monitoração e gestão de riscos (Risk mitigation, monitoring and management — RMMM), ou um conjunto de formulários de informação de risco.

Como tenho certeza de que fiz corretamente? Os riscos analisados e geridos devem ser originados de um profundo estudo do pessoal, do produto, do processo e do projeto. O plano R deve ser revisto à medida que o projeto evolui, para garantir que os riscos sejam atualizados. Os planos de contingência para a administração dos riscos devem ser realistas.

25.1 Estratégias de Risco Proativas *versus* Reativas

Estratégias de risco reativas têm sido zombeteiramente chamadas "escola de gestão de riscos Indiana Jones" [THO92]. Nos filmes da década de 1980 que levam seu nome, Indiana Jones, ao deparar com uma dificuldade intransponível, diria invariavelmente, "não se preocupe, vou pensar em alguma coisa!". Nunca se preocupando com problemas, até eles ocorrerem, Indiana reagiria de algum modo heróico.

> "Se você não atacar ativamente os riscos eles irão atacá-lo ativamente."
>
> Tom Gilb

Infelizmente, o gerente de projeto de software médio não é Indiana Jones e os membros da equipe de projeto de software não são seus fiéis seguidores. No entanto, a maioria das equipes de software conta somente com estratégias reativas a risco. Quando muito, uma estratégia reativa monitora o projeto quanto a riscos prováveis. São reservados recursos para lidar com eles quando se tornarem problemas reais. Mais comumente, a equipe de software não faz nada a respeito de risco até que algo dê errado. Então, a equipe corre para a ação, em uma tentativa de corrigir o problema rapidamente. Isso é freqüentemente chamado *modo de combate ao fogo.* Quando isso falha, os "administradores de crises" [CHA92] assumem e o projeto está em perigo iminente.

Uma estratégia consideravelmente mais inteligente para a gestão de risco é ela ser proativa. Uma estratégia proativa começa muito antes de o trabalho técnico ser iniciado. Riscos potenciais são identificados, suas probabilidades e impactos são avaliados, e eles são classificados por importância. Então, a equipe de software estabelece um plano para administrar riscos. O objetivo principal é evitar riscos, mas, como nem todos podem ser evitados, a equipe trabalha para desenvolver um plano de contingência, que vai permitir a ela responder de modo controlado e efetivo. No restante deste capítulo, discutimos uma estratégia proativa para a administração de risco.

25.2 Riscos de Software

Apesar de debates consideráveis sobre a definição adequada para risco, há um consenso geral de que o risco sempre envolve duas características [HIG95]:

- *Incerteza* — o risco pode ou não acontecer; isto é, não há riscos 100% prováveis[1].
- *Perda* — se o risco tornar-se real, conseqüências indesejadas ou perdas ocorrerão.

? Que tipos de risco são provavelmente encontrados à medida que o software é construído?

Quando os riscos são analisados, é importante quantificar o nível de incerteza e o grau de perda associados a cada risco. Para conseguir isso, diferentes categorias de risco são consideradas.

Riscos de projeto ameaçam o plano do projeto. Isto é, se riscos de projetos tornam-se reais, é provável que o cronograma do projeto se atrase e que os custos aumentem. Riscos de projeto identificam problemas potenciais orçamentários, de cronograma, de pessoal (quantidade e organização), de recursos, de interessados, e de requisitos e seu impacto em um projeto de software. No Capítulo 23, a complexidade, o tamanho e o grau de incerteza estrutural de um projeto foram também definidos como fatores de risco do projeto (e de estimativa).

1 Um risco 100% provável é uma restrição ao projeto de software.

Document shared on www.docsity.com

Riscos técnicos ameaçam a qualidade e a pontualidade do software a ser produzido. Se um risco técnico se torna realidade, a implementação pode tornar-se difícil ou impossível. Os riscos técnicos identificam problemas potenciais de projeto, de implementação, de interface, de verificação e de manutenção. Além disso, ambigüidade nas especificações, incerteza técnica, obsolescência técnica e tecnologia "de ponta" também são fatores de risco. Riscos técnicos ocorrem porque o problema é mais difícil de resolver do que pensávamos.

Riscos do negócio ameaçam a viabilidade do software a ser construído. Riscos de negócio freqüentemente comprometem o projeto ou o produto. Os prováveis cinco principais riscos de negócio são: (1) construir um produto ou um sistema excelente que ninguém realmente quer (risco de mercado); (2) construir um produto que não se encaixa mais na estratégia geral de negócios da empresa (risco estratégico); (3) construir um produto que a equipe de vendas não sabe como vender; (4) perda de apoio da gerência superior por causa de modificação de enfoque ou de pessoal (risco gerencial) e (5) perda de comprometimento orçamentário ou de pessoal (riscos de orçamento).

É extremamente importante observar que categorização simples de risco nem sempre funciona. Alguns riscos são simplesmente imprevisíveis antecipadamente.

Outra categorização geral de riscos foi proposta por Charette [CHA89]. *Riscos conhecidos* são aqueles que podem ser descobertos após a avaliação cuidadosa do plano de projeto, do ambiente técnico e comercial, no qual o projeto está sendo desenvolvido, e de outras fontes de informação confiáveis (por exemplo, prazo de entrega irreal, falta de documentação dos requisitos ou do escopo do software e mau ambiente de desenvolvimento). *Riscos previsíveis* são extrapolados de experiência de projetos anteriores (por exemplo, rotatividade do pessoal, má comunicação com o cliente, diluição do esforço de pessoal à medida que os pedidos de manutenção que surgem são atendidos). *Riscos imprevisíveis* são o curinga do baralho. Eles podem e efetivamente ocorrem, mas são extremamente difíceis de identificar antecipadamente.

INFO

Sete Princípios de Gestão de Risco

O Software Engineering Institute (SEI) (www.sei.cmu.edu) identifica sete princípios que "fornecem um arcabouço para conseguir uma gestão efetiva do risco". Eles são:

Mantenha uma perspectiva global — observar os riscos de software no contexto do sistema no qual ele é um componente e o problema de negócio que ele pretende resolver.

Adote uma visão de olhar adiante — pense sobre os riscos que podem surgir no futuro (por exemplo, por modificações no software); estabeleça planos de contigência para que eventos futuros sejam gerenciáveis.

Encoraje comunicação aberta — se alguém menciona um risco potencial, não o ignore. Se um risco é proposto de maneira informal, considere-o. Encoraje todos os interessados e usuários a sugerir riscos a qualquer momento.

Integre — uma consideração de risco deve ser integrada no processo de software.

Enfatize um processo contínuo — a equipe deve estar vigilante ao longo de todo o processo de software, modificando os riscos identificados à medida que mais informação é conhecida e adicionando novos quando melhor visão é conseguida.

Desenvolva uma visão de produto compartilhada — se todos os interessados compartilham a mesma visão do software, é provável que uma melhor identificação e avaliação de riscos ocorra.

Encoraje o trabalho em equipe — os talentos, aptidões e conhecimento de todos os interessados devem ser consultados quando as atividades de gestão de riscos são conduzidas.

25.3 Identificação de Riscos

Identificação de risco é uma tentativa sistemática de especificar ameaças ao plano de projeto (estimativas, cronograma, suprimento de recursos etc.). Pela identificação dos riscos conhecidos e previsíveis, o gerente de projeto dá um primeiro passo no sentido de evitá-los, quando possível, e controlá-los, quando necessário.

Há dois tipos distintos de risco para cada uma das categorias apresentadas na Seção 25.2: riscos genéricos e riscos específicos do produto. *Riscos genéricos* são uma ameaça potencial a todo projeto de software. *Riscos específicos do produto* podem ser identificados somente por aqueles que têm um claro entendimento da tecnologia, do pessoal e do ambiente que são específicos do projeto em mãos. Para identificar riscos específicos do produto, o plano de projeto e a declaração de escopo do software são examinados, e é dada uma resposta à seguinte questão: "Que características especiais desse produto podem ameaçar nosso plano de projeto?".

> "Projetos sem riscos reais são fracassados. Eles são quase sempre incapazes de benefício; essa é a razão pela qual eles não foram feitos anos atrás."
>
> Tom DeMarco e Tim Lister

AVISO

Apesar de ser importante considerar riscos genéricos, os riscos específicos do produto, geralmente, causam a maioria das dores de cabeça. Vale a pena gastar tempo para identificar tantos riscos específicos do produto quanto possível.

Um método para a identificação de riscos é a criação de uma checklist de itens de risco. A checklist pode ser usada para identificação de risco, e concentra-se em algum subconjunto de riscos conhecidos e previsíveis das seguintes subcategorias genéricas:

- *Tamanho do produto* — riscos associados com a dimensão geral do software a ser construído ou modificado.
- *Impacto no negócio* — riscos associados com restrições impostas pela gerência ou pelo mercado.
- *Características do cliente* — riscos associados com a sofisticação do cliente e com a capacidade do desenvolvedor de se comunicar com o cliente a tempo.
- *Definição do processo* — riscos associados com o grau em que o processo de software foi definido e é seguido pela organização de desenvolvimento.
- *Ambiente de desenvolvimento* — riscos associados com a disponibilidade e a qualidade das ferramentas a serem usadas para construir o produto.
- *Tecnologia para a construção* — riscos associados com a complexidade do sistema a ser construído e com a "novidade" da tecnologia que é incorporada ao sistema.
- *Tamanho e experiência da equipe* — riscos associados com a técnica em geral e com a experiência no projeto dos engenheiros de software que vão fazer o trabalho.

A checklist dos itens de risco pode ser organizada de vários modos. As questões relevantes a cada um dos tópicos podem ser respondidas para cada projeto de software. As respostas a essas questões permitem ao planejador estimar o impacto do risco. Um formato diferente da checklist de itens de risco simplesmente lista as características que são relevantes a cada subcategoria genérica. Finalmente, um conjunto de "componentes e fatores de risco" [AFC88] é listado juntamente com sua probabilidade de ocorrência. Fatores de desempenho, de apoio, de custo e de cronograma são discutidos em resposta às últimas questões.

Várias checklists abrangentes para riscos de projetos de software foram propostas na literatura (por exemplo, [SEI93], [KAR96]). Elas fornecem entendimento útil dos riscos genéricos para projetos de software e devem ser usadas sempre que a análise e a gestão de riscos são instituídas. Todavia, uma lista de questões relativamente curta [KEI98] pode ser usada para fornecer uma indicação preliminar sobre se um projeto está "em risco".

25.3.1 Avaliação do Risco Global do Projeto

? O projeto de software em que estamos trabalhando está correndo risco sério?

As seguintes questões foram derivadas de dados de riscos obtidos por levantamento feito com gerentes de projeto de software experientes, em diferentes partes do mundo [KEI98]. As questões estão ordenadas por sua importância relativa em relação ao sucesso de um projeto.

1. A alta administração do software e do cliente empenhou-se formalmente em apoiar o projeto?
2. Os usuários finais estão entusiasticamente empenhados com relação ao projeto e ao sistema/produto a ser construído?

Document shared on www.docsity.com

Veja na Web

Risk Radar é uma ferramenta e banco de dados que ajuda gerentes a identificar, ordenar e comunicar riscos de projeto. Pode ser encontrado em www.spmn.com.

3. Os requisitos estão plenamente entendidos pela equipe de engenharia de software e por seus clientes?
4. Os clientes envolveram-se totalmente na definição dos requisitos?
5. Os usuários finais têm expectativas realísticas?
6. O escopo do projeto é estável?
7. A equipe de engenharia de software tem a combinação de aptidões adequada?
8. Os requisitos do projeto são estáveis?
9. A equipe de projeto tem experiência com a tecnologia a ser implementada?
10. A quantidade de pessoal da equipe de projeto é adequada para fazer o serviço?
11. Todos os membros da equipe do cliente/usuário envolvidas concordam com a importância do projeto e com os requisitos do sistema/produto a ser construído?

> "Gestão de risco é gestão de projeto para adultos."
>
> Tim Lister

Se qualquer dessas questões for respondida negativamente, os passos de atenuação, monitoração e gestão devem ser formalizados imediatamente. O grau em que o projeto está em risco é diretamente proporcional ao número de respostas negativas a essas questões.

25.3.2 Componentes e Fatores de Risco

A força aérea americana [AFC88] elaborou um folheto que contém excelentes diretrizes para identificação e redução de riscos. A abordagem da força aérea exige que o gerente de projeto identifique os fatores de risco que afetam os componentes de risco do software — desempenho, custo, apoio e cronograma. No contexto dessa discussão, os componentes de risco são definidos do seguinte modo:

- *Risco de desempenho* — o grau de incerteza quanto ao produto atender a seus requisitos e ser adequado para seu uso planejado.
- *Risco de custo* — o grau de incerteza quanto ao orçamento do projeto ser mantido.
- *Risco de suporte* — o grau de incerteza quanto ao software resultante ser fácil de corrigir, adaptar e aperfeiçoar.
- *Risco de cronograma* — o grau de incerteza quanto ao cronograma do projeto ser cumprido e de o produto ser entregue no prazo.

O impacto de cada fator de risco, no componente de risco, é dividido em uma das quatro categorias de impacto seguintes — negligível, marginal, crítica ou catastrófica. Com referência à Figura 25.1 [BOE89], é descrita uma caracterização das conseqüências potenciais de erros (linhas com rótulo 1) ou falhas, para conseguir um resultado desejado (linhas com rótulo 2). A categoria de impacto é escolhida com base na caracterização que melhor se encaixa na descrição da tabela.

25.4 Previsão de Risco

A *previsão de risco,* também chamada *estimativa de risco,* tenta avaliar cada risco de dois modos — (1) a probabilidade de que o risco seja real e (2) as conseqüências dos problemas associados ao risco, se ele ocorrer. O planejador do projeto, junto com outros gerentes e com o pessoal técnico, desenvolve quatro atividades de previsão de risco:

1. estabelece uma escala que reflete a probabilidade percebida de um risco.
2. delineia as conseqüências do risco.
3. estima o impacto do risco no projeto e no produto.
4. anota a precisão da previsão de risco, de modo que não haja mal-entendidos.

FIGURA 25.1

Avaliação de impacto [BOE89]

Categoria \ Componentes		Desempenho	Apoio	Custo	Cronograma
Catastrófica	1	Falha em satisfazer o requisito resultaria na falha da missão		A falha resulta em aumento de custo e atraso de cronograma com valores previsto que excedem $500 mil	
	2	Da degradação significativa a não-alcance do desempenho técnico	Apoio de software não-responsivo ou software inapoiável	Falta significativa de recursos financeiros, provável estouro de orçamento	Data de entrega inexeqüível
Crítica	1	Falha em satisfazer o requisito degradaria o desempenho do sistema até um ponto em que o sucesso da missão é questionável		A falha resulta em atrasos operacionais e/ou aumento de custos com valor previsto de $100 a $500 mil	
	2	Alguma redução no desempenho técnico	Pequenos atrasos nas modificações do software	Alguma falta de recursos financeiros, possível estouro de orçamento	Possível ultrapassagem da data de entrega
Marginal	1	Falha em satisfazer o requisito resultaria na degradação da missão secundária		Custos, impactos e/ou atrasos de cronograma recuperáveis com valor previsto de $1 a $100 mil	
	2	De mínima a pequena redução do desempenho técnico	Apoio de software responsivo	Recursos financeiros suficientes	Cronograma realístico exeqüível
Negligível	1	Falha em satisfazer o requisito criaria inconveniência ou impacto não operacional		Erro resulta em pequeno impacto no custo e/ou cronograma com valor previsto de menos que $1 mil	
	2	Sem redução no desempenho técnico	Software facilmente apoiável	Possível sobra de orçamento	Data de entrega antecipável

Nota: (1) Potencial conseqüência de erros ou falhas de software não-detectadas.
(2) Potencial conseqüência se o resultado desejado não é conseguido.

A intenção desses passos é considerar riscos de modo que conduza à priorização. Nenhuma equipe de software tem os recursos para tratar todos os riscos possíveis com o mesmo grau de rigor. Priorizando riscos, a equipe pode alocar recursos onde eles vão ter maior impacto.

25.4.1 Desenvolvimento de uma Tabela de Risco

Uma tabela de risco fornece a um gerente de projeto uma técnica simples para previsão de risco.[2] Um exemplo de tabela de risco é ilustrado na Figura 25.2.

AVISO

Pense bastante sobre o software que você está prestes a construir e pergunte a si mesmo o que pode dar errado? Crie uma lista para você e peça aos outros membros da equipe de software para fazer o mesmo.

Uma equipe de software começa listando todos os riscos (não importa quão remotos sejam) na primeira coluna da tabela. Isso pode ser conseguido com a ajuda da checklist de itens de risco mencionada na Seção 25.3. Cada risco é caracterizado na segunda coluna (por exemplo, PS — *Project Size* — implica risco de tamanho do projeto, BU — *BUsiness* — implica risco de negócio). A probabilidade de ocorrência de cada risco é colocada na próxima coluna da tabela. O valor da probabilidade para cada risco pode ser estimado individualmente pelos membros da equipe. Cada membro da equipe opina seqüencialmente até que sua avaliação da probabilidade de risco comece a convergir.

Em seguida, é estimado o impacto de cada risco. Cada componente de risco é avaliado usando a categorização apresentada na Figura 25.1 e uma categoria de impacto é determinada. As categorias para cada um dos quatro componentes de risco — desempenho, suporte, custo e cronograma — são submetidas a um cálculo de média[3] para determinar um valor global de impacto.

Uma vez completadas as quatro primeiras colunas, a tabela é ordenada por probabilidade e por impacto. Riscos com alta probabilidade e alto impacto vão para o alto da tabela e riscos com baixa probabilidade vão para a parte de baixo. Assim se consegue a priorização de riscos de primeira ordem.

2 A tabela de risco deve ser implementada como um modelo de planilha. Isso permite manipulação fácil e ordenação das entradas.

3 Uma média ponderada pode ser usada se um componente de risco tem maior significância para o projeto.

Document shared on www.docsity.com

FIGURA 25.2

Exemplo de tabela de risco antes da ordenação

Riscos	Categoria	Probabilidade	Impacto	RMMM
A estimativa de tamanho pode ser significativamente baixa	Tamanho do produto	60%	2	
Número de usuários maior que o planejado	Tamanho do produto	30%	3	
Menos reuso que o planejado	Tamanho do produto	70%	2	
Usuários finais resistem ao sistema	Impacto do negócio	40%	3	
Prazo de entrega será apertado	Impacto do negócio	50%	2	
Financiamento será perdido	Características do cliente	40%	1	
Cliente modificará os requisitos	Tamanho do produto	80%	2	
Tecnologia não satisfará as expectativas	Tecnologia a ser usada	30%	1	
Falta de treinamento no uso das ferramentas	Ambiente de desenvolvimento	80%	3	
Pessoal inexperiente	Tamanho e experiência da equipe	30%	2	
Rotatividade do pessoal será alta	Tamanho e experiência da equipe	60%	2	
⋮				

Valores de impacto:
1— catastrófico
2— crítico
3— marginal
4— negligível

PONTO CHAVE

A tabela de risco é ordenada por probabilidade e impacto para classificar os riscos.

O gerente de projeto estuda a tabela ordenada resultante e define uma linha de corte. A *linha de corte* (traçada horizontalmente em algum ponto da tabela) sugere que apenas riscos situados acima da linha receberão atenção subseqüente. Riscos que ficam abaixo da linha são reavaliados para se chegar à priorização de segunda ordem. Observando a Figura 25.3, o impacto e a probabilidade de risco têm influência distinta na preocupação da gerência. Um fator de risco que tem alto impacto, mas probabilidade de ocorrência muito baixa, não deve absorver parcela significativa do tempo da gerência. Todavia, riscos de alto impacto, com probabilidade de moderada a alta e riscos de baixo impacto, com alta probabilidade, devem ser considerados nos passos de análise de risco seguintes.

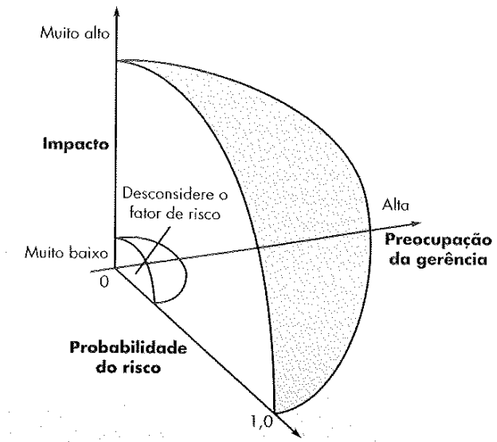
Todos os riscos que ficam acima da linha de corte devem ser administrados. A coluna com rótulo RMMM contém um ponteiro para um plano de *atenuação (Mitigation), monitoração (Monitoring) e gestão de risco (Management)* ou, alternativamente, para uma coleção de *folhas de informação de risco* desenvolvida para todos os riscos que ficam acima da linha de corte. O plano RMMM e as folhas de informação de risco são discutidos nas Seções 25.5 e 25.6.

> "[Hoje em dia,] ninguém se dá ao luxo de ficar conhecendo uma tarefa tão bem que ela não guarde nenhuma surpresa, e surpresas significam risco."
>
> Stephen Grey

A probabilidade de risco pode ser determinada fazendo-se estimativas individuais e depois desenvolvendo um único valor de consenso. Apesar de essa abordagem funcionar, técnicas mais sofisticadas para determinação da probabilidade de risco foram desenvolvidas [AFC88]. Fatores de risco podem ser avaliados em uma escala de probabilidades qualitativa que tem os seguintes valores: impossível, improvável, provável e freqüente. A probabilidade matemática pode então ser associada a cada valor qualitativo (por exemplo, uma probabilidade de 0,7 a 1,0 implica um risco altamente provável).

FIGURA 25.3

Risco de preocupação da gerência



25.4.2 Avaliação do Impacto do Risco

Três fatores afetam as conseqüências que são prováveis se um risco ocorrer: sua natureza, seu escopo e sua época. A natureza do risco indica os problemas que podem surgir se ele ocorrer. Por exemplo, uma interface externa mal definida para o hardware do cliente (risco técnico) prejudicará o início do projeto e dos testes, e vai provavelmente levar a problemas de integração do sistema no fim do projeto. O escopo de um risco combina a severidade (quão sério ele é?) com sua distribuição geral (quanto do projeto será afetado ou quantos clientes serão prejudicados?). Finalmente, a época de um risco considera quando e por quanto tempo o impacto será sentido. Na maioria dos casos, um gerente de projeto pode querer que as "más notícias" ocorram tão cedo quanto possível, mas, em alguns casos, quanto mais tarde, melhor.

Voltando novamente à abordagem de análise de risco proposta pela força aérea americana [AFC88], os seguintes passos são recomendados para determinar as conseqüências gerais de um risco:

? Como avaliamos as conseqüências de um risco?

1. Determine o valor médio da probabilidade de ocorrência de cada componente de risco.
2. Usando a Figura 25.1, determine o impacto de cada componente, com base nos critérios mostrados.
3. Complete a tabela de risco e analise os resultados como descrito na seção anterior.

A *exposição ao risco* (*risk exposure* — RE) é determinada usando a seguinte relação [HAL98]:

$$RE = P \times C$$

em que P é a probabilidade de ocorrência de um risco e C é o custo para o projeto no caso de o risco ocorrer.

Por exemplo, considere que a equipe de software defina um risco de projeto da seguinte maneira:

Identificação do risco. Apenas 70% dos componentes de software programados para reuso serão, de fato, integrados à aplicação. A funcionalidade restante terá que ser desenvolvida sob medida.

Probabilidade do risco. 80% (aproximadamente).

Impacto do risco. Sessenta componentes de software reusáveis foram planejados. Se apenas 70% podem ser efetivamente usados, 18 componentes teriam que ser desenvolvidos a partir do zero (além de outros software sob medida, que foram programados para serem desenvolvidos). Como o tamanho médio do componente é 100 LOC e os dados locais indicam que o custo de engenharia de software para cada LOC é de 14 dólares, o custo geral (impacto) para desenvolver os componentes seria 18 × 100 × 14 = U$25.200.

Exposição ao risco. RE = 0,80 × 25.200 U$20.200.

Document shared on www.docsity.com

AVISO

Compare RE, para todos os riscos, com a estimativa de custo do projeto. Se RE for maior do que 50% do custo do projeto, a viabilidade do projeto deve ser avaliada.

A exposição ao risco pode ser calculada para cada risco da tabela de riscos, assim que é feita a estimativa de custo do risco. A exposição total ao risco, para todos os riscos (acima da linha de corte na tabela de riscos), pode fornecer um modo de ajustar a estimativa final de custo de um projeto. Pode também ser usada para prever o aumento provável nos recursos de pessoal, necessários em vários pontos do cronograma do projeto.

As técnicas de previsão e análise de riscos descritas na Seção 25.4.1 e 25.4.2 são aplicadas iterativamente à medida que o projeto de software prossegue. A equipe de projeto deve revisitar a tabela de riscos em intervalos regulares, reavaliando cada risco para determinar quando novas circunstâncias causam modificações na sua probabilidade e seu impacto. Como conseqüência dessa atividade, pode ser necessário adicionar riscos à tabela, remover alguns riscos que não são mais relevantes e mudar as posições relativas de outros.

CASASEGURA

Análise de Risco

A cena: Escritório de Doug Miller, antes do início do projeto de software do *CasaSegura*.

Os personagens: Doug Miller (gerente da equipe de engenharia de software do *CasaSegura*) e Vinod Raman, Jamie Lazar, e outros membros da equipe de engenharia de software do produto.

A conversa:

Doug: Eu gostaria de gastar algum tempo fazendo *brainstorming* de riscos para o projeto do *CasaSegura*.

Jamie: Como o que pode dar errado?

Doug: É. Aqui estão algumas categorias em que as coisas podem dar errado. [Ele mostra a todos as categorias mencionadas na introdução da Seção 25.3.]

Vinod: Humm... você quer apenas chamar a nossa atenção para eles, ou...

Doug: Eis aqui o que eu acho que deveríamos fazer. Todos fariam uma lista de riscos... agora...

(Dez minutos depois; todos estão escrevendo.)

Doug: Certo, parem.

Jamie: Mas eu ainda não fiz!

Doug: Está certo. Vamos revisitar a lista novamente. Agora, para cada item da sua lista, atribua uma porcentagem de probabilidade que o risco vai ocorrer. Depois, atribua um impacto para o projeto na escala de 1 (insignificante) a 5 (catastrófico).

Vinod: Então se eu acho que o risco é alto eu especifico uma probabilidade de 50%, e se eu acho que ele vai ter um impacto moderado no projeto, eu especifico 3, certo?

Doug: Exatamente.

(Cinco minutos depois; todos estão escrevendo.)

Doug: Certo, parem. Agora vamos fazer uma lista do grupo no quadro branco. Eu vou escrever, nós vamos consultar uma entrada da sua lista de forma circular.

(Quinze minutos depois; a lista está criada.)

Jamie (apontando para o quadro e rindo): Vinod, aquele risco (apontando para uma entrada no quadro) é ridículo. Há uma maior probabilidade de que nós todos sejamos atingidos por um raio. Precisamos removê-lo.

Doug: Não, deixe por enquanto. Consideramos todos os riscos, não importa quão estranho. Depois limparemos a lista.

Jamie: Mas, nós já temos 40 riscos... como é que vamos poder geri-los todos?

Doug: Não podemos. Aí está a razão pela qual nós definimos um ponto de corte depois que os ordenarmos. Eu vou fazer isso no intervalo, e nós nos reunimos novamente amanhã. Por enquanto, voltem ao seu trabalho... e no tempo livre, pensem sobre outros riscos que nós esquecemos.

25.5 REFINAMENTO DE RISCO

Durante os primeiros estágios de planejamento do projeto, um risco pode ser declarado de maneira bastante geral. À medida que o tempo passa e aumenta o conhecimento sobre o projeto e o risco, pode ser possível refinar o risco em um conjunto de riscos mais detalhados, cada um deles de algum modo mais fácil de atenuar, monitorar e administrar.

? Qual é uma boa maneira de descrever um risco?

Um modo de fazer isso é representar o risco no formato *condição-transição-conseqüência* (*condition-transition-consequence* — CTC) [GLU94]. Isto é, o risco é declarado da seguinte forma:

> Considerando que <condição> então há a preocupação de que (possivelmente) <conseqüência>.

Usando o formato CTC para o risco de reuso mencionado na Seção 25.4.2, podemos escrever:

> Considerando que os componentes de software reusáveis devem satisfazer padrões de projeto específicos, e que alguns não satisfazem, então há a preocupação de que (possivelmente) apenas 70% dos módulos reusáveis planejados possam efetivamente ser integrados ao sistema em construção, resultando na necessidade de fazer a engenharia sob medida dos 30% restantes dos componentes.

Essa condição geral pode ser refinada do seguinte modo:

> **Subcondição 1.** Certos componentes reusáveis foram desenvolvidos por terceiros sem conhecimento dos padrões internos de projeto.
>
> **Subcondição 2.** O padrão de projeto para as interfaces de componentes não foi consolidado e pode não estar de acordo com alguns componentes reusáveis existentes.
>
> **Subcondição 3.** Certos componentes reusáveis foram implementados em uma linguagem que não está disponível no ambiente-alvo.

As conseqüências associadas com essas subcondições refinadas permanecem as mesmas (por exemplo, 30% dos componentes de software devem ser submetidos à engenharia sob medida), mas o refinamento ajuda a isolar os riscos subjacentes e pode levar à análise e resposta mais fáceis.

25.6 ATENUAÇÃO, MONITORAÇÃO E GESTÃO DE RISCO

Todas as atividades de análise de risco apresentadas até aqui têm uma única meta — ajudar a equipe de projeto a desenvolver uma estratégia para lidar com o risco. Uma estratégia efetiva deve considerar três pontos:

- evitar risco.
- monitorar risco.
- gerenciar risco e planejar para a contingência.

Se uma equipe de software adota uma abordagem proativa para o risco, evitar é sempre a melhor estratégia. Isso é conseguido pelo desenvolvimento de um plano para atenuação de risco. Por exemplo, considere que a alta rotatividade de pessoal é notada como um risco de projeto, r_i. Com base no histórico anterior e na intuição da gerência, a probabilidade, l_i, de alta rotatividade é estimada como sendo de 0,70 (70%, bastante alta) e o impacto x_i é projetado como crítico. Isto é, a alta rotatividade terá impacto crítico no custo e no cronograma do projeto.

> "Se eu tomo tantas medidas de precaução, é porque não deixo nada para o acaso."
>
> Napoleão

Para atenuar o risco, o gerente de projeto deve desenvolver uma estratégia para reduzir a rotatividade. Entre os possíveis passos a serem tomados estão:

- Reúna-se com a equipe atual para determinar as causas da rotatividade (por exemplo, más condições de trabalho, baixa remuneração, mercado de trabalho competitivo).
- Atenue as causas que estão sob controle antes que o projeto se inicie.
- Uma vez iniciado o projeto, considere que a rotatividade vai ocorrer e desenvolva técnicas para assegurar a continuidade quando as pessoas saírem.

Document shared on www.docsity.com

- Organize equipes de projeto de modo que a informação sobre cada atividade de desenvolvimento seja amplamente difundida.
- Defina padrões de documentação e estabeleça mecanismos para assegurar que os documentos sejam desenvolvidos a tempo.
- Conduza revisões de todo o trabalho pelos pares (de modo que mais de uma pessoa esteja "por dentro").
- Designe um membro da equipe para dar retaguarda a cada técnico essencial.

À medida que o projeto avança, as atividades de monitoração de riscos se iniciam. O gerente de projeto monitora fatores que podem indicar se o risco é mais ou menos provável. No caso de alta rotatividade de pessoal, os seguintes fatores podem ser monitorados:

- Atitude geral dos membros da equipe com base nas pressões do projeto.
- Grau com que a equipe se aglutinou.
- Relacionamento interpessoal entre os membros da equipe.
- Problemas em potencial com remuneração e benefícios.
- Disponibilidade de emprego dentro da empresa e fora dela.

Além de monitorar esses fatores, o gerente de projeto deve monitorar a efetividade dos passos para atenuação de risco. Por exemplo, um passo para a atenuação de risco mencionado aqui pedia a definição de normas de documentação e de mecanismos para assegurar que a documentação seja desenvolvida a tempo. Esse é um mecanismo para garantir a continuidade, no caso de uma pessoa essencial deixar o projeto. O gerente de projeto deve monitorar cuidadosamente os documentos, para certificar-se de que cada um é auto-suficiente e que contém a informação que seria necessária se um novato fosse forçado a entrar para a equipe de software em algum ponto no meio do projeto.

Gestão de risco e planejamento de contingência considera que os esforços para atenuação falharam e que o risco tornou-se realidade. Continuando o exemplo, o projeto está bem avançado e algumas pessoas avisam que vão sair. Se a estratégia de atenuação foi seguida, o apoio está disponível, a informação está documentada e o conhecimento foi difundido por toda a equipe. Além disso, o gerente de projeto pode realocar recursos temporariamente (e reajustar o cronograma do projeto) para aquelas funções que estão com todo o pessoal necessário, permitindo que novatos, que tiverem que ser incorporados à equipe, "acertem o passo". As pessoas que estão saindo são solicitadas a interromper todo o trabalho e passar suas últimas semanas envolvidas em "transferência de conhecimento". Isso pode incluir captação de conhecimento baseada em vídeo, desenvolvimento de "documentos de comentário", e/ou reunião com outros membros da equipe que permanecerão no projeto.

AVISO

Se a exposição a um risco específico é menor que o custo da atenuação deste, não tente atenuar o risco, mas continue a monitorá-lo.

É importante notar que os passos de atenuação, monitoração e gestão de risco (RMMM) implicam custo adicional do projeto. Por exemplo, gastar tempo para "ter um substituto" para cada técnico essencial custa dinheiro. Assim, parte da gestão de risco é avaliar quando os benefícios, obtidos pelos passos RMMM, são superados pelos custos associados com sua implementação. Na essência, o planejador do projeto efetua uma análise custo/benefício clássica. Se os passos para atenuação do risco de alta rotatividade aumentarem o custo e o prazo do projeto em um valor estimado de 15%, mas o fator de custo predominante for "dar apoio", a gerência pode decidir não implementar esse passo. Por outro lado, se os passos para atenuação de risco têm previsão de aumentar o custo em 5% e o prazo em apenas 3%, a gerência vai provavelmente colocá-los em ação.

Para um projeto grande podem ser identificados 30 ou 40 riscos. Se de três a sete passos de gestão de risco são identificados para cada um, a gestão de risco pode se tornar um projeto em si mesma! Por essa razão, adaptamos a regra 80-20 de Pareto para o risco de software. A experiência indica que 80% de todo o risco do projeto (isto é, 80% do potencial de falha do projeto) pode ser responsável por apenas 20% dos riscos identificados. O trabalho realizado durante os primeiros passos de análise de risco vai ajudar o planejador a determinar quais dos riscos estão nesses 20%

(por exemplo, riscos que levam à maior exposição ao risco). Por esse motivo, alguns dos riscos identificados, avaliados e projetados podem não ir para o plano RMMM — eles não caem nos críticos 20% (os riscos com mais alta prioridade no projeto).

Veja na Web

Uma volumosa base de dados contendo todos os registros do fórum da ACM sobre riscos para o público pode ser encontrada no endereço **catless.ncl.ac.uk/Risks.**

O risco não é limitado ao projeto de software propriamente dito. Riscos podem ocorrer depois que o software foi desenvolvido com sucesso e entregue ao cliente. Esses riscos são tipicamente associados com as conseqüências da falha do software em campo.

Segurança de software e *análise de imprevistos* [LEV95] são atividades de garantia de qualidade de software (Capítulo 26) que focalizam a identificação e a avaliação de imprevistos em potencial, que podem afetar negativamente o software e causar falha de todo o sistema. Se imprevistos podem ser identificados antecipadamente no processo de engenharia de software, então, características do projeto de software que eliminam ou controlam os imprevistos em potencial podem ser especificadas.

25.7 O PLANO RMMM

Uma estratégia de gestão de risco pode ser incluída no plano do projeto de software, ou os passos de gestão de risco podem ser organizados em um plano separado de *Atenuação, Monitoração e Gestão de Risco*. O plano RMMM (*Risk Mitigation, Monitoring and Management*) documenta todo o trabalho realizado como parte da análise de risco e é usado pelo gerente do projeto como parte do plano geral de projeto.

Algumas equipes de software não desenvolvem um documento formal RMMM. Em vez disso, cada risco é documentado individualmente usando um *formulário de informação de risco* (*risk information sheet* — RIS) [WIL97]. Na maioria dos casos, o RIS é mantido por um sistema de banco de dados, de modo que a criação e a entrada da informação, a ordenação da prioridade, as buscas e outras análises podem ser realizadas facilmente. O formato do RIS é ilustrado na Figura 25.4.

FIGURA 25.4

Formulário de informação de risco [WIL97]

Formulário de informação de risco

Identificação do risco: P02-4-32	Data: 09/05/05	Probabilidade: 80%	Impacto: alto

Descrição:
Apenas 70% dos componentes de software programados para reuso serão, de fato, integrados na aplicação. A funcionalidade restante terá de ser desenvolvida sob medida.

Refinamento/contexto:
Subcondição 1: certos componentes resusáveis foram desenvolvidos por terceiros sem conhecimento dos padrões internos de projeto.
Subcondição 2: o padrão de projeto para interface de componentes ainda não foi consolidado e pode não estar de acordo com alguns componentes reusáveis.
Subcondição 3: certos componentes reusáveis foram implementados em uma linguagem não-disponível no ambiente-alvo.

Atenuação/monitoração:
1. Contate terceiros para determinar que os padrões de projeto sejam seguidos.
2. Force a conclusão dos padrões de interface; considere a estrutura do componente quando decidir sobre o protocolo da interface.
3. Verifique a quantidade de componentes que está na categoria da subcondição 3; procure determinar se o apoio para linguagem pode ser adquirido.

Administração/plano de contigência/disparo:
Exposição ao risco calculada como sendo de 20.200 dólares. Reserve essa quantia no custo de contingência do projeto. Desenvolva cronograma revisado considerando que mais de 18 componentes terão de ser construídos sob medida; distribua a equipe de acordo com isso.
Disparo: passos para atenuação de improdutivos em 01/07/05

Estado atual: 12/05/05: passos de atenuação iniciados

Emissor: D. Gagne	Assinado: B. Laster

Document shared on www.docsity.com

Uma vez documentado o plano RMMM e iniciado o projeto, os passos de atenuação e monitoração têm início. Como já discutimos, a atenuação de risco é uma atividade para evitar problemas. A monitoração de risco é uma atividade de acompanhamento de projeto com três objetivos principais: (1) avaliar se os riscos previstos de fato ocorrem; (2) assegurar que os passos de atenuação de risco definidos estão sendo adequadamente aplicados; e (3) coletar informação que pode ser usada para análise de risco futura. Em muitos casos, os problemas que ocorrem durante um projeto podem ser atribuídos a mais de um risco. Outro serviço da monitoração de risco é tentar atribuir origem (que risco[s] causou[aram] que problemas durante o projeto).

FERRAMENTAS DE SOFTWARE

Gestão de Risco

Objetivo: O objetivo das ferramentas de gestão de risco é apoiar uma equipe de projeto na definição de riscos, avaliando seu impacto e probabilidades, e monitorar riscos ao longo de um projeto de software.

Mecânica: Em geral, ferramentas de gestão de risco apóiam a identificação de risco genérica, fornecendo uma lista de riscos típicos de projeto e de negócio, fornecendo checklists ou outras técnicas de "entrevista"que apóiam a identificação de riscos específicos do projeto, atribuindo probabilidade e impacto a cada risco, apoiando as estratégias de atenuação de riscos e gerando vários diferentes relatórios relacionados a risco.

Ferramentas Representativas[4]

Riskman, desenvolvida por Arizona State University (*www.eas.asu.edu/sdm/merrill/riskman.html*), é um sistema especialista de avaliação de risco que identifica os riscos relacionados ao projeto.

Risk Radar, desenvolvida por SPMN (*www.spmn.com*), apóia os gerentes de projeto na identificação e gestão de riscos de projeto.

RiskTrak, desenvolvida por RST (*www.risktrac.com*), apóia a identificação, análise, relatos e gestão de riscos ao longo de um projeto de software.

Risk+, desenvolvida por C/S Solutions (*www.CSsolutions.com*), integra-se com Microsoft Project para quantificar custo e cronogramar incertezas.

X:PRIMER, desenvolvida por GrafP Technologies (*www.grafp.com*), é uma ferramenta genérica baseada na Web que prevê o que pode dar errado em um projeto e identifica as causas principais de falhas potenciais e medidas preventivas efetivas.

25.8 RESUMO

Sempre que há muita coisa em jogo em um projeto de software, o bom senso indica uma análise de risco. Todavia, a maioria dos gerentes de projeto de software a faz informal e superficialmente, se é que faz. O tempo empregado identificando, analisando e administrando riscos tem retorno de vários modos: menos transtornos durante o projeto, maior capacidade de acompanhar e controlar o projeto, e a confiança que advém do planejamento da resolução de problemas antes que eles ocorram.

A análise de risco pode absorver uma parcela significativa do esforço de planejamento do projeto. A identificação, previsão, avaliação, administração e monitoração, todas requerem tempo. Mas o esforço vale a pena. Para citar Sun Tzu, um general chinês que viveu há 2.500 anos: "Se você conhece o inimigo e conhece a si mesmo, não precisa temer o resultado de cem batalhas". Para o gerente de projeto de software, o inimigo é o risco.

REFERÊNCIAS BIBLIOGRÁFICAS

[AFC88] *Software Risk Abatement*, AFCS/AFLC Pamphlet 800-45, U.S. Air Force, 30 set. 1988.
[BOE89] Boehm, B. W., *Software Risk Management*, IEEE Computer Society Press, 1989.

4 As ferramentas mencionadas aqui não representam uma recomendação, mas em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

[CHA89] Charette, R. N., *Software Engineering Risk Analysis and Management*, McGraw-Hill/Intertext, 1989.
[CHA92] Charette, R. N., "Building Bridges over Intelligent Rivers", *American Programmer*, v. 5, n. 7, set. 1992, p. 2-9.
[DRU75] Drucker, P., *Management*, W. H. Heinemann, 1975.
[GIL88] Gilb, T., *Principles of Software Engineering Management*, Addison-Wesley, 1988.
[GLU94] Gluch, D. P., "A Construct for Describing Software Development Risks", CMU/SEI-94-TR-14, Software Engineering Institute, 1994.
[HAL98] Hall, E. M., *Managing Risk: Methods for Software Systems Development*, Addison-Wesley, 1998.
[HIG95] Higuera, R. P., "Team Risk Management", *CrossTalk, U.S.* Dept. of Defense, jan. 1995, p. 2-4.
[KAR96] Karolak, D. W., *Software Engineering Risk Management*, IEEE Computer Society Press, 1996.
[KEI98] Keil, M. et al., "A Framework for Identifying Software Project Risks", *CACM*, v. 41, n. 11, nov. 1998, p. 76-83.
[LEV95] Leveson, N. G., *Safeware: System Safety and Computers*, Addison-Wesley, 1995.
[ROW88] Rowe, W. D., *An Anatomy of Risk*, Robert E. Krieger Publishing Co., 1988.
[SEI93] "Taxonomy-Based Risk Identification", Software Engineering Institute, CMU/SEI-93-TR-6, 1993.
[THO92] Thomsett, R., "The Indiana Jones School of Risk Management", *American Programmer*, v. 5, n. 7, set. 1992, p. 10-18.
[WIL97] Williams, R. C., Walker, J. A. e Dorofee, A. J. "Putting Risk Management into Practice", *IEEE Software*, maio 1997, p. 75-81.

PROBLEMAS E PONTOS A CONSIDERAR

25.1. Dê cinco exemplos oriundos de outros campos que ilustrem os problemas associados com uma estratégia de risco reativa.

25.2. Descreva a diferença entre "riscos conhecidos" e "riscos previsíveis".

25.3. Acrescente três questões ou tópicos adicionais a cada um dos itens de risco da lista de verificação apresentada no site deste livro.

25.4. Foi solicitada a você a construção de software de apoio para um sistema de baixo custo de edição de vídeo. O sistema aceita videoteipe como entrada, armazena o vídeo em disco e depois permite ao usuário fazer um amplo conjunto de edições no vídeo digitalizado. O resultado pode então ser gravado em DVD ou outra mídia. Faça uma pesquisa sobre sistemas desse tipo e depois liste os riscos tecnológicos que enfrentaria ao iniciar um projeto desse tipo.

25.5. Você é um gerente de projeto de uma importante empresa de software. Foi solicitado a você que lidere uma equipe que está desenvolvendo software de processamento de palavra "de próxima geração". Crie uma tabela de risco para o projeto.

25.6. Descreva a diferença entre componentes de risco e fatores de risco.

25.7. Desenvolva uma estratégia de atenuação de risco e atividades específicas de atenuação de risco para três dos riscos mencionados na Figura 25.2.

25.8. Desenvolva uma estratégia de monitoração de riscos e atividades específicas de monitoração de risco para três dos riscos mencionados na Figura 25.2. Certifique-se de identificar os fatores que você vai monitorar para determinar se o risco está se tornando mais ou menos provável.

25.9. Desenvolva uma estratégia de gestão de risco e atividades específicas de gestão de risco para três dos riscos mencionados na Figura 25.2.

25.10. Tente refinar três dos riscos mencionados na Figura 25.2 e depois crie *folhas* de informação de risco para cada um.

25.11. Represente três dos riscos mencionados na Figura 25.2 usando o formato CTC.

25.12. Recalcule a exposição ao risco discutida na Seção 25.4.2 quando o custo/LOC é 16 dólares e a probabilidade é 60%.

25.13. Você pode pensar em uma situação em que um risco de alta probabilidade e alto impacto não seria considerado como parte do seu plano RMMM?

25.14. Descreva cinco áreas de aplicação de software nas quais a análise de segurança e imprevistos de software seriam uma preocupação importante.

Document shared on www.docsity.com

Leituras e Fontes de Informação Adicionais

A literatura sobre gestão de risco de software foi amplamente disseminada na última década. DeMarco e Lister (*Dancing with Bears*, Dorset House, 2003) escreveram um livro interessante e criterioso que guia os gerentes e desenvolvedores de software durante a gestão de risco. Moynihan (*Coping with IT/IS Risk Management*, Springer-Verlag, 2002) apresenta uma recomendação pragmática para gerentes de projeto que lidam com risco continuamente. Royer (*Project Risk Management*, Management Concepts, 2002) e Smith e Merritt (*Proactive Risk Management*, Productivity Press, 2002) sugerem um processo proativo para gestão de risco. Karolak (*Software Engineering Risk Management*, Wiley, 2002) escreveu um roteiro que introduz um modelo fácil de usar de análise de risco com checklists e questionários, apoiados por um pacote de software que são valiosos.

Schuyler (*Risk and Decision Analysis in Projects*, PMI, 2001) considera análise de risco de uma perspectiva estatística. Hall (*Managing Risk: Methods for Software Systems Development*, Addison-Wesley, 1998) apresenta uma das mais completas abordagens do assunto. Myerson (*Risk Management Processing for Software Engineering Models*, Artech House, 1997) considera métricas, segurança, modelos de processo e outros tópicos. Um instântaneo útil de avaliação de risco foi escrito por Grey (*Practical Risk Assessment for Project Management*, Wiley, 1995). Seu tratamento resumido fornece uma boa introdução ao assunto.

Capers Jones (*Assessment and Control of Software Risks*, Prentice-Hall, 1994) apresenta uma discussão detalhada de riscos de software que inclui dados coletados de centenas de projetos de software. Jones define 60 fatores de risco que podem afetar o resultado de projetos de software. Boehm [BOE89] sugere excelentes formatos de questionário e checklist que podem mostrar-se inestimáveis na identificação de risco. Charette [CHA89] apresenta um tratamento detalhado da mecânica de análise de risco, usando teoria de probabilidade e técnicas estatísticas para analisar riscos. Em um volume parecido Charette (*Application Strategies for Risk Analysis*, McGraw-Hill, 1990) discute risco no contexto tanto de sistemas quanto de engenharia de software e sugere estratégias pragmáticas para gestão de risco. Gilb (*Principles of Software Engineering Management*, Addison-Wesley, 1988) apresenta um conjunto de "princípios" (que são freqüentemente divertidos e algumas vezes profundos) que podem servir como um guia valioso de gestão de risco.

Ewusi-Mensah (*Software Development Failures: Anatomy of Abandoned Projects*, MIT Press, 2003) e Yourdon (*Death March*, Prentice-Hall, 1997) discutem o que acontece quando riscos engolfam uma equipe de projeto de software. Bernstein (*Against the Gods*, Wiley, 1998) apresenta uma história interessante de risco que vai até a Antigüidade.

O Software Engineering Institute publicou muitos relatórios detalhados e roteiros sobre análise e gestão de risco. O Air Force Systems Command criou o folheto AFSCP 800-45 [AFC88] que descreve técnicas de identificação e redução de risco. Todo número da *ACM Software Engineering Notes* tem uma seção intitulada "Risks to the Public" (editor, P. G. Neumann). Se você quiser as mais recentes e melhores histórias de horror de software, esse é o lugar para encontrar.

Uma ampla variedade de fontes de informação sobre gestão de risco de software está disponível na Internet. Uma lista atualizada de referências da World Wide Web que são relevantes pode ser encontrada no site deste livro: **http://www.mhhe.com/pressman**.

Gestão de Qualidade

CAPÍTULO 26

Conceitos-Chave

amostras 588
amplificação de defeitos .. 583
confiabilidade 592
controle de qualidade 579
custos da qualidade 579
custos de defeito 580
ISO 9001:2000 594
plano de SQA 595
qualidade 578
revisões (FTRs) 582
segurança de software 593
seis sigma 591
estatística SQA 589

A abordagem de engenharia de software descrita neste livro trabalha buscando uma única meta: produzir software de alta qualidade. Todavia, muitos leitores serão desafiados pela questão: "O que é qualidade de software?".

Philip Crosby [CRO79], no seu livro marcante sobre qualidade, fornece uma resposta singular para essa questão:

> O problema da gestão de qualidade não é o que as pessoas não sabem a respeito dela. O problema é o que elas pensam que sabem...
>
> Nesse sentido, a qualidade tem muito a ver com sexo. Todo mundo é a favor. (Sob certas circunstâncias, certamente.) Todo mundo se considera um entendido no assunto. (Mesmo que não queiram explicá-lo.) Todo mundo pensa que a execução é apenas uma questão de seguir as inclinações naturais. (Apesar de tudo, conseguimos de alguma forma.) E, certamente, a maioria das pessoas acha que problemas nessas áreas são causados pelos outros. (Como se somente eles usassem o tempo para fazer as coisas direito.)

Alguns dos desenvolvedores de software continuam a acreditar que qualidade de software é algo com que você começa a se preocupar depois que o código foi gerado. Nada poderia estar mais longe da verdade! *Gestão de qualidade* (*quality management*) (freqüentemente chamada *garantia de qualidade de software — software quality assurance*) é uma atividade guarda-chuva (Capítulo 2) que é aplicada ao longo do processo de software.

Gestão de qualidade abrange: (1) um processo de garantia de qualidade de software (*Software Quality Assurance* — SQA); (2) tarefas específicas de garantia de qualidade e controle de qualidade (incluindo revisões técnicas e estratégia de teste multicamadas); (3) prática de engenharia de software efetiva (métodos e ferramentas); (4) controle de todos os produtos de trabalho de software e das modificações feitas neles (Capítulo27); (5) um procedimento para garantir a satisfação de normas de desenvolvimento de software (quando aplicável) e (6) mecanismos de medição e relatório.

Panorama

O que é? Não é suficiente dizer que a qualidade de software é importante, você tem que (1) definir explicitamente o que quer dizer "qualidade de software", (2) criar um conjunto de atividades que ajudarão a garantir que todo o produto de trabalho de engenharia de software exibe alta qualidade, (3) realizar atividades de controle e garantia de qualidade de software em todo o projeto, (4) usar métricas para desenvolver estratégias para aperfeiçoar seu processo de software e, como conseqüência, a qualidade do produto final.

Quem faz? Todos os envolvidos no processo de engenharia de software são responsáveis pela qualidade.

Por que é importante? Você pode fazer direito ou fazer novamente. Se uma equipe de software enfatiza a qualidade em todas as atividades de engenharia de software, reduz a quantidade de trabalho que tem que refazer. Isso resulta em menores custos e, mais importante, melhor prazo para colocação no mercado.

Quais são os passos? Antes que as atividades de garantia da qualidade de software possam ser iniciadas, é importante definir "qualidade de software" em diferentes níveis de abstração. Entendido o que é qualidade, uma equipe de software deve identificar um conjunto de atividades de garantia de qualidade do software (software quality assurance — SQA), que filtrarão erros em produtos de trabalho antes que eles sejam passados adiante.

Qual é o produto do trabalho? Um Plano de Garantia de Qualidade de Software é criado para definir a estratégia de SQA de uma equipe de software. Durante a análise, projeto e geração de código, o principal produto do trabalho de SQA é o relatório resumido da revisão técnica formal. Durante o teste, planos e procedimentos de testes

Document shared on www.docsity.com

são produzidos. Outros produtos de trabalho associados com o aperfeiçoamento do processo podem também ser gerados.

Como tenho certeza de que fiz corretamente? Encontre erros antes que eles se tornem defeitos. Isto é, trabalhe para aperfeiçoar sua eficiência na remoção de defeitos (Capítulo 22), reduzindo assim a quantidade de trabalho a ser refeito que sua equipe de software tem que realizar.

Neste capítulo nós focalizamos os tópicos gerenciais e as atividades específicas de processo que permitem a uma organização de software garantir que faz as coisas certas no momento certo e do jeito certo.

26.1 Conceitos de Qualidade[1]

Ponto Chave

Controlar a variação é a chave para conseguir um produto de alta qualidade. No contexto de software, lutamos para controlar a variação no processo genérico que aplicamos e a ênfase em qualidade que permeia o trabalho de engenharia de software.

Controle de variação é o âmago do controle de qualidade. Um fabricante deseja minimizar a variação entre os produtos que são produzidos, mesmo quando faz algo relativamente simples como a duplicação de DVDs. Certamente, isso não pode constituir problema — duplicar DVDs é uma operação trivial de fabricação e podemos garantir que duplicatas exatas do software são sempre criadas.

Será que podemos? Precisamos garantir que as trilhas estão colocadas nos DVDs dentro da tolerância especificada, de modo que a imensa maioria dos "drives" de DVDs possa ler a mídia. As máquinas de duplicação de disco podem, e de fato se desgastam, e saem da tolerância. Assim, mesmo um processo "simples" como a duplicação de um DVD pode encontrar problemas devido à variação entre as amostras.

Mas como isso se aplica ao trabalho de software? Como é possível uma organização de desenvolvimento de software precisar controlar a variação? De um projeto para outro, desejamos minimizar a diferença entre os recursos previstos, necessários para completar um projeto, e os recursos usados de fato, incluindo pessoal, equipamento e tempo no calendário. Em geral, gostaríamos de nos certificar de que nosso programa de teste cobre uma porcentagem conhecida do software, de uma versão para outra. Desejamos não apenas minimizar a quantidade de defeitos que vão para o campo, gostaríamos também de nos certificar de que a variância no número de erros seja também minimizada de uma versão para outra. (Nossos clientes provavelmente vão ficar chateados se a terceira versão de um produto tem 10 vezes mais defeitos que a versão anterior.) Gostaríamos de minimizar as diferenças na velocidade e na precisão das nossas respostas, na linha telefônica (*hotline*) de apoio aos problemas dos clientes. A lista continua e continua.

26.1.1 Qualidade

O *American Heritage Dictionary* define *qualidade* como "uma característica ou atributo de alguma coisa". Como atributo de um item, a qualidade se refere a características mensuráveis — coisas que nós podemos comparar com padrões conhecidos tais como comprimento, cor, propriedades elétricas e maleabilidade. Todavia, o software, que é essencialmente uma entidade intelectual, é mais difícil de caracterizar do que objetos físicos.

No entanto, existem medidas das características de um programa. Essas propriedades incluem a complexidade ciclomática, a coesão, o número de pontos por função, de linhas de código e muitas outras propriedades discutidas no Capítulo 15. Quando examinamos um item baseado em suas características mensuráveis, duas espécies de qualidade podem ser encontradas: qualidade de projeto e qualidade de conformidade.

Qualidade de projeto refere-se a características que os projetistas especificam para um certo item. *Qualidade de conformidade* é o grau com que as especificações de projeto são seguidas durante a fabricação.

1 Esta seção, escrita por Michael Stovsky, foi adaptada de "Fundamentals of ISO 9000", um manual desenvolvido para *Essential Software Engineering*, um currículo em vídeo desenvolvido por R. S. Pressman & Associates, Inc. Reproduzido com permissão.

> "As pessoas esquecem quão rapidamente você fez um trabalho — mas elas sempre se lembram de quão bem você o fez."
>
> Howard Newton

No desenvolvimento de software, a qualidade de projeto abrange os requisitos, as especificações e o projeto do sistema. A qualidade de conformidade é um assunto concernente, principalmente, à implementação. Se a implementação segue o projeto e o sistema resultante satisfaz os requisitos e metas de desempenho, a qualidade de conformidade é alta.

Mas as qualidades de projeto e de conformidade são os únicos tópicos que os engenheiros de software devem considerar? Robert Glass [GLA98] alega que uma relação mais "intuitiva" é oportuna:

Satisfação do usuário ≈ produto adequado + máxima qualidade
+ entrega dentro do orçamento e do cronograma

No fundo, Glass argumenta que a qualidade é importante, mas se o usuário não está satisfeito, nada mais realmente importa. DeMarco [DEM99] reforça essa opinião quando afirma: "A qualidade de um produto corresponde a quanto ele muda o mundo para melhor". Essa visão da qualidade alega que se um produto de software fornece substancial benefício para seus usuários finais, eles podem estar dispostos a tolerar problemas eventuais de confiabilidade ou desempenho.

26.1.2 Controle de Qualidade

? O que é controle de qualidade de software?

O controle de variação pode ser equiparado ao controle de qualidade. Mas como se consegue controle de qualidade? Controle de qualidade envolve a série de inspeções, revisões e testes usada ao longo do processo de software para garantir que cada produto de trabalho satisfaça os requisitos para ele estabelecidos. O controle de qualidade inclui um ciclo de realimentação no processo de trabalho que criou o produto. A combinação de medida e realimentação nos permite ajustar o processo quando os produtos de trabalho criados deixam de satisfazer suas especificações.

Um conceito-chave do controle de qualidade é que todos os produtos de trabalho têm especificações definidas e mensuráveis com as quais nós podemos comparar o resultado de cada processo. O ciclo de realimentação é essencial para minimizar os defeitos produzidos.

26.1.3 Garantia da Qualidade

Veja na Web

Links úteis para recursos de SQA podem ser encontrados em www.qualitytree.com/links/index.htm.

Garantia da qualidade consiste de um conjunto de funções para auditar e relatar que avalia a efetividade e completeza das atividades de controle de qualidade. A meta da garantia da qualidade é fornecer à gerência os dados necessários para que fique informada sobre a qualidade do produto, ganhando assim compreensão e confiança de que a qualidade do produto está satisfazendo suas metas. É claro que, se os dados fornecidos por meio da garantia da qualidade identificam problemas, é responsabilidade da gerência cuidar dos problemas e aplicar os recursos necessários para resolver as questões da qualidade.

26.1.4 Custo da Qualidade

? Quais são os componentes do custo da qualidade?

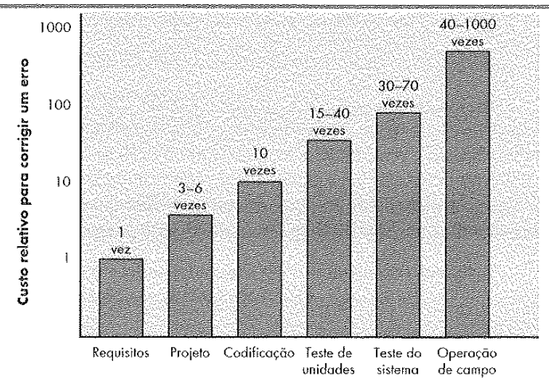
O custo da qualidade inclui todos os custos decorrentes da busca da qualidade ou da execução das atividades relacionadas à qualidade. Estudos de custo da qualidade são conduzidos para obter um referencial para os custos reais da qualidade, identificar as oportunidades para reduzir o custo da qualidade e fornecer uma base de comparação normalizada. A base de normalização é quase sempre a moeda local (ou dólares). Quando temos custos de qualidade normalizados na base do dólar, obtemos os dados necessários para avaliar onde estão as oportunidades e aperfeiçoar nossos processos. Além disso, podemos avaliar o efeito dos aperfeiçoamentos em termos de dólares.

Os *custos da qualidade* podem ser divididos em custos associados com a prevenção, com a avaliação e com as falhas. *Os custos de prevenção* incluem planejamento da qualidade, revisões técnicas formais, equipamento de teste, treinamento. *Os custos de avaliação* incluem atividades para obter entendimento da condição do produto na "primeira execução" de cada projeto. Exemplos de custos de avaliação incluem inspeção intra e interprocessos, calibração e manutenção do equipamento e teste.

Document shared on www.docsity.com

FIGURA 26.1

Custo relativo para corrigir um erro



AVISO

Não tenha medo de assumir custos de prevenção significativos. Fique tranqüilo que seu investimento terá excelente retorno.

Os custos de falha são aqueles que desapareceriam se nenhum defeito aparecesse antes de se entregar um produto ao cliente. Os custos de falha podem ser subdivididos em custos de falhas internas e custos de falhas externas. *Os custos de falhas internas* ocorrem quando detectamos um defeito no nosso produto antes do embarque. Os custos de falha interna incluem refazer, reparar, análise do modo como a falha ocorreu. *Os custos de falha externa* são associados com os defeitos encontrados depois que o produto foi enviado ao cliente. Exemplos de custos de falha externa são solução das queixas, devolução e substituição do produto, manutenção da linha de suporte, trabalho de garantia.

Como esperado, os custos relativos para encontrar e reparar um defeito aumentam significativamente à medida que migramos dos custos de prevenção para os de detecção de falhas internas até os de falhas externas. A Figura 26.1, baseada em dados coletados por Boehm [BOE81] e outros, ilustra esse fenômeno.

> "Leva menos tempo fazer uma coisa corretamente do que explicar por que ela foi feita errada."
>
> **H. W. Longfellow**

26.2 GARANTIA DA QUALIDADE DE SOFTWARE

Mesmo o mais exausto desenvolvedor de software vai concordar que software de alta qualidade é uma meta importante. Mas, como definimos qualidade? Um humorista disse certa vez: "Todo programa faz alguma coisa direito, apenas pode não ser a coisa que desejamos que ele faça".

? Como definimos qualidade de software?

Muitas definições de qualidade de software têm sido propostas na literatura. Para o nosso objetivo, *a qualidade de software* é definida como:

> Conformidade com requisitos funcionais e de desempenho explicitamente declarados, normas de desenvolvimento explicitamente documentadas e características implícitas, que são esperadas em todo software desenvolvido profissionalmente.

Há pouca dúvida de que essa definição poderia ser modificada ou estendida. De fato, uma definição definitiva de qualidade de software poderia ser debatida indefinidamente. Para a finalidade deste livro, a definição serve para enfatizar três pontos importantes:

1. Os requisitos de software são a base pela qual a qualidade é medida. A falta de conformidade com os requisitos é falta de qualidade.

Document shared on www.docsity.com

2. As normas especificadas definem um conjunto de critérios de desenvolvimento que guia o modo pelo qual o software é submetido à engenharia. Se os critérios não são seguidos, irá ocorrer, quase certamente, falta de qualidade.
3. Um conjunto de requisitos implícitos freqüentemente não é mencionado (por exemplo, o desejo de facilidade de uso e boa manutenibilidade). Se o software satisfaz seus requisitos explícitos, mas deixa de satisfazer os requisitos implícitos, a qualidade do software é suspeita.

26.2.1 Panorama Histórico

A garantia da qualidade é uma atividade essencial para qualquer negócio que faz produtos para serem usados por outros. Antes do século XX, a garantia da qualidade era de responsabilidade tão somente do artesão que construía um produto. A primeira função formal de garantia e controle de qualidade foi introduzida nos laboratórios Bell, em 1916, e disseminou-se rapidamente por todo o mundo industrializado. Durante os anos 40, abordagens mais formais para o controle de qualidade foram sugeridas. Elas se concentravam na medição e no aperfeiçoamento contínuo do processo [DEM86] como elementos-chave da gerência de qualidade.

> "Cometemos erros inadmissíveis em excesso."
>
> **Yogi Berra**

Hoje, toda empresa tem mecanismos para garantir a qualidade de seus produtos. Na realidade, declarações explícitas da preocupação da empresa com qualidade tornaram-se estratégia de mercado durante as últimas décadas.

A história da garantia da qualidade no desenvolvimento de software segue em paralelo com a história da qualidade na fabricação de hardware. Durante os primeiros tempos da computação (1950 e 1960), a qualidade era de responsabilidade tão-somente do programador. Normas para garantia da qualidade de software foram introduzidas em contratos militares de desenvolvimento de software durante os anos 70 e se disseminaram rapidamente pelo desenvolvimento de software no mundo comercial [IEE94]. Ampliando a definição antes apresentada, a garantia da qualidade de software é um "padrão planejado e sistemático de ações" [SCH98], que são necessárias para garantir a alta qualidade do software. A implicação para o software é que várias partes diferentes têm responsabilidade sobre a garantia da qualidade do software — engenheiros de software, gerentes de projeto, clientes, pessoal de vendas e indivíduos que servem em um grupo SQA.

O grupo SQA atua como o representante residente do cliente. Isto é, o pessoal que efetua SQA deve olhar o software do ponto de vista do cliente. O software satisfaz adequadamente os fatores de qualidade mencionados no Capítulo 15? O desenvolvimento de software foi conduzido de acordo com padrões preestabelecidos? As disciplinas técnicas desempenharam seu papel adequadamente como parte da atividade de SQA? O grupo SQA tenta responder a essas e a outras questões para garantir que a qualidade do software seja mantida.

26.2.2 Atividades de SQA

A garantia da qualidade de software é composta de uma variedade de tarefas associadas a duas partes diferentes — os engenheiros de software que fazem o trabalho técnico e um grupo de SQA, que tem responsabilidade pelo planejamento, supervisão, registro, análise e relato da garantia da qualidade.

Os engenheiros de software buscam qualidade (e desenvolvem atividades de garantia da qualidade e de controle de qualidade) aplicando métodos e medidas técnicas sólidas, conduzindo revisões técnicas formais e efetuando teste de software bem planejado. Apenas as revisões são discutidas neste capítulo. Os assuntos tecnológicos são discutidos nas Partes 1, 2, 3 e 5 deste livro.

A missão do grupo de SQA é ajudar a equipe de software a conseguir um produto final de alta qualidade. O Software Engineering Institute recomenda um conjunto de atividades de SQA que trata do planejamento, supervisão, registro, análises e relato da garantia da qualidade. Essas ativi-

dades são executadas (ou facilitadas) por um grupo independente de SQA que conduz as seguintes atividades:

? Qual é o papel de um grupo de SQA?

Prepara um plano SQA para um projeto. O plano é desenvolvido durante o planejamento do projeto e é revisado por todas as partes interessadas. As atividades de garantia da qualidade, realizadas pela equipe de engenharia de software e pelo grupo de SQA, são regidas pelo plano. O plano identifica avaliações a serem realizadas, auditorias e revisões a serem realizadas, normas que são aplicáveis ao projeto, procedimentos para relato e acompanhamento de erros, documentos a serem produzidos pelo grupo de SQA e quantidade de realimentação fornecida à equipe de projeto do software.

Participa no desenvolvimento da descrição do processo de software do projeto. A equipe de software seleciona um processo para o trabalho a ser realizado. O grupo de SQA revisa a descrição do processo para verificar a satisfação da política empresarial, padrões internos de software, padrões externamente impostos (por exemplo, ISO-9001) e outras partes do plano de projeto de software.

Revê as atividades de engenharia de software para verificar a satisfação do processo de software definido. O grupo de SQA identifica, documenta e acompanha desvios do processo e verifica se correções foram feitas.

Audita os produtos do trabalho de software encomendado para verificar a satisfação do que foi definido como parte do processo de software. O grupo de SQA revê produtos selecionados do trabalho, identifica, documenta e acompanha desvios, verifica se as correções são feitas e periodicamente relata os resultados do seu trabalho ao gerente do projeto.

Garante que os desvios do trabalho de software e dos produtos do trabalho sejam documentados e manipulados de acordo com um procedimento documentado. Os desvios podem ser encontrados no plano de projeto, na descrição do processo, nas normas aplicáveis ou nos produtos do trabalho técnico.

Registra qualquer eventual não-satisfação e a relata à gerência superior. Os itens que não atendem ao padrão são acompanhados até que sejam resolvidos.

Além dessas atividades, o grupo de SQA coordena o controle e a gestão das mudanças (Capítulo 27) e ajuda a coletar e analisar métricas de software.

26.3 Revisões de Software

AVISO

Revisões são como os filtros em um fluxo de trabalho de processo de software. Muito poucas, e o fluxo fica "sujo". Em excesso, e o fluxo míngua. Use métricas para determinar quais revisões funcionam e enfatize-as.

As revisões de software são um "filtro" para o processo de engenharia de software. Isto é, as revisões são aplicadas em vários pontos durante a engenharia de software e servem para descobrir erros e defeitos que podem depois ser removidos. As revisões de software "purificam" as atividades de engenharia de software, que chamamos análise, projeto e codificação. Freedman e Weinberg [FRE90] discutem a necessidade de revisões do seguinte modo:

> O trabalho técnico precisa de revisões, pela mesma razão que lápis precisam de borrachas: *errar é humano.* A segunda razão pela qual precisamos de revisões técnicas é que, apesar de as pessoas serem boas para encontrar alguns dos seus próprios erros, várias classes de erros escapam mais facilmente aos que as originaram do que escapam a outras pessoas.

Muitos tipos diferentes de revisões podem ser conduzidos como parte da engenharia de software. Cada qual tem seu lugar. Um encontro informal em torno de uma cafeteira é uma forma de revisão, se problemas técnicos são discutidos. Uma apresentação formal do projeto de software para uma audiência de clientes, gerência e pessoal técnico é também uma forma de revisão. Neste livro, todavia, focalizamos a *revisão técnica formal*, algumas vezes chamada de *"walkthrough"* ou de *inspeção*. Uma revisão técnica formal (*Formal Technical Review — FTR*) é o filtro mais efetivo do ponto de vista de garantia da qualidade. Conduzida por engenheiros de software (e outros) para engenheiros de software, a FTR é um meio efetivo de descobrir erros e aperfeiçoar a qualidade de software.

INFO

Falhas, Erros e Defeitos

O objetivo da SQA é remover problemas de qualidade no software. Esses problemas são referenciados por vários nomes – "*bugs*", "falhas", "erros", ou "defeitos" para citar alguns. Todos esses termos são sinônimos, ou há diferenças sutis entre eles?

Neste livro fizemos uma clara distinção entre um *erro* (um problema de qualidade encontrado *antes* que o software seja entregue aos usuários finais) e um *defeito* (um problema de qualidade encontrado somente *depois* que o software foi entregue aos usuários finais[2]). Fazemos essa distinção porque erros e defeitos têm impacto econômico, de negócio, psicológico e humano muito diferente. Como engenheiros de software, queremos encontrar e corrigir tantos erros quanto possível antes que o cliente e/ou usuário final os encontrem. Queremos evitar defeitos – porque defeitos (justificavelmente) fazem o pessoal de software aparecer mal. É importante observar, no entanto, que a distinção temporal feita entre erros e defeitos neste livro não é o pensamento da maioria. O consenso geral em uma comunidade de engenharia de software é que defeitos e erros, falhas e *bugs* são sinônimos. Isto é, a época em que o problema foi encontrado não tem a ver com o termo usado para descrever o problema. Parte do argumento em favor de nosso ponto de vista é que é algumas vezes difícil fazer uma clara distinção entre uma pré e pós-entrega (por exemplo, considere um processo incremental usado no desenvolvimento ágil [Capítulo 4]).

Independentemente de como você escolheu interpretar esses termos, reconheça que a época em que um problema é descoberto interessa e que os engenheiros de software devem tentar arduamente – muito arduamente – encontrar problemas antes que seus clientes e usuários finais os encontrem. Se você tiver mais interesse neste tópico, uma discussão mais detalhada da terminologia em torno de "*bugs*" pode ser encontrada em *www.software development.ca/bugs.shtml*.

AVISO

O objetivo principal de uma FTR é encontrar erros antes que eles sejam passados para outra atividade de engenharia de software ou entregues ao usuário final.

26.3.1 Impacto no Custo de Defeitos de Software

O objetivo principal das revisões técnicas formais é achar erros durante o processo, de modo que eles não se transformem em defeitos depois da entrega do software. O benefício óbvio das revisões técnicas formais é a descoberta antecipada de erros, de forma que eles não se propaguem para o passo seguinte do processo de software.

Vários estudos da indústria (pela TRW, Nippon Electric, Mitre Corp., entre outros) indicam que as atividades de projeto introduzem entre 50 e 65% de todos os erros (e em última análise, de todos os defeitos) durante o processo de software. Todavia, foi mostrado que as revisões técnicas formais são 75% efetivas [JON86] na descoberta de erros de projeto. Detectando e removendo uma alta porcentagem desses erros, o processo de revisão reduz substancialmente o curso dos passos subseqüentes nas fases de desenvolvimento e manutenção.

Para ilustrar o impacto do custo da detecção antecipada de erros, consideramos uma série de custos relativos, que são baseados em dados reais de custo coletados para grandes projetos de software [IBM81].[3] Considere que um erro descoberto durante o projeto vai custar 1,0 unidade monetária para ser corrigido. Relativamente a esse custo, o mesmo erro, descoberto imediatamente antes do início do teste, vai custar 6,5 unidades; durante o teste, 15 unidades; e depois da entrega, entre 60 e 100 unidades.

26.3.2 Amplificação e Remoção de Defeitos

Um modelo de amplificação de defeitos [IBM81] pode ser usado para ilustrar a geração e a detecção de erros durante os passos de projeto preliminar, de projeto detalhado e de codificação do processo de engenharia de software. O modelo é ilustrado esquematicamente na Figura 26.2. Uma caixa representa um passo de desenvolvimento de software. Durante o passo, erros podem ser gerados inadvertidamente. A revisão pode deixar de descobrir erros recém-gerados e erros advindos de

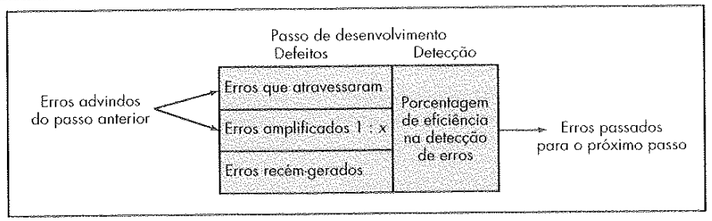
2 Se a melhoria do processo de software é considerada um problema da qualidade que é propagado de uma atividade de arcabouço de processo (por exemplo, modelagem) para outra (por exemplo, construção), pode também ser chamado de "defeito" (porque o problema deveria ter sido encontrado antes que um produto de trabalho (por exemplo, um modelo de projeto) fosse "liberado" para a atividade seguinte.

3 Apesar de esses dados terem mais de 20 anos, permanecem aplicáveis em um contexto moderno.

Document shared on www.docsity.com

FIGURA 26.2

Modelo de amplificação de defeitos



passos anteriores, resultando em um certo número de erros que passaram de um passo para outro. Em alguns casos, os erros que atravessam, vindos de passos anteriores, são amplificados (fator de amplificação, *x*) pelo trabalho atual. As subdivisões da caixa representam cada uma dessas características e uma porcentagem de eficiência na detecção de erros, que é função do rigor da revisão.

> "Algumas doenças, como os médicos dizem, são fáceis de curar no seu início, mas difíceis de reconhecer... mas com o decorrer do tempo, se não tiverem sido reconhecidas e tratadas no início, tornam-se fáceis de reconhecer, mas difíceis de curar."
>
> **Niccolo Machiavelli**

A Figura 26.3 ilustra um exemplo hipotético de amplificação de defeitos para um processo de desenvolvimento de software no qual não são feitas revisões. Com referência à figura, é considerado que cada passo de teste descobre e corrige 50% de todos os erros que chegam, sem introduzir nenhum erro novo (suposição otimista). Dez erros de projeto preliminar são amplificados para 94 erros, antes que o teste comece. Doze erros latentes vão para o campo. A Figura 26.4 considera as mesmas condições, exceto que revisões de projeto e códigos são conduzidas como parte de cada passo de desenvolvimento. Nesse caso, 10 erros iniciais de projeto preliminar são amplificados para 24 antes que o teste comece. Apenas três erros latentes existem. Recordando os custos relativos, associados com a descoberta e correção de erros, o custo total (com e sem revisões para o nosso exemplo hipotético) pode ser estabelecido. O número de erros descobertos durante cada um dos passos mostrados nas Figuras 26.3 e 26.4 é multiplicado pelo custo de remoção de um erro (1,5 unidade de custo para projeto, 6,5 unidades de custo antes do teste, 15 unidades de custo durante o teste e 67 unidades de custo após a entrega). Usando esses dados, o custo total, para desenvolvimento e manutenção, quando são feitas revisões, é de 783 unidades de custo. Quando não são feitas revisões, o custo total é de 2.177 unidades — quase três vezes maior.

Para realizar as revisões, o engenheiro de software precisa investir tempo e esforço, e a organização de desenvolvimento precisa gastar dinheiro. Todavia, os resultados do exemplo anterior

FIGURA 26.3

Amplificação de defeitos, sem revisões

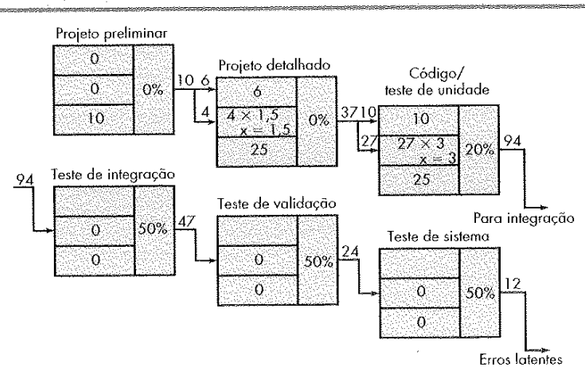
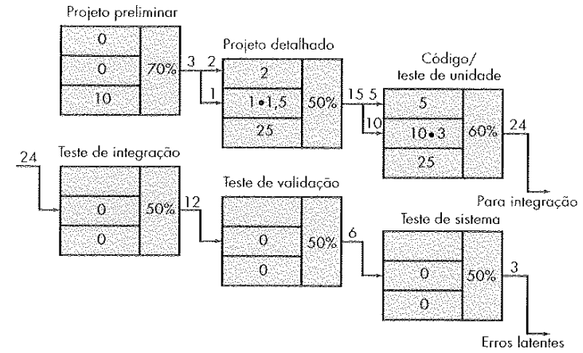


FIGURA 26.4

Amplificação de defeitos, com revisões



deixam pouca dúvida de que podemos pagar agora ou pagar muito mais depois. As revisões técnicas formais (para o projeto e outras atividades técnicas) fornecem um custo-benefício demonstrável. Elas devem ser realizadas.

26.4 REVISÕES TÉCNICAS FORMAIS

? Quando conduzimos FTR, quais são os nossos objetivos?

Uma revisão técnica formal é uma atividade de garantia da qualidade de software realizada por engenheiros de software (e outros). Os objetivos das FTR são (1) descobrir erros na função, na lógica ou na implementação, para qualquer representação do software; (2) verificar se o software sob revisão satisfaz seus requisitos; (3) garantir que o software tenha sido representado de acordo com padrões predefinidos; (4) conseguir software que seja desenvolvido de modo uniforme; e (5) tornar os projetos mais administráveis. Além disso, a FTR serve como uma oportunidade de treinamento, permitindo a jovens engenheiros observar abordagens diferentes para a análise, projeto e implementação de software. A FTR também serve para promover retaguarda e continuidade, porque algumas pessoas ficam familiarizadas com parte do software, que de outra forma poderiam não chegar a ver.

> "Não há nenhuma ânsia tão grande quanto a de um homem editar o trabalho de outro homem."
>
> **Mark Twain**

A FTR é na realidade uma classe de revisões que inclui *walkthroughs*, inspeções, revisões circulares ou "em rodízio" e outras avaliações técnicas de software feitas por pequenos grupos. Cada FTR é conduzida como uma reunião e será bem-sucedida apenas se for adequadamente planejada, controlada e assistida. Nas seções que se seguem, diretrizes semelhantes àquelas para um *walkthrough* (por exemplo, [FRE90], [GIL93]) são apresentadas como representativas de uma revisão técnica formal.

Veja na Web

O Guia de Inspeção Formal da Nasa SATC pode ser encontrado para download no endereço satc.gsfc.nasa.gov/fi/fipage.html.

26.4.1 A Reunião de Revisão

Independentemente do formato de FTR escolhido, cada reunião de revisão deve atender às seguintes restrições:

- Entre três e cinco pessoas (em geral) devem ser envolvidas na revisão. Preparativos devem ser feitos, os quais não devem exigir mais de duas horas de trabalho de cada pessoa.
- A duração da reunião de revisão deve ser inferior a duas horas.
- À vista dessas restrições, fica óbvio que uma FTR focaliza uma parte específica (e pequena) de todo o software. Por exemplo, em vez de tentar revisar todo um projeto, são conduzidos

Document shared on www.docsity.com

os *walkthroughs* de cada componente ou pequeno grupo de componentes. Restringindo o foco, a FTR tem maior probabilidade de descobrir erros.

PONTO CHAVE

A FTR focaliza uma parte relativamente pequena de um produto de trabalho.

O foco de uma FTR é um produto de trabalho (por exemplo, uma parte da especificação de requisitos, o projeto detalhado de um componente, uma listagem do código-fonte de um componente). O indivíduo que desenvolveu o produto de trabalho — o *produtor* — informa ao líder do projeto que o produto do trabalho foi completado e que é necessária uma revisão. O líder do projeto contata um *líder de revisão,* que avalia se o produto está efetivamente pronto, gera cópias dos materiais do produto e as distribui a dois ou três *revisores* para preparativos antecipados. Espera-se que cada revisor gaste entre uma e duas horas revisando o produto, tomando notas e ganhando familiaridade com o trabalho. Concomitantemente, o líder de revisão também revisa o produto e estabelece uma agenda para a reunião de revisão, que é usualmente marcada para o dia seguinte.

AVISO

Em algumas situações é uma boa idéia ter alguém, além do produtor, percorrendo o produto durante a revisão. Isso leva a uma interpretação literal do produto de trabalho e melhor reconhecimento de erro.

A reunião de revisão tem a participação do líder de revisão, de todos os revisores e do produtor. Um dos revisores assume o papel de *registrador,* isto é, o indivíduo que registra (por escrito) todas as questões importantes levantadas durante a revisão. A FTR começa com a introdução da agenda e uma breve introdução pelo produtor. O produtor então prossegue, "percorrendo" ("walk through") o produto de trabalho e explicando o material, enquanto os revisores levantam questões baseadas na sua preparação antecipada. Quando problemas ou erros válidos são descobertos, o registrador os anota.

No fim da revisão, todos os participantes da FTR devem decidir se (1) aceitam o produto sem maiores modificações, (2) rejeitam o produto devido a erros graves (ao serem corrigidos, outra revisão deve ser realizada), ou (3) aceitam o produto condicionalmente (erros triviais foram encontrados e devem ser corrigidos, mas não será necessária nova revisão). Tomada a decisão, todos os participantes da FTR assinam uma lista na qual indicam sua participação na revisão e sua concordância com os resultados da equipe de revisão.

26.4.2 Relatório e Manutenção de Registros das Revisões

Durante a FTR, um revisor (o registrador) registra ativamente todos os tópicos que foram levantados. Estes são resumidos no fim da reunião de revisão e uma *lista de tópicos de revisão* é produzida. Além disso, um *relatório resumido da revisão técnica formal* é completado. Um relatório resumido da revisão responde a três questões:

1. O que foi revisado?
2. Quem fez a revisão?
3. Quais foram as descobertas e conclusões?

O relatório resumido da revisão é um formulário de uma única página (com possíveis anexos). Ele se torna parte do registro histórico do projeto e pode ser distribuído ao líder do projeto e a outras partes interessadas.

A lista de questões da revisão serve a duas finalidades: (1) identificar áreas problemáticas no produto e (2) servir como checklist de itens de ação, que orienta o produtor à medida que as correções são efetuadas. Uma lista de tópicos é normalmente anexada ao relatório sumário.

É importante estabelecer um procedimento de acompanhamento para garantir que os itens na lista de tópicos tenham sido adequadamente corrigidos. A menos que isso seja feito, é possível que as questões levantadas possam "escapar pelas frestas". Uma abordagem é atribuir a responsabilidade do acompanhamento ao líder da revisão.

> "Uma reunião é freqüentemente um evento em que minutos são tomados e horas são desperdiçadas."
>
> **Autor desconhecido**

26.4.3 Diretrizes de Revisão

Diretrizes para a condução de revisões técnicas formais devem ser estabelecidas previamente, distribuídas a todos os revisores, receber a concordância de todos e depois ser seguidas. Uma revi-

são fora de controle pode freqüentemente ser pior do que nenhuma revisão. A seguir, um conjunto mínimo de diretrizes para revisões técnicas formais:

AVISO

Não mostre erros bruscamente. Um modo de fazê-lo gentilmente é formular uma questão, que permita ao produtor descobrir o erro.

1. *Revise o produto não o produtor.* Uma FTR envolve pessoas e egos. Conduzida adequadamente, a FTR deve deixar todos os participantes com um caloroso sentimento de realização. Conduzida inadequadamente, a FTR pode assumir a aura de inquisição. Os erros devem ser mostrados gentilmente; o tom da reunião deve ser descontraído e construtivo; o objetivo não deve ser embaraçar ou perturbar.
2. *Estabeleça uma agenda e siga-a.* Uma das doenças-chave das reuniões de todos os tipos é a derivação. Uma FTR deve ser mantida na linha e no cronograma. O líder de revisão é investido da responsabilidade de manter o cronograma da reunião e não deve temer repreender as pessoas quando a reunião fica "à deriva".
3. *Limite o debate e a réplica.* Quando um tópico é levantado por um revisor, pode não haver concordância de todos quanto ao seu impacto. Em vez de gastar tempo debatendo a questão, o tópico deve ser registrado para posterior discussão fora da reunião.
4. *Enuncie as áreas problemáticas, mas não tente resolver todo problema anotado.* Uma revisão não é uma seção de solução de problemas. A solução de um problema pode freqüentemente ser conseguida pelo produtor sozinho ou com a ajuda de apenas outro indivíduo. A solução do problema pode ser adiada até depois da reunião de revisão.
5. *Tome nota por escrito.* Algumas vezes é uma boa idéia para o registrador tomar nota em um quadro de parede, de modo que a redação e as prioridades possam ser avaliadas pelos outros revisores à medida que a informação é registrada.
6. *Limite o número de participantes e insista nos preparativos antecipados.* Duas cabeças são melhores do que uma, mas 14 não são necessariamente melhores do que 4. Mantenha o número de pessoas envolvidas no mínimo necessário. Todavia, todos os membros da equipe de revisão devem se preparar antecipadamente. Comentários escritos devem ser solicitados pelo líder de revisão (indicando que o revisor reviu o material).
7. *Desenvolva uma checklist para cada produto que será provavelmente revisado.* Uma checklist ajuda o líder de revisão a estruturar a reunião FTR e ajuda cada revisor a se concentrar em tópicos importantes.
8. *Aloque recursos e programe tempo para as FTR.* Para que as revisões sejam efetivas, elas devem ser cronogramadas como uma tarefa durante o processo de engenharia de software. Além disso, deve ser reservado tempo para as modificações inevitáveis que irão ocorrer como resultado de uma FTR.
9. *Faça treinamento significativo para todos os revisores.* Para serem efetivos, todos os participantes de uma revisão devem receber algum treinamento formal. O treinamento deve enfatizar tanto os assuntos relativos ao processo quanto o lado humano, psicológico, das revisões.
10. *Reveja suas primeiras revisões.* A análise de uma reunião de revisão pode ser benéfica para a descoberta de problemas no próprio processo de revisão. O primeiro produto a ser revisado deve ser as próprias diretrizes de revisão.

> "Uma das mais belas compensações da vida é que nenhum homem pode sinceramente tentar ajudar outro sem ajudar a si mesmo."
>
> **Ralph Waldo Emerson**

Como muitas variáveis (por exemplo, número de participantes, tipo de produtos de trabalho, horário e duração, abordagem específica de revisão) têm impacto sobre uma revisão bem-sucedida, uma organização de software deve experimentar para determinar qual abordagem funciona melhor em um contexto local. Porter e seus colegas [POR95] fornecem excelente diretriz para esse tipo de experimentação.

26.4.4 Revisões Guiadas por Amostras

Em um cenário ideal, todo produto de trabalho de engenharia de software deveria passar por uma revisão técnica formal. Nos projetos de software do mundo real, recursos são limitados e o

Document shared on www.docsity.com

prazo é curto. Como conseqüência, revisões são freqüentemente puladas, mesmo que seu valor seja reconhecido como um mecanismo de controle de qualidade. Thelin e seus colegas [THE01] tratam esse tópico quando afirmam:

> Inspeções [FTRs] são somente vistas como eficientes se muitas falhas forem encontradas durante a parte de busca por falha. Se muitas falhas são encontradas nos artefatos [produtos de trabalho], as inspeções são necessárias. Se, por outro lado, somente poucas falhas são encontradas, a inspeção foi um desperdício de tempo para muitas pessoas nela envolvida.[4] Além disso, projetos de software que estão atrasados freqüentemente diminuem o tempo das atividades de inspeção, o que leva à falta de qualidade. Uma solução seria estabelecer prioridades para uso dos recursos destinados às atividades de inspeção e, conseqüentemente, concentrar os recursos disponíveis nos artefatos que são mais propensos a erros.

AVISO

Revisões levam tempo, mas é tempo bem empregado. No entanto, se o tempo é curto e você não tem outra opção, não o gaste em revisões. Em vez disso, use revisões guiadas por amostras.

Thelin e seus colegas sugerem um processo de revisão baseado em amostra no qual amostras de todos os produtos de trabalho de engenharia de software são inspecionadas para determinar quais produtos de trabalho são mais propensos a erro. Recursos FTR completos são então enfocados somente naqueles produtos de trabalho que são provavelmente (baseados em dados coletados durante a amostragem) propensos a erro.

Para ser efetivo, o processo de revisão guiado por amostras deve tentar quantificar aqueles produtos de trabalho que são alvos principais para FTRs completas. Para conseguir isso, os seguintes passos são sugeridos [THE01]:

1. Inspecione uma fração a_i de cada produto de trabalho de software, i. Registre o número de falhas f_i encontrado em a_i.
2. Desenvolva uma estimativa grosseira do número de falhas em um produto de trabalho i pela multiplicação de f_i por $1/a_i$.
3. Ordene os produtos de trabalho em ordem descendente de acordo com a estimativa grosseira do número de falhas em cada um.
4. Enfoque os recursos de revisão disponíveis naqueles produtos de trabalho que têm o maior número estimado de falhas.

A fração dos produtos de trabalho que é amostrada deve (1) ser representativa do produto de trabalho como um todo e (2) suficientemente grande para ser significativa para o(s) revisor(es) que faz(em) a amostragem. À medida que a_i aumenta, a probabilidade de que a amostra seja uma representação válida do produto de trabalho também aumenta. No entanto, os recursos necessários para fazer amostragem também aumentam. Uma equipe de engenharia de software deve estabelecer o melhor valor de a_i para todos os tipos de produtos de trabalho produzidos[5].

CASASEGURA

Tópicos de SQA

A cena: Escritório de Doug Miller quando o projeto de software do *CasaSegura* tem início.

Os personagens: Doug Miller (gerente da equipe de engenharia de software do *CasaSegura*) e outros membros da equipe de engenharia de software.

A conversa:

Doug: Eu sei que não perdemos tempo desenvolvendo um plano de SQA para este projeto, mas já estamos nele e temos que considerar qualidade... certo?

Jamie: Lógico. Já decidimos isso quando desenvolvemos o modelo de requisitos [Capítulos 7 e 8], Ed comprometeu-se a desenvolver um procedimento V&V para cada requisito.

4 Naturalmente, pode-se alegar que ao conduzir revisões encorajamos os produtores a enfocar a qualidade, mesmo que nenhum erro seja encontrado.

5 Thelin e seus colegas conduziram uma simulação detalhada que pode apoiar essa determinação. Ver [THE01] para detalhes.

Doug: Isso é realmente bom, mas não vamos ficar esperando até o teste para avaliar a qualidade, vamos?

Vinod: Não! Lógico que não. Nós temos revisões programadas no plano de projeto para este incremento de software. Nós iniciaremos controle de qualidade com as revisões.

Jamie: Estou um pouco preocupada com não termos tempo suficiente para conduzir todas as revisões. De fato, eu sei que não vamos ter.

Doug: Hmmm. Então o que você propõe?

Jamie: Eu digo que devemos selecionar os elementos do modelo de análise e projeto que são mais críticos para o *CasaSegura* e os revisemos.

Vinod: Mas e se nós perdermos algo em uma parte do modelo que não revisamos?

Shakira: Eu li algo sobre uma técnica de amostragem [Seção 26.4.4] que poderia nos ajudar a escolher candidatos a revisão. (Shakira explica a abordagem.)

Jamie: Talvez... mas não estou certa de que tenhamos tempo até para amostrar cada elemento dos modelos.

Vinod: O que você quer que nós façamos, Doug?

Doug: Vamos roubar algo de Extreme Programming [Capítulo 4]. Desenvolvemos os elementos de cada modelo aos pares – duas pessoas – e conduzimos uma revisão informal de cada um, à medida que prosseguimos. Depois escolhemos os elementos "críticos" para uma revisão mais formal pela equipe, mas mantemos essas revisões em um mínimo. Desse modo, tudo é examinado por mais do que um conjunto de olhos, mas ainda mantemos nossas datas de entrega.

Jamie: Isso significa que nós vamos ter que revisar o cronograma.

Doug: Que seja. A qualidade ganha do cronograma neste projeto.

26.5 ABORDAGENS FORMAIS PARA SQA

Durante as duas últimas décadas, um pequeno mas ativo segmento da comunidade de engenharia de software tem argumentado que uma abordagem mais formal para a garantia da qualidade de software é necessária. Pode-se alegar que um programa de computador é um objeto matemático [SOM01]. Uma sintaxe e semântica rigorosas podem ser definidas para cada linguagem de programação e uma abordagem rigorosa para a especificação de requisitos de software (Capítulo 28) está disponível. Se o modelo de requisitos (especificação) e a linguagem de programação podem ser representados de um modo rigoroso, deve ser possível aplicar prova matemática de correção para demonstrar que um programa satisfaz exatamente suas especificações.

Tentativas para provar a correção (Capítulos 28 e 29) de programas não são novas. Dijkstra [DIJ76] e Linger, Mills e Witt [LIN79], entre outros, advogaram a favor das provas da correção de programas e as ligaram ao uso dos conceitos de programação estruturada (Capítulo 11).

26.6 GARANTIA ESTATÍSTICA DE QUALIDADE DE SOFTWARE

? Que passos são necessários para realizar SQA estatística?

Garantia estatística de qualidade reflete uma tendência crescente em toda a indústria, de tornar-se mais quantitativa a respeito da qualidade. Para o software, a garantia estatística de qualidade implica os seguintes passos:

1. Informação sobre defeitos de software é coletada e categorizada.
2. Uma tentativa de rastrear cada defeito até sua causa subjacente (por exemplo, não-conformidade com as especificações, erro de projeto, violação de normas, pouca comunicação com o cliente) é feita.
3. Usando o princípio de Pareto (80% dos defeitos podem ser rastreados até 20% de todas as causas possíveis), isole os 20% (os "poucos vitais").
4. Uma vez identificadas as poucas causas vitais, trate de corrigir os problemas que causaram os defeitos.

Document shared on www.docsity.com

Esse conceito relativamente simples representa um importante passo em direção à criação de um processo adaptativo de software no qual são feitas modificações para aperfeiçoar os elementos do processo que introduzem erro.

> "Vinte por cento do código têm 80% dos erros. Encontre-os, conserte-os!"
>
> Lowell Arthur

26.6.1 Um Exemplo Genérico

Para ilustrar o uso de métodos estatísticos no trabalho de engenharia de software, considere que uma organização de engenharia de software coleta informação sobre defeitos durante um período de um ano. Alguns dos defeitos são descobertos à medida que o software é desenvolvido. Outros são encontrados depois que o software foi entregue a seus usuários finais. Apesar de centenas de diferentes erros serem descobertos, todos podem ser rastreados até uma (ou mais) das seguintes causas:

- especificações incompletas ou errôneas (*Incomplete or Erroneous Specifications* — IES)
- má interpretação da comunicação com o cliente (*Misinterpretation of Customer Communication* — MCC)
- desvio intencional das especificações (*Intentional Deviation from Specifications* — IDS)
- violação das normas de programação (*Violation of Programming Standards* — VPS)
- erro na representação dos dados (*Error in Data Representation* — IDR)
- interface de componente inconsistente (*Inconsistent Component Interface* — ICI)
- erro na lógica do projeto (*Error in Design Logic* — EDL)
- teste incompleto ou errôneo (*Incomplete or Erroneous Testing* — IET)
- documentação imprecisa ou incompleta (*Inaccurate or Incomplete Documentation* — IID)
- erro na tradução do projeto para a linguagem de programação (*Error in Programming Language Translation of Design* — PLT)
- interface homem-computador ambígua ou inconsistente (*Ambiguous or Inconsistent Human/Computer Interface* — HCI)
- miscelânia (*MIScellaneous*, MIS)

Para aplicar SQA estatística, a tabela da Figura 26.5 foi construída. A tabela indica que IES, MCC e EDR são as poucas causas vitais que correspondem a 53% de todos os erros. Deve-se notar, todavia, que IES, EDR, PLT e EDL seriam selecionadas como as poucas causas vitais, se apenas fossem considerados erros sérios. Uma vez determinadas as poucas causas vitais, a organização de engenharia de software pode iniciar ação corretiva. Por exemplo, para corrigir MCC, o desenvolvedor de software poderia implementar técnicas facilitadas de coleta de requisitos (Capítulo 7) para aperfeiçoar a qualidade das especificações e da comunicação com o cliente. Para aperfeiçoar EDR, o desenvolvedor poderia adquirir ferramentas para a modelagem de dados e realizar revisões mais estritas do projeto dos dados.

É importante notar que a ação corretiva focaliza principalmente as poucas causas vitais. À medida que as poucas causas vitais são corrigidas, novas candidatas despontam no topo da pilha.

Tem sido mostrado que as técnicas de garantia estatística de qualidade de software propiciam substanciais aperfeiçoamentos na qualidade [ART97]. Em alguns casos, organizações de software conseguiram uma redução de 50% dos defeitos por ano, depois de aplicar essas técnicas.

A aplicação da SQA estatística e do princípio de Pareto podem ser resumidos em uma única sentença: *gaste seu tempo focalizando as coisas que realmente importam, mas primeiro esteja certo de que você compreende o que realmente importa!*

FIGURA 26.5

Coleta de dados para SQA estatística

	Total		Grave		Moderado		Trivial	
Erro	**Quantidade**	**%**	**Quantidade**	**%**	**Quantidade**	**%**	**Quantidade**	**%**
IES	205	22%	34	27%	68	18%	103	24%
MCC	156	17%	12	9%	68	18%	76	17%
IDS	48	5%	1	1%	24	6%	23	5%
VPS	25	3%	0	0%	15	4%	10	2%
EDR	130	14%	26	20%	68	18%	36	8%
ICI	58	6%	9	7%	18	5%	31	7%
EDL	45	5%	14	11%	12	3%	19	4%
IET	95	10%	12	9%	35	9%	48	11%
IID	36	4%	2	2%	20	5%	14	3%
PLT	60	6%	15	12%	19	5%	26	6%
HCI	28	3%	3	2%	17	4%	8	2%
MIS	56	6%	0	0%	15	4%	41	9%
Totais	942	100%	128	100%	379	100%	435	100%

Uma discussão abrangente da SQA estatística foge do escopo deste livro. Leitores interessados devem ver [GOH02], [SCH98] ou [KAN95].

26.6.2 Seis Sigma para Engenharia de Software

Seis Sigma (Six Sigma) é a estratégia mais amplamente usada para garantia de qualidade estatística na indústria hoje em dia. Originalmente popularizada pela Motorola na década de 1980, a estratégia Seis Sigma "é uma metodologia rigorosa e disciplinada que usa dados e análise estatística para medir e aperfeiçoar o desempenho operacional de uma empresa pela identificação e eliminação de 'defeitos' nos processos de fabricação e relacionados a serviço" [ISI03]. O termo "seis sigma" é derivado de seis desvios padrão — 3.4 instâncias (defeitos) por milhão de ocorrências — implicando uma norma de qualidade extremamente alta. A metodologia Seis Sigma define três passos centrais:

? Quais são os passos centrais da metodologia seis sigma?

- *Defina* os requisitos do cliente, os artefatos passíveis de entrega e os objetivos do projeto por meio de métodos bem definidos de comunicação com o cliente.
- *Meça* o processo existente e sua saída para determinar o atual desempenho de qualidade (colete métricas de defeitos).
- *Analise* métricas de defeito e determine as poucas causas vitais.

Se um processo de software existente está em ação, mas é necessário aperfeiçoamento, Seis Sigma sugere dois passos adicionais:

- *Aperfeiçoe* o processo pela eliminação das causas básicas dos defeitos.
- *Controle* o processo para garantir que o trabalho futuro não reintroduza as causas dos defeitos.

Esses passos centrais e adicionais são algumas vezes referidos como o método DMAIC (Defina [*Define*], Meça [*Measure*], Analise [*Analyse*], Melhore [*Improve*] e Controle [*Control*].

Se uma organização estiver desenvolvendo um processo de software (em vez de aperfeiçoando um processo existente), os passos centrais são ampliados como se segue:

- *Projete* o processo para (1) evitar as causas básicas de defeitos e (2) para satisfazer os requisitos do cliente.
- *Verifique* se o modelo de processo vai, de fato, evitar defeitos e satisfazer os requisitos do cliente.

Document shared on www.docsity.com

Essa variação é algumas vezes chamada de método DMADV (Defina [*Define*], Meça [*Measure*], Analise [*Analyze*], Projete [*Design*] e Verifique [*Verify*]).

Uma discussão abrangente de Seis Sigma está além do escopo deste livro. O leitor interessado deve ver [ISI03], [SNE03] e [PAN00].

26.7 Confiabilidade de Software

Confiabilidade de software, ao contrário de muitos outros fatores de qualidade, pode ser medida diretamente e estimada usando dados históricos e de desenvolvimento. *Confiabilidade de software* é definida em termos estatísticos como "a probabilidade de operação livre de falhas de um programa de computador, em um ambiente especificado, durante um tempo especificado" [MUS87]. Para ilustrar, é estimado que o programa X tenha uma confiabilidade de 0,96, por oito horas corridas de processamento. Em outras palavras, se o programa X tiver que ser executado 100 vezes e exigir oito horas de tempo corrido de processamento (tempo de execução), é provável que funcionará corretamente (sem falha) 96 das 100 vezes.

> "O inevitável preço da confiabilidade é a simplicidade."
>
> **C. A. R. Hoare**

Sempre que a confiabilidade de software é discutida, surge uma questão-chave: O que se quer dizer com o termo *falha?* No contexto de qualquer discussão de qualidade e confiabilidade de software, falha é a não-conformidade com os requisitos do software. Todavia, mesmo dentro dessa definição, há gradações. Falhas podem ser apenas incômodas ou catastróficas. Uma falha pode ser corrigida em segundos, enquanto outra exige semanas ou meses para ser corrigida. Complicando ainda mais o assunto, a correção de uma falha pode, na verdade, resultar na introdução de outros erros, que em última análise resultarão em outras falhas.

26.7.1 Medidas de Confiabilidade e Disponibilidade

PONTO CHAVE

Problemas de confiabilidade de software podem quase sempre ser rastreados até erros de projeto ou implementação.

Os primeiros trabalhos em confiabilidade de software tentaram extrapolar a matemática da teoria da confiabilidade do hardware (por exemplo, [ALV64]) para a previsão da confiabilidade de software. A maioria dos modelos de confiabilidade relacionados a hardware é voltada a falhas devidas a desgaste, em vez de falhas por causa de defeitos de projeto. No hardware, falhas devidas a desgaste físico (por exemplo, os efeitos de temperatura, de corrosão e de choque) são mais prováveis do que uma falha relacionada a projeto. Infelizmente, o oposto é verdadeiro para o software. De fato, todas as falhas de software podem ser rastreadas até problemas de projeto ou implementação; desgaste (ver Capítulo 1) não entra nesse quadro.

Tem havido debate sobre o relacionamento entre conceitos-chave de confiabilidade de hardware e sua aplicabilidade ao software (por exemplo, [LIT89], [ROO90]). Apesar de ter ainda que ser estabelecida uma ligação irrefutável, vale a pena considerar alguns conceitos simples que se aplicam a ambos os elementos do sistema.

PONTO CHAVE

É importante notar que MTBF e medidas relacionadas são baseadas em tempo de CPU, não em tempo de relógio de parede.

Se considerarmos um sistema baseado em computador, uma medida simples de confiabilidade é *tempo médio entre falhas* (*Mean-Time-Between-Failure* — MTBF), em que

$$MTBF = MTTF + MTTR$$

As siglas MTTF e MTTR significam *tempo médio até a falha* (*Mean-Time-To-Failure*) e *tempo médio de reparo*[6] (*Mean-Time-To-Repair*), respectivamente.

Muitos pesquisadores argumentam que MTBF é uma métrica muito mais útil do que defeitos/KLOC ou defeitos/FP. Dito de maneira simples, o usuário final está preocupado com falhas, não com

6 Embora depuração (e correções relacionadas) possam ser necessárias como conseqüência de falha, em muitos casos o software vai trabalhar adequadamente depois de um reinício sem nenhuma outra modificação.

AVISO

Alguns tópicos de disponibilidade (não discutidos aqui) não têm nada a ver com falhas. Por exemplo, tempo de parada programada (para funções de suporte) causa não-disponibilidade do software.

a contagem total de erros. Como cada erro contido em um programa não tem a mesma taxa de falha, a contagem total de erros fornece pouca indicação da confiabilidade de um sistema.

Em adição a uma medida de confiabilidade, visamos desenvolver uma medida de disponibilidade. *Disponibilidade de software* é a probabilidade de que um programa esteja operando de acordo com os requisitos em um dado momento, e é definida como

$$Disponibilidade = [MTTF/(MTTF + MTTR)] \times 100\%$$

A medida de confiabilidade MTBF é sensível igualmente a MTTF e MTTR. A medida de disponibilidade é de alguma forma mais sensível a MTTR, que é uma medida indireta da manutenibilidade de software.

26.7.2 Segurança de Software

Segurança de software [LEV86] é uma atividade de garantia da qualidade de software, que focaliza a identificação e a avaliação de riscos potenciais, que podem afetar o software negativamente e causar a falha de todo o sistema. Se esses riscos puderem ser identificados cedo, no processo de software, podem ser especificadas características do projeto de software que eliminem ou controlem esses riscos em potencial.

> "Eu não posso imaginar nenhuma condição que causaria o afundamento deste navio. A construção moderna de navios ultrapassou isso."
>
> **E. J. Smith, capitão do Titanic**

Um processo de modelagem e análise é conduzido como parte da segurança de software. Inicialmente, os riscos são identificados e categorizados por criticalidade e risco. Por exemplo, alguns dos riscos associados com um controle de cruzeiro, para um automóvel baseado em computador, poderiam ser:

- causa aceleração descontrolada que não pode ser interrompida.
- não reage ao acionamento do pedal do freio (desligando).
- não liga quando a chave é ativada.
- lentamente perde ou ganha velocidade.

Veja na Web

Uma coleção de artigos valiosos sobre segurança de software pode ser encontrada em www.safeware-eng.com/.

Uma vez identificados esses riscos do sistema, são usadas técnicas de análise para atribuir severidade e probabilidade de ocorrência.[7] Para ser eficaz, o software precisa ser analisado no contexto de todo o sistema. Por exemplo, um erro sutil de entrada do usuário (pessoas são componentes do sistema) pode ser ampliado por uma falha de software para produzir dados de controle que posicionam inadequadamente um dispositivo mecânico. Se um conjunto de condições ambientais externas ocorre (e somente se elas ocorrem), a posição inadequada do dispositivo mecânico vai causar uma falha desastrosa. Técnicas de análise, como análise de falha em árvore [VES81], lógica de tempo real [JAN86] ou modelos em rede de Petri [LEV87] podem ser usadas para prever a cadeia de eventos que pode causar riscos e a probabilidade de cada evento ocorrer para criar a cadeia.

Uma vez identificados e analisados os riscos, requisitos relacionados a segurança podem ser especificados para o software. Isto é, a especificação pode conter uma lista de eventos indesejáveis e as respostas desejadas do sistema a esses eventos. O papel do software na gestão de eventos indesejáveis é então indicado.

Apesar de a confiabilidade de software e da segurança de software serem intimamente relacionadas entre si, é importante entender a diferença sutil entre elas. Confiabilidade de software usa análise estatística para determinar a probabilidade de que uma falha do software ocorra. Todavia, a ocorrência de uma falha não necessariamente resulta em um risco ou infortúnio. A segurança de software examina os modos pelos quais as falhas resultam em condições que podem levar ao

7 Essa abordagem é análoga aos métodos de análise de risco descritos no Capítulo 25. A diferença principal está na ênfase e nos tópicos tecnológicos em contraposição aos tópicos relacionados ao projeto.

Document shared on www.docsity.com

infortúnio. Isto é, as falhas não são consideradas em um vazio, mas são avaliadas no contexto de todo o sistema baseado em computador e seu ambiente. Os leitores com maior interesse devem ver o livro de Leveson [LEV95] sobre o assunto.

26.8 As Normas de Qualidade ISO 9000[8]

PONTO CHAVE

A ISO 9000 descreve o que deve ser feito para ficar em conformidade, mas não descreve como isso deve ser feito.

Veja na Web

Diversos links para os recursos ISO 9000/9001 podem ser encontrados em **www.tantara.ab.ca/info.htm.**

Um *sistema de garantia da qualidade* pode ser definido como a estrutura organizacional, responsabilidades, procedimentos, processos e recursos para implementar a gestão de qualidade [ANS87]. Sistemas de garantia da qualidade são criados para ajudar organizações a garantir que seus produtos e serviços satisfaçam as expectativas do cliente, estando de acordo com suas especificações. ISO 9000 descreve os elementos de garantia de qualidade em termos genéricos, que podem ser aplicados a qualquer negócio independentemente dos produtos ou serviços oferecidos.

Para se registrar em um dos modelos de sistema de garantia de qualidade contidos na ISO 9000, o sistema e operações de qualidade da empresa são escrutinados por auditores externos para verificação do atendimento das normas e da efetividade da operação. Após o registro bem-sucedido, a empresa recebe um certificado de um corpo de registro representado pelos auditores. Auditorias de vigilância semestrais garantem a continuidade de conformidade com as normas.

Os requisitos delineados pela ISO 9001:2000 tratam de tópicos, tais como responsabilidade de gestão, sistema de qualidade, revisão de contrato, controle de projeto, controle de documento e dados, identificação de produtos e rastreabilidade, controle de processo, inspeção e teste, ação corretiva e preventiva, registros de controle de qualidade, auditorias de qualidade interna, treinamento, serviço e técnicas estatísticas. Para uma organização de software tornar-se certificada pela ISO 9001:2000, ela deve estabelecer políticas e procedimentos para tratar cada um dos requisitos mencionados (e outros) e depois ser capaz de demonstrar que essas políticas e procedimentos estão sendo seguidos. Para mais informação sobre ISO 9001, o leitor interessado deve ver [HOY02], [GAA01] ou [CIA01].

INFO

A Norma ISO 9001:2000

A seguinte diretriz define os elementos básicos da norma ISO 9001:2000. Informação abrangente sobre a norma pode ser obtida de International Organization for Standardization (*www.iso.ch*) e outras fontes da Internet (por exemplo, *www.praxiom.com*).

Estabelecer os elementos de um sistema de gestão de qualidade.
- Desenvolver, implementar e aperfeiçoar o sistema.
- Definir uma política que enfatiza a importância do sistema.

Documentar o sistema de qualidade.
- Descrever o processo.
- Produzir um manual operacional.
- Desenvolver métodos para controlar (atualizar) documentos.
- Estabelecer métodos para manter registros.

Apoiar o controle e a garantia de qualidade.
- Promover a importância da qualidade entre todos os interessados.
- Enfocar a satisfação do cliente.
- Definir um plano de qualidade que atenda aos objetivos, às responsabilidades e à autoridade.
- Definir mecanismos de comunicação entre os interessados.

Estabelecer mecanismos de revisão para sistema de gestão de qualidade.
- Identificar métodos de revisão e mecanismos de realimentação.
- Definir procedimentos de acompanhamento.

Identificar recursos de qualidadde incluindo pessoal, treinamento e elementos de infra-estrutura.

Estabelecer mecanismos de controle.
- Para planejamento.
- Para requisitos de cliente.
- Para atividades técnicas (por exemplo, análise, projeto e teste).
- Para monitoramento e gestão de projeto.

Definir métodos para reparação.

Avaliar dados e métricas de qualidade.

Definir abordagem para aperfeiçoamento contínuo de processo e qualidade.

8 Essa seção, escrita por Michael Stovsky, foi adaptada de "Fundamentals of ISO 9000", manual de trabalho desenvolvido para *Essential Software Engineering*, um curso em vídeo preparado por R. S. Pressman & Associates, Inc. Reproduzido com permissão.

26.9 O Plano de SQA

O *Plano de SQA* fornece um roteiro para a instituição da garantia da qualidade de software. Desenvolvido pelo grupo de SQA (ou equipe de software se um grupo SQA não existir), o plano serve como gabarito para as atividades SQA que são instituídas para cada projeto de software.

Uma norma para planos de SQA foi recomendada pelo IEEE [IEE94]. A norma recomenda uma estrutura que identifique (1) o objetivo e o escopo do plano; (2) uma descrição de todos os produtos de trabalho de engenharia de software (por exemplo, modelos, documentos, código-fonte) que ficam no âmbito da SQA; (3) todas as normas e práticas aplicáveis que são aplicadas durante o processo de software; (4) ações e tarefas de SQA (inclusive revisões e auditorias) e sua colocação ao longo do processo de software; (5) as ferramentas e os métodos que apóiam as ações e tarefas de SQA; (6) procedimentos de gestão de configuração de software (Capítulo 27) para gestão de modificação; (7) métodos para montagem, proteção e manutenção de todos os registros relacionados a SQA; e (8) papéis e responsabilidades organizacionais relativos à qualidade do produto.

FERRAMENTAS DE SOFTWARE

Gestão de Qualidade de Software

Objetivo: O objetivo das ferramentas de SQA é apoiar uma equipe de projeto na avaliação e no aperfeiçoamento da qualidade de produto de trabalho de software.

Mecânica: A mecânica das ferramentas varia. Em geral, o objetivo é avaliar a qualidade de um produto de trabalho específico. Obs.: uma vasta gama de ferramentas de teste de software (ver Capítulos 13 e 14) é freqüentemente incluída na categoria de ferramentas de SQA.

Ferramentas Representativas[9]

ARM, desenvolvida pela NASA (**satc.gsfc.nasa.gov/tools/index.html**), fornece medidas que podem ser usadas para avaliar a qualidade de um documento de requisitos de software.

QPR ProcessGuide and Scorecard, desenvolvida por QPRSoftware (**www.qpronline.com**), fornece apoio para Seis Sigma e outras abordagens de gestão de qualidade.

Quality Tools Cookbook, desenvolvida por Systma e Manley (**www.sytsma.com/tqmtools/tqmtoolmenu.html**), fornece descrições úteis de ferramentas de gestão clássica de qualidade tal como diagramas de controle, diagramas de espalhamento, diagramas de afinidade e diagramas matriciais.

Quality Tools and Templates, desenvolvida por iSixSigma (**http://www.isixsigma.com/tt/**), descreve uma ampla gama de ferramentas e métodos úteis para gestão de qualidade.

TQM Tools, desenvolvida por Bain & Company (www.bain.com), fornece descrições úteis de uma variedade de ferramentas de gestão usadas para TQM e métodos de gestão de qualidade relacionados.

9 As ferramentas mencionadas aqui não representam uma recomendação, mas em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

Document shared on www.docsity.com

26.10 Resumo

Garantia de qualidade de software é uma atividade guarda-chuva — que incorpora controle de qualidade e garantia de qualidade — que é aplicada em cada passo do processo de software. SQA abrange procedimentos para a aplicação efetiva de métodos e ferramentas, revisões técnicas formais, estratégias e técnicas de teste, procedimentos para controle de modificações, procedimentos para garantir o cumprimento de normas e mecanismos de medição e preparação de relatórios.

SQA é complicada pela natureza complexa da qualidade de software — um atributo de programas de computador, que é definido como "satisfação de requisitos especificados implícita e explicitamente". Mas quando considerada de maneira mais geral, a qualidade de software abrange muitos fatores de produto e de processo diferentes, e métricas relacionadas.

Revisões de software é uma das mais importantes atividades de controle de qualidade. Revisões servem como filtros ao longo de todas as atividades de engenharia de software, removendo erros enquanto eles ainda são relativamente baratos de encontrar e corrigir. A revisão técnica formal é uma reunião estilizada que tem se mostrado extremamente efetiva na descoberta de erros.

Para conduzir adequadamente a garantia de qualidade de software, dados sobre o processo de engenharia de software devem ser coletados, avaliados e disseminados. SQA estatística ajuda a aperfeiçoar a qualidade do produto e o próprio processo de software. Modelos de confiabilidade de software ampliam as medições, permitindo que os dados de defeitos coletados sejam extrapolados para taxas projetadas de falha e previsões de confiabilidade.

Em resumo, lembramos as palavras de Dunn e Ullman [DUN82]: "Garantia da qualidade de software é a aplicação dos preceitos gerenciais e disciplinas de projeto de garantia de qualidade, no espaço gerencial e tecnológico aplicável de engenharia de software". A capacidade de garantir qualidade é a medida de uma disciplina de engenharia amadurecida. Quando a aplicação é alcançada com sucesso, o resultado é uma engenharia de software amadurecida.

Referências Bibliográficas

[ALV64] Alvin, W. H., von (ed.), *Reliability Engineering*, Prentice-Hall, 1964.

[ANS87] ANSI/ASQC A3-1987, *Quality Systems Terminology*, 1987.

[ART92] Arthur, L. J., *Improving Software Quality: An Insider's Guide to TQM, Wiley*, 1992.

[ART97] Arthur, L. J., "Quantum Improvements in Software System Quality", *CACM*, v. 40, n. 6, jun. 1997, p. 47-52.

[BOE81] Boehm, B., *Software Engineering Economics*, Prentice-Hall, 1981.

[CIA01] Cianfrani, C. A. et al., *ISO 9001:2000 Explained*, 2. ed., American Society for Quality, 2001.

[CRO79] Crosby, P., *Quality Is Free*, McGraw-Hill, 1979.

[DEM86] Deming, W. E., *Out of the Crisis*, MIT Press, 1986.

[DEM99] DeMarco, T., "Management Can Make Quality (Im)possible", Cutter IT Summit, Boston, abr. 1999.

[DIJ76] Dijkstra, E., *A Discipline of Programming*, Prentice-Hall, 1976.

[DUN82] Dunn, R. e Ullman, R., *Quality Assurance for Computer Software*, McGraw-Hill, 1982.

[FRE90] Freedman, D. P. e Weinberg, G. M., *Handbook of Walkthroughs, Inspections and Technical Reviews*, 3. ed., Dorset House, 1990.

[GAA01] Gaal, A., *ISO 9001:2000 for Small Business*, Saint Lucie Press, 2001.

[GIL93] Gilb, T. e Graham, D. *Software Inspections*, Addison-Wesley, 1993.

[GLA98] Glass, R., "Defining Quality Intuitively", *IEEE Software*, maio 1998, p. 103-104, 107.

[GOH02] Goh, T., Kuralmani, V. e Xie, M., *Statistical Models and Control Charts for High Quality Processes*, Kluwer Academic Publishers, 2002.

[HOY02] Hoyle, D., *ISO 9000 Quality Systems Development Handbook: A Systems Engineering Approach*, 4. ed., Butterworth-Heinemann, 2002.

[IBM81] "Implementing Software Inspections", course notes, IBM Systems Sciences Institute, IBM Corporation, 1981.

[IEE94] *Software Engineering Standards*, 1994, IEEE Computer Society, 1994.

[ISI03] iSixSigma, LLC, "New to Six Sigma: A Guide for Both Novice and Experienced Quality Practitioners", 2003, disponível em http://www.isixsigma.com/library/content/six-sigmanewbie. asp.

[JAN86] Jahanian, F. e Mok, A. K. "Safety Analysis of Timing Properties of Real-Time Systems", *IEEE Trans. Software Engineering*, v. SE-12, n. 9, set. 1986, p. 890-904.

[JON86] Jones, T. C., *Programming Productivity*, McGraw-Hill, 1986.

[KAN95] Kan, S. H., *Metrics and Models in Software Quality Engineering*, Addison-Wesley, 1995.

[LEV86] Leveson, N. G., "Software Safety: Why, What, and How", *ACM Computing Surveys*, v. 18, no. 2, jun. 1986, p. 125-163.

[LEV87] Leveson, N. G. e Stolzy, J. L., "Safety Analysis Using Petri Nets", *IEEE Trans. Software Engineering*, v. SE-13, n. 3, março 1987, p. 386-397.

[LEV95] Leveson, N. G., *Safeware: System Safety and Computers*, Addison-Wesley, 1995.

[LIN79] Linger, R., Mills, H. e Witt, B. *Structured Programming*, Addison-Wesley, 1979.

[LIT89] Littlewood, B., "Forecasting Software Reliability", in *Software Reliability: Modeling and Identification*, (S. Bittanti, ed.), Springer-Verlag, 1989, p. 141-209.

[MUS87] Musa, J. D., Iannino, A. e Okumoto, K. *Engineering and Managing Software with Reliability Measures*, McGraw-Hill, 1987.

[PAN00] Pande, P. et al., *The Six Sigma Way*, McGraw-Hill, 2000.

[POR95] Porter, A., Siy, H. Toman, C. A. e Votta, L. G., "An Experiment to Assess the Cost-Benefits of Code Inspections in Large Scale Software Development", *Proc. Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Washington, D.C., out. 1995, ACM Press, p. 92-103.

[ROO90] Rook, J., *Software Reliability Handbook*, Elsevier, 1990.

[SCH98] Schulmeyer, G. C. e McManus, J. I. (eds.), *Handbook of Software Quality Assurance*, 3. ed., Prentice-Hall, 1998.

[SOM01] Somerville, I., *Software Engineering*, 6. ed., Addison-Wesley, 2001.

[SNE03] Snee, R. e Hoerl, R. *Leading Six Sigma*, Prentice-Hall, 2003.

[THE01] Thelin, T., Petersson, H. e Wohlin, C., "Sample Driven Inspections", *Proceedings Workshop on Inspection in Software Engineering (WISE'01)*, Paris, France, jul. 2001, p. 81-91, pode ser obtido em http://www.cas.mcmaster.ca/wise/wise01/ThelinPetersson-Wohlin.pdf.

[VES81] Veseley, W. E. et al., *Fault Tree Handbook*, U.S. Nuclear Regulatory Commission, NUREG-0492, jan. 1981.

Problemas e Pontos a Considerar

26.1. Anteriormente neste capítulo mencionamos que "controle de variação é o coração do controle de qualidade". Como todo programa que é criado é diferente de qualquer outro programa, quais são as variações que estamos procurando e como as controlamos?

26.2. É possível avaliar a qualidade de software se o cliente muda continuamente o que deve fazer?

26.3. Qualidade e confiabilidade são conceitos relacionados, mas são fundamentalmente diferentes em vários ângulos. Discuta-os.

26.4. Um programa pode estar correto e ainda assim não ser confiável? Explique.

26.5. Um programa pode estar correto e ainda assim não exibir boa qualidade? Explique.

26.6. Por que freqüentemente há tensão entre um grupo de engenharia de software e um grupo independente de garantia de qualidade de software? Isso é saudável?

26.7. Você recebeu a responsabilidade de aperfeiçoar a qualidade de software na sua organização. Qual a primeira coisa que deve fazer? Qual a seguinte?

26.8. Além da contagem de erros e defeitos, há outras características enumeráveis do software que implicam qualidade? Quais são elas e podem ser medidas diretamente?

26.9. Uma revisão técnica formal é efetiva apenas se todos tiverem se preparado previamente. Como você reconhece um participante da revisão que não está preparado? Se você for o líder de revisão o que faz nessa circunstância?

26.10. Algumas pessoas argumentam que uma FTR deve avaliar o estilo da programação bem como a sua correção. Essa é uma boa idéia? Por quê?

26.11. Veja novamente a tabela apresentada na Figura 26.5 e selecione quatro das poucas causas vitais de erros sérios e moderados. Sugira ações corretivas usando as informações apresentadas em outros capítulos.

26.12. Pesquise a literatura sobre confiabilidade de software e faça uma monografia que descreva um modelo de confiabilidade de software. Não deixe de fornecer exemplo.

26.13. O conceito MTBF de software está aberto a críticas. Você pode pensar em algumas razões para isso?

26.14. Considere dois sistemas de segurança crítica, que são controlados por computadores. Liste pelo menos três perigos para cada um que possa ser ligado diretamente a falhas de software.

Document shared on www.docsity.com

26.15. Consiga uma cópia da ISO 9001:2000 e ISO 9000-3. Prepare uma apresentação que discuta três requisitos da ISO 9001 e como eles se aplicam em um contexto de software.

Leituras e Fontes de Informação Adicionais

Livros de Moriguchi (*Software Excellence: A Total Quality Management Guide*, Productivity Press, 1997) e Horch (*Practical Guide to Software Quality Management*, Artech Publishing, 1996) são excelentes apresentações em nível gerencial dos benefícios dos programas de garantia formal de qualidade para software de computador. Livros de Deming [DEM86], Juran (*Juran on Quality by Design*, Free Press, 1992), Crosby [CRO79] e *Quality Is Still Free*, McGraw-Hill, 1995) não se concentram no software, mas são leitura obrigatória para gerentes de nível superior, com responsabilidade sobre o desenvolvimento de software. Gluckman e Roome (*Everyday Heroes of the Quality Movement*, Dorset House, 1993) humanizam aspectos de qualidade contando a história de personagens no processo de qualidade. Kan (*Metrics and Models in Software Quality Engineering*, Addison-Wesley, 1995) apresenta uma visão quantitativa da qualidade de software.

A norma de qualidade ISO 9001:2000 é discutida por Cianfani e seus colegas (*ISO 9001:2000 Explained*, second edition, American Society for Quality, 2001) e Gaal (*ISO 9001:2000 for Small Business: Implementing Process-Approach Quality Management*, St. Lucie Press, 2001). Tingley (*Comparing ISO 9000, Malcolm Baldrige, e o SEI CMM for Software*, Prentice-Hall, 1996) fornece diretrizes úteis para organizações que estão lutando para aperfeiçoar seus processos de gestão de qualidade.

Livros de George (*Lean Six Sigma*, McGraw-Hill, 2002), Pande e seus colegas (*The Six Sigma Way Fieldbook*, McGraw-Hill, 2001) e Pyzdek (*The Six Sigma Handbook*, McGraw-Hill, 2000) descrevem Seis Sigma, uma técnica estatística de gestão de qualidade que leva a produtos que têm taxas de defeitos muito baixas.

Radice (*High Quality, Low Cost Software Inspections*, Paradoxicon Publishers, 2002), Wiegers (*Peer Reviews in Software: A Practical Guide*, Addison-Wesley, 2001), Gilb e Graham (*Software Inspection*, Addison-Wesley, 1993) e Freedman e Weinberg (*Handbook of Walkthroughs, Inspections and Technical Reviews*, Dorset House, 1990) fornecem diretrizes valiosas para conduzir revisões técnicas formais eficazes.

Musa (*Software Reliability Engineering: More Reliable Software, Faster Development and Testing*, McGraw-Hill, 1998) escreveu um guia prático para técnicas aplicadas de confiabilidade de software. Antologias de importantes artigos sobre confiabilidade de software foram editadas por Kapur et al. (*Contributions to Hardware and Software Reliability Modelling*, World Scientific Publishing Co., 1999), Gritzalis (*Reliability, Quality and Safety of Software-Intensive Systems*, Kluwer Academic Publishers, 1997) e Lyu (*Handbook of Software Reliability Engineering*, McGraw-Hill, 1996).

Hermann (*Software Safety and Reliability*, Wiley-IEEE Press, 2000), Storey (*Safety-Critical Computer Systems*, Addison-Wesley, 1996) e Leveson [LEV95] continuam a ser as discussões mais abrangentes sobre segurança de software publicadas até hoje. Além disso, Van der Meulen (*Definitions for Hardware and Software Safety Engineers*, Springer-Verlag, 2000) oferece um compêndio completo de conceitos e termos importantes para confiabilidade e segurança. Gartner (*Testing Safety-Related Software*, Springer-Verlag, 1999) fornece diretrizes especializadas para testar sistemas de segurança crítica. Friedman e Voas (*Software Assessment: Reliability Safety and Testability*, Wiley, 1995) fornece modelos úteis para avaliar confiabilidade e segurança.

Uma ampla variedade de fontes de informação sobre gestão de qualidade de software está disponível na Internet. Uma lista atualizada de referências da World Wide Web pode ser encontrada no site deste livro: **http://www.mhhe.com/pressman**.

Gestão de Modificações

CAPÍTULO 27

Conceitos-Chave

auditoria 612
controle de modificação ... 609
controle de versão 610
gestão de configuração para WebE 613
gestão de conteúdo 615
identificação 607
itens de configuração de software 603
referenciais 602
modificação 600
normas 620
objetos de configuração ... 607
preparação de relatórios de estado 612
processo de gestão de configuração de software .. 606
repositório 604
sistema de versões concorrentes (CVS) 609

Modificações são inevitáveis quando o software de computador é construído. E as modificações aumentam o nível de confusão entre os engenheiros de software que estão trabalhando em um projeto. A confusão surge quando as modificações não são analisadas antes de serem feitas, não são registradas antes de serem implementadas, não são relatadas àqueles que têm necessidade de saber delas ou não são controladas para melhorar a qualidade e reduzir os erros. Babich [BAB86] discute isso quando afirma:

> A arte de coordenar desenvolvimento de software para minimizar... confusão é chamada de *gestão de configuração*, que é a arte de identificar, organizar e controlar modificações no software que está sendo construído por uma equipe de programação. O objetivo é maximizar a produtividade pela minimização dos erros.

Gestão de modificação (*Change Management*, CM), mais comumente chamada de *gestão de configuração de software* (*Software Configuration Management*, SCM), é uma atividade guarda-chuva que é aplicada ao longo de todo o processo de software. Como modificações podem ocorrer em qualquer época, as atividades de SCM são desenvolvidas para (1) identificar modificações, (2) controlar modificações, (3) garantir que as modificações sejam adequadamente implementadas e (4) relatar as modificações a outros que possam ter interesse.

É importante fazer uma distinção clara entre suporte de software e gestão de configuração de software. Suporte é um conjunto de atividades de engenharia de software que ocorre depois que o software foi entregue ao cliente e colocado em operação. Gestão de configuração de software é um conjunto de atividades de acompanhamento e controle que começam quando o projeto de engenharia de software tem início e só terminam quando o software é retirado de operação.

Um objetivo primordial da engenharia de software é melhorar a facilidade com a qual modificações podem ser acomodadas e reduzir a quantidade de esforço despendido quando elas tiverem de ser feitas. Neste capítulo, discutimos as ações específicas que nos permitem gerir modificações.

Panorama

O que é? Quando você constrói software de computador, mudanças ocorrem. E porque elas ocorrem, você precisa controlá-las efetivamente. Gestão de modificação, também chamada de gestão de configuração de software (Software Configuration Management, SCM) é um conjunto de atividades projetadas para gerir modificações, identificando os produtos de trabalho que podem ser modificados, estabelecendo relacionamentos entre eles, definindo mecanismos para administrar as diferentes versões desses produtos de trabalho, controlando as modificações impostas e fazendo auditoria, e preparando relatórios sobre as modificações efetuadas.

Quem faz? Todos os envolvidos no processo de engenharia de software estão envolvidos na gestão de modificação de algum modo, mas posições especializadas de suporte são às vezes criadas para administrar o processo SCM.

Por que é importante? Se você não controla as modificações, elas controlam você. E isso nunca é bom. É muito fácil para uma seqüência de modificações fora de controle transformar um projeto de software bem gerenciado em caos. Por esse motivo, gestão de modificação é uma parte essencial da boa gestão de projetos e da sólida prática de engenharia de software.

Quais são os passos? Como muitos produtos de trabalho são produzidos quando o software é construído, cada um deve ser identificado de modo único. Isso feito, mecanismos para controle de versão e de modificação podem ser estabelecidos. Para garantir que a qualidade seja mantida

Document shared on www.docsity.com

à medida que modificações são feitas, o processo sofre auditoria; e para assegurar que aqueles que necessitam saber sejam informados das modificações, relatos são conduzidos.

Qual é o produto do trabalho? Um Plano de Gestão de Configuração de Software define a estratégia do projeto para a gestão de modificação. Além disso, quando a SCM formal é usada, o processo de controle de modificação produz solicitações de modificação de software, relatórios e ordens de modificação de engenharia.

Como tenho certeza que fiz corretamente? Quando todo o produto de trabalho tiver sido considerado, rastreado e controlado; quando toda a modificação tiver sido rastreada e analisada; quando todos os que precisam saber de uma modificação tiverem sido informados, você fez direito.

27.1 Gestão de Configuração de Software

A saída do processo de software é informação, que pode ser dividida em três amplas categorias: (1) programas de computador (tanto na forma de código-fonte quanto executável); (2) produtos de trabalho que descrevem programas de computador (voltados tanto para profissionais técnicos quanto para usuários); e (3) dados (contidos em um programa ou externos a ele). Os itens que compreendem toda a informação produzida como parte do processo de software são chamados coletivamente de *configuração de software*.

Se cada item de configuração simplesmente levasse a outros itens, pouca confusão resultaria. Infelizmente, outra variável entra no processo — *a modificação*. A modificação pode ocorrer em qualquer época, por qualquer motivo. Na verdade, a primeira lei da engenharia de sistemas [BER80] diz: "Independentemente de onde você está no ciclo de vida do sistema, o sistema vai se modificar e o desejo de modificá-lo vai persistir ao longo de todo o ciclo de vida".

> "Não há nada permanente, exceto a mudança."
>
> Heráclito, 500 a.C.

Qual a origem dessas modificações? A resposta a essa questão é tão variada quanto as próprias modificações. No entanto, há quatro fontes fundamentais de modificação:

? Qual a origem das modificações que são solicitadas no software?

- Novas condições do negócio ou do mercado ditam modificações nos requisitos do produto ou nas regras do negócio.
- Novas necessidades do cliente exigem modificação dos dados produzidos por sistemas de informação, da funcionalidade incorporada a produtos ou dos serviços realizados por um sistema baseado em computador.
- Reorganização ou crescimento/diminuição dos negócios causa modificações nas prioridades do projeto ou na estrutura da equipe de engenharia de software.
- Restrições de orçamento ou cronograma causam redefinição do sistema ou produto.

Gestão de configuração de software é um conjunto de atividades desenvolvidas para administrar modificações ao longo do ciclo de vida do software de computador. A SCM pode ser vista como uma atividade de garantia de qualidade de software, que é aplicada ao longo de todo o processo de software. Nas seções que se seguem, examinamos as principais tarefas de SCM e conceitos importantes que nos ajudam a gerenciar as modificações.

27.1.1 Um Cenário[1] SCM

Um cenário operacional típico de CM envolve um gerente de projeto encarregado de um grupo de software, um gerente de configuração encarregado dos procedimentos e políticas de CM, os

1 Está seção é extraída de [DAR01]. Permissão especial para reproduzir "Spectrum of Functionality in CM Systems", de Susan Dart [DAR01], © 2001 da Carnegie Mellon University garantida pelo *Software Engineering Institute*.

engenheiros de software responsáveis por desenvolver e manter o produto de software e o cliente que usa o produto. No cenário, considere que é um produto de tamanho pequeno envolvendo cerca de 15 mil linhas de código sendo desenvolvido por uma equipe de seis pessoas. (Note que outros cenários de equipes menores ou maiores são possíveis, mas, em essência, existem tópicos genéricos com que cada um desses projetos deparam a respeito de CM.)

? Quais são os objetivos e atividades realizadas por cada um dos envolvidos na gestão de modificações?

No nível operacional, o cenário envolve vários papéis e tarefas. Para o gerente de projeto, o objetivo é garantir que o produto seja desenvolvido em um certo intervalo de tempo. Assim, o gerente acompanha o progresso do desenvolvimento e reconhece e reage a problemas. Isso é feito pela geração e análise de relatórios sobre o estado do sistema de software e pela realização de revisões no sistema.

Os objetivos do gerente de configuração são garantir que os procedimentos e políticas para criar, modificar e testar o código estão sendo seguidos, bem como tornar acessível a informação sobre o projeto. Para implementar técnicas para manter controle sobre modificações de código, esse gerente introduz mecanismos a fim de fazer solicitação oficial de modificações, para avaliá-las (por meio de uma Comissão de Controle de Modificação [*Change Control Board*] que é responsável pela aprovação de modificações ao sistema de software), e para autorizar as modificações. O gerente cria e distribui listas de tarefa para os engenheiros e basicamente cria o contexto do projeto. Ele também coleta estatísticas sobre componentes do sistema de software, tais como determinação de informação sobre quais componentes do sistema são problemáticos.

PONTO CHAVE

Deve haver um mecanismo para garantir que modificações simultâneas em algum componente sejam adequadamente acompanhadas, geridas e executadas.

Para os engenheiros de software, a meta é trabalhar efetivamente. Isso significa que eles não interferem desnecessariamente uns com os outros na criação e teste de código e na produção de documentos de apoio. Mas, ao mesmo tempo, eles tentam se comunicar e coordenar de modo eficiente. Especificamente, engenheiros usam ferramentas que ajudam a construir um produto de software consistente. Eles se comumicam e coordenam notificando uns aos outros sobre tarefas necessárias e tarefas completadas. Modificações são propagadas entre os trabalhos de cada um pela intercalação de arquivos. Mecanismos existem para garantir que, para componentes que passam por modificações simultâneas, existe algum modo de resolver conflitos e combinar modificações. Um histórico é mantido da evolução de todos os componentes do sistema junto com um registro das razões para as modificações e um registro de quais efetivamente foram modificados. Os engenheiros têm seus próprios espaços de trabalho para criação, modificação, teste e integração de código. Em um certo ponto, o código é transformado em um referencial a partir do qual o desenvolvimento adicional continua e a partir do qual variantes para outras máquinas-alvo são feitas.

O cliente usa o produto. Como o produto está sob controle CM, o cliente segue procedimentos formais para solicitar modificações e para indicar *bugs* no produto.

Preferencialmente, um sistema CM usado nesse cenário deve apoiar todos esses papéis e tarefas, isto é, os papéis determinam a funcionalidade exigida de sistema CM. O gerente de projeto vê CM como um mecanismo de auditoria; o gerente de configuração o vê como um mecanismo de controle, acompanhamento e construção de política; o engenheiro de software o vê como um mecanismo de controle de modificação, construção e acesso, e o cliente o vê como um mecanismo de garantia de qualidade.

27.1.2 Elementos de um Sistema de Gestão de Configuração

No seu abragente trabalho sobre gestão de configuração de software, Susan Dart [DAR01] identifica quatro importantes elementos que devem estar presentes quando um sistema de gestão de configuração é desenvolvido:

- *Elementos de componente* — um conjunto de ferramentas acoplado a um sistema de gestão de arquivos (por exemplo, um banco de dados) que possibilita acesso e gestão de cada item de configuração de software.
- *Elementos de processo* — uma coleção de procedimentos e tarefas que definem uma abordagem efetiva para a gestão de modificação (e atividades relacionadas) para todas as partes envolvidas na gestão, na engenharia e no uso de software de computador.

Document shared on www.docsity.com

- *Elementos de construção* — um conjunto de ferramentas que automatizam a construção de software garantindo que o conjunto adequado de componentes validados (isto é, a versão correta) foi montado.
- *Elementos humanos* — para implementar SCM efetiva, a equipe de software usa um conjunto de ferramentas e características de processo (inclusive outros elementos de CM).

Esses elementos (que serão discutidos em mais detalhes nas seções posteriores) não são mutuamente exclusivos. Por exemplo, elementos de componente trabalham em conjunto com elementos de construção à medida que o processo de software evolui. Elementos de processo guiam muitas atividades humanas que estão relacionadas a SCM e podem, conseqüentemente, ser considerados elementos humanos também.

27.1.3 Referenciais

AVISO

A maioria das modificações no software é justificável. Assim, não vale a pena se queixar delas. Em vez disso, certifique-se de que dispõe de mecanismos para cuidar delas.

Modificação é um fato normal no desenvolvimento de software. Clientes querem modificar requisitos. Desenvolvedores querem modificar a abordagem técnica. Gerentes querem modificar a estratégia do projeto. Por que todas essas modificações? A resposta é realmente muito simples. Com o passar do tempo todas as partes envolvidas ficam sabendo mais (sobre o que necessitam, qual a melhor abordagem, como conseguir acabar e ainda ganhar dinheiro). Esse conhecimento adicional é a força motora por trás da maioria das modificações e leva a uma declaração, que é difícil de aceitar para muitos profissionais de engenharia de software: a maioria das modificações é justificável!

Um *referencial* é um conceito de gestão de configuração de software que nos ajuda a controlar modificações sem impedir seriamente modificações justificáveis. O IEEE (IEEE Std nº 610.12-1990) define um referencial como:

> Uma especificação ou produto que foi formalmente revisto e aprovado, o qual daí em diante serve como base para o desenvolvimento futuro e que pode ser modificado apenas por meio de procedimentos formais de controle de modificação.

PONTO CHAVE

Um produto de trabalho de engenharia de software torna-se um referencial apenas depois de ter sido revisto e aprovado.

Antes que um item de configuração de software se torne referencial, qualquer modificação pode ser feita rápida e informalmente. No entanto, uma vez estabelecido o referencial, passamos figurativamente por uma porta que dá passagem em um único sentido. Modificações podem ser feitas, mas um procedimento formal específico deve ser aplicado para avaliar e verificar cada modificação.

No contexto da engenharia de software, o referencial é um marco no desenvolvimento de software. O referencial é constituído pela entrega de um ou mais itens de configuração de software que tenham sido aprovados como conseqüência de uma revisão técnica formal (Capítulo 26). Por exemplo, os elementos de um modelo de projeto foram documentados e revisados. Erros foram encontrados e corrigidos. Quando todas as partes do modelo tiverem sido revisadas, corrigidas e depois aprovadas, o modelo de projeto torna-se um referencial. Outras modificações na arquitetura do programa (documentada no modelo de projeto) podem ser feitas apenas depois que cada uma tenha sido avaliada e aprovada. Apesar de referenciais poderem ser definidos em qualquer nível de detalhe, os referenciais de software mais comuns são mostrados na Figura 27.1.

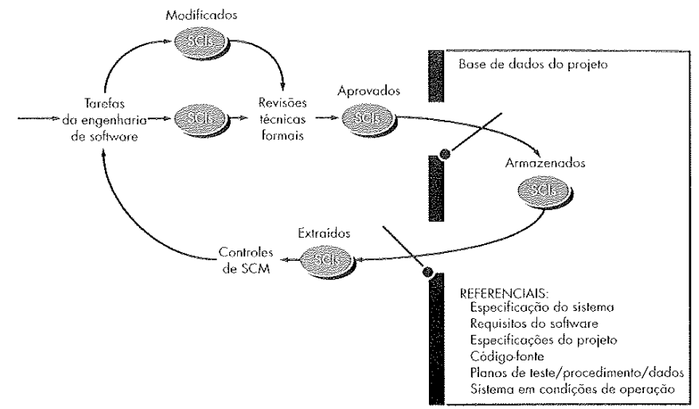
AVISO

Certifique-se de que a base de dados do projeto seja mantida numa localização centralizada e controlada.

A progressão de eventos que leva a uma linha-base também é ilustrada na Figura 27.1. As tarefas de engenharia de software produzem um ou mais SCIs. Depois de os SCIs serem revisados e aprovados, são colocados em um *banco de dados do projeto* (também chamado de *biblioteca do projeto* ou *repositório do software* discutido na Seção 27.2). Quando um membro de uma equipe de engenharia de software deseja fazer uma modificação em um SCI tornado referencial, este é copiado do banco de dados do projeto em uma área de trabalho privativa do engenheiro. No entanto, esse SCI extraído pode ser modificado somente se controles de SCM (discutidos posteriormente neste capítulo) forem seguidos. As setas da Figura 27.1 ilustram o caminho de modificação de um SCI que se tornou linha-base.

FIGURA 27.1

SCIs que se tornaram referenciais e o banco de dados do projeto



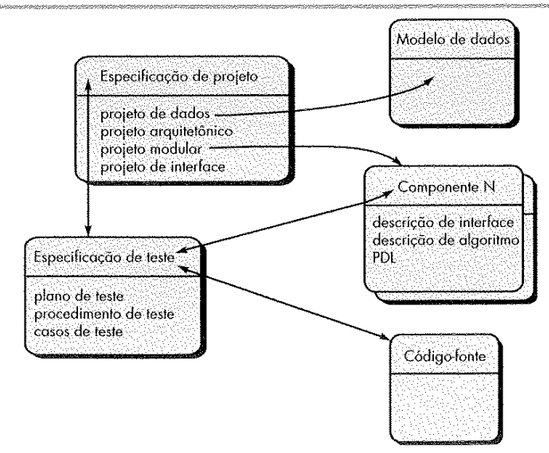
27.1.4 Itens de Configuração de Software

Um item de configuração de software é a informação criada como parte do processo de engenharia de software. Em caso extremo, pode-se considerar um SCI como sendo uma única seção de uma especificação grande ou um caso de teste em uma seqüência de testes grande. Mais realisticamente, um SCI é um documento, toda uma seqüência de casos de teste ou um componente de programa que tem nome (por exemplo, uma função C++ ou um applet Java).

Além dos SCIs que são derivados dos produtos do trabalho de software, muitas organizações de engenharia de software colocam também ferramentas de software sob controle de configuração, isto é, versões específicas de editores, compiladores, navegadores e outras ferramentas automatizadas são "congeladas" como parte da configuração de software. Como essas ferramentas foram usadas para produzir documentação, código-fonte e dados, precisam estar disponíveis quando modificações tiverem que ser feitas na configuração de software. Apesar de os problemas serem raros, é possível que uma nova versão de ferramenta (por exemplo, um compilador) possa produzir resultados diferentes que os da versão original. Por essa razão, ferramentas, como o software que elas ajudam a produzir, podem se tornar referencial como parte de um processo abrangente de gestão de configuração.

FIGURA 27.2

Objetos de configuração



Document shared on www.docsity.com

De fato, SCIs são organizados para formar objetos de configuração que podem ser catalogados no banco de dados do projeto com um único nome. Um *objeto de configuração* tem nome, atributos e é "conectado" a outros objetos por relacionamentos. Observando a Figura 27.2, os objetos de configuração EspecificaçãodeProjeto, ModelodeDados, ComponenteN, CódigoFonte e EspecificaçãodeTeste são cada um definido separadamente. No entanto, cada um desses objetos é relacionado a outros conforme mostrado pelas setas. Uma seta curva indica uma relação de composição. Isto é, ModelodeDados e ComponenteN são parte do objeto EspecificaçãodeProjeto. Uma seta retilínea de dois sentidos indica um inter-relacionamento. Se uma modificação fosse feita no objeto CódigoFonte, o inter-relacionamento permitiria a um engenheiro de software determinar que outros objetos (e SCIs) poderiam ser afetados.[2]

27.2 O Repositório SCM

No primórdios da engenharia de software, itens de configuração de software eram mantidos em documentos de papel (ou cartões perfurados!), colocados em pasta de arquivos ou pastas suspensas, e armazenados em armários de metal. Essa abordagem era problemática por várias razões: (1) encontrar um item de configuração era freqüentemente difícil; (2) determinar que itens foram modificados, quando e por quem era freqüentemente um desafio; (3) construir uma nova versão de um programa existente consumia tempo e era propenso a erros; (4) descrever relacionamentos detalhados e complexos entre itens de configuração era virtualmente impossível.

Hoje em dia, SCIs são mantidos em um banco de dados ou repositório. O Dicionário Webster define a palavra *repositório* como "qualquer coisa ou pessoa vista como um centro de acumulação ou armazenamento". Durante a história inicial da engenharia de software, o repositório era realmente uma pessoa — o programador que tinha de lembrar a localização de toda informação relevante para um projeto de software, que precisava recordar informação que nunca havia sido escrita, e reconstruir informação que fora perdida. Lamentavelmente, usando uma pessoa como "o centro de acumulação e armazenamento" (embora isso esteja de acordo com a definição de Webster) o trabalho não funciona muito bem. Atualmente, o repositório é uma "coisa" — um banco de dados que atua como o centro tanto para acumulação quanto para armazenamento de informação de engenharia de software. O papel da pessoa (o engenheiro de software) é interagir com o repositório usando ferramentas que estão integradas com ele.

27.2.1 O Papel do Repositório

O repositório SCM é o conjunto de mecanismos e estruturas de dados que permite a uma equipe de software gerir modificação de modo efetivo. Fornece as funções óbvias de um sistema de gestão de banco de dados, mas, além disso, executa ou propicia as seguintes funções [FOR89]:

? Que funções são implementadas por um repositório SCM?

- *Integridade de dados* inclui funções para validar entradas no repositório, garantir consistência entre objetos relacionados e, automaticamente, executar modificações "em cascata" quando uma modificação em um objeto exige alguma modificação em objetos a ele relacionados.
- *Compartilhamento de informação* fornece um mecanismo para compartilhar a informação entre vários desenvolvedores e entre várias ferramentas, gerencia e controla o acesso de diferentes usuários aos dados, e bloqueia ou desbloqueia objetos de modo que as modificações não sejam inadvertidamente sobrepostas umas às outras.
- *Integração de ferramenta* estabelece um modelo de dados ao qual podem ter acesso várias ferramentas de engenharia de software, controla o acesso aos dados e executa funções de gestão de configuração adequadas.

2 Esses relacionamentos são definidos no banco de dados. A estrutura do banco de dados (repositório) é discutida em mais detalhes na Seção 27.2.

- *Integração de dados* fornece funções de banco de dados que permitem que várias tarefas de SCM sejam executadas em um ou mais SCIs.
- *Imposição de metodologia* define um modelo entidade-relacionamento armazenado no repositório que implica um modelo de processo específico de engenharia de software; no mínimo, os relacionamentos entre objetos definem um conjunto de passos que precisam ser dados para construir o conteúdo do repositório.
- *Padronização de documentação* é a definição de objetos no banco de dados que leva diretamente a uma abordagem normalizada para a criação de documentos de engenharia de software.

Para realizar essas funções, o repositório é definido em termos de um metamodelo. O *metamodelo* determina como a informação é armazenada no repositório, como se pode ter acesso aos dados por ferramentas e como eles podem ser vistos pelos engenheiros de software, como a segurança e integridade dos dados pode ser bem mantida e como o modelo existente pode ser facilmente estendido para acomodar novas necessidades. Para mais informações o leitor interessado deve ver [SHA95] e [GRI95].

Veja na Web

Exemplos de repositórios comercialmente disponíveis podem ser obtidos em www.www.software.hp.com/products/SCMGR ou otn.oracle.com/documentation/repository.html.

27.2.2 Características e Conteúdo Geral

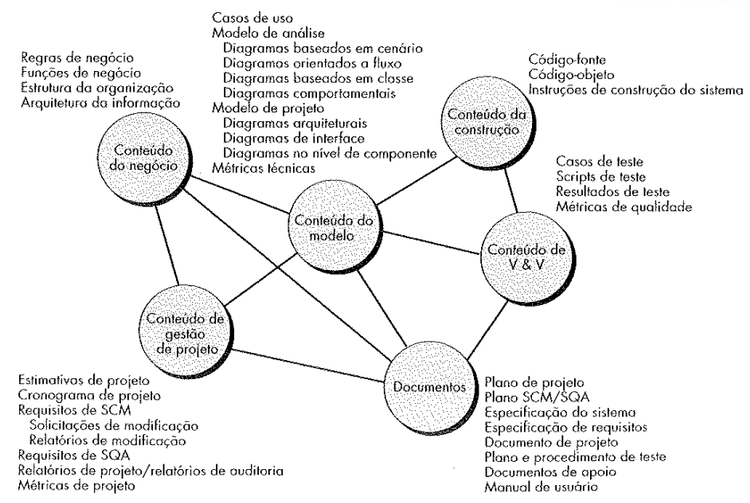
As características e o conteúdo do repositório são mais bem entendidos olhando-o de duas perspectivas: o que deve ser guardado e que serviços específicos são fornecidos pelo repositório. Uma subdivisão detalhada dos tipos de representação, documentos e produtos de trabalho que são armazenados no repositório é apresentada na Figura 27.3

Um repositório robusto fornece duas diferentes classes de serviços: (1) os mesmos tipos de serviços que poderiam ser esperados de qualquer sistema de gestão de banco de dados sofisticado e (2) serviços que são específicos do ambiente de engenharia de software.

Um repositório que serve uma equipe de engenharia de software deve (1) integrar-se com ou apoiar diretamente funções de gestão de processo; (2) apoiar regras específicas que governam a função SCM e os dados mantidos no repositório; (3) fornecer uma interface para outras ferramentas de engenharia de software; e (4) acomodar o armazenamento de objetos dados sofisticados (por exemplo, texto, gráficos, vídeo, áudio).

FIGURA 27.3

Conteúdo do repositório



Document shared on www.docsity.com

27.2.3 Características de SCM

Para apoiar SCM, o repositório precisa ter um conjunto de ferramentas que forneça apoio às seguintes características:

PONTO CHAVE

O repositório precisa ser capaz de manter SCIs relacionados a muitas versões diferentes do software. Mais importante, precisa fornecer os mecanismos para montar esses SCIs em uma configuração específica da versão.

Determinação de versão. À medida que um projeto progride, várias versões (Seção 27.3.2) de produtos de trabalho individuais são criadas. O repositório precisa ser capaz de salvar todas essas versões para permitir a gestão efetiva das entregas do produto e permitir aos desenvolvedores voltar a versões anteriores durante o teste e depuração.

O repositório deve ser capaz de controlar uma ampla variedade de tipos de objetos, inclusive texto, gráficos, mapas de bit, documentos complexos e objetos especiais como definições de tela e relatório, arquivos de objetos, dados de teste e resultados. Um repositório maduro acompanha versões de objetos em níveis de granularidade arbitrários; por exemplo, uma única definição de dado ou um agregado de módulos que podem ser acompanhados.

Acompanhamento de dependência e gestão de modificação. O repositório gera uma grande variedade de relacionamentos entre os objetos de configuração armazenados nele. Eles incluem relacionamentos entre entidade e processos da empresa, entre as partes de um projeto de aplicação, entre componentes de projeto e a arquitetura da informação da empresa, entre elementos de projeto e outros produtos de trabalho etc. Alguns desses relacionamentos são meramente associações e alguns são dependências ou relacionamentos obrigatórios.

A capacidade de acompanhar todos esses relacionamentos é crucial para a integridade da informação armazenada no repositório e para a geração de produtos de trabalho baseados nela, e é uma das mais importantes contribuições do conceito de repositório para o aperfeiçoamento do processo de desenvolvimento de software. Por exemplo, se um diagrama de classe UML é modificado, o repositório pode detectar se classes relacionadas, definições de interface e componentes de código também exigem modificação e pode trazer as SCIs afetadas para a atenção dos desenvolvedores.

Acompanhamento dos requisitos. Essa função especial fornece a capacidade de acompanhar todos os componentes e produtos passíveis de entrega de projeto e construção que resultam de uma especificação de requisitos específica (acompanhamento avante). Além disso, fornece a capacidade de identificar qual requisito gerou um produto de trabalho definido (retroacompanhamento).

Gestão de configuração. Uma facilidade de gestão de configuração acompanha uma série de configurações representando marcos de projeto ou versões de produção específicas.

Pistas de auditoria. Uma pista de auditoria estabelece informação adicional sobre quando, por que e por quem modificações foram feitas. Informação sobre a fonte das modificações pode ser introduzida como atributos de objetos específicos do repositório.

27.3 O Processo de SCM

O processo de gestão de configuração de software define uma série de tarefas que têm quatro objetivos principais: (1) identificar todos os itens que definem coletivamente a configuração de software; (2) gerir modificações em um ou mais desses itens; (3) facilitar a construção de diferentes versões de uma aplicação e (4) garantir que a qualidade do software seja mantida à medida que a configuração evolui ao longo do tempo.

Um processo que atinja esses objetivos não precisa ser burocrático e pesado, mas precisa ser caracterizado de um modo que permita a uma equipe de software desenvolver respostas para um conjunto de questões complexas:

? A quais questões o processo de SCM deve ser projetado para responder?

- Como uma equipe de software identifica os elementos discretos de uma configuração de software?
- Como uma organização gera as várias versões existentes de um programa (e sua documentação) para possibilitar que as modificações sejam acomodadas eficientemente?

- Como uma organização controla modificações antes e depois de o software ser entregue a um cliente?
- Quem tem responsabilidade pela aprovação e classificação das modificações?
- Como podemos garantir que as modificações foram feitas adequadamente?
- Qual o mecanismo usado para comunicar a terceiros as modificações feitas?

Essas questões nos levam à definição de cinco tarefas de SCM — identificação, controle de versão, controle de modificação, auditoria de configuração e preparação de relatórios — ilustrados na Figura 27.4.

Com referência à figura, as tarefas de SCM podem ser vistas como camadas concêntricas. Os SCIs fluem para fora por essas camadas durante sua vida útil, no final tornando-se parte da configuração de software de uma ou mais versões de uma aplicação ou sistema. À medida que um SCI se move por uma camada, as ações implicadas por cada camada do processo de SCM podem ou não ser aplicáveis. Por exemplo, quando um novo SCI é criado, deve ser identificado. No entanto, se nenhuma modificação é solicitada para o SCI, a camada de controle de modificação não se aplica. O SCI é atribuído a uma versão específica do software (mecanismos de controle de versão entram em ação). Um registro do SCI (seu nome, data de criação, designação de versão etc.) é mantido para finalidades de auditoria de configuração e relatado àqueles que têm necessidade de saber. Nas seções seguintes, examinamos cada uma dessas camadas de processo em mais detalhe.

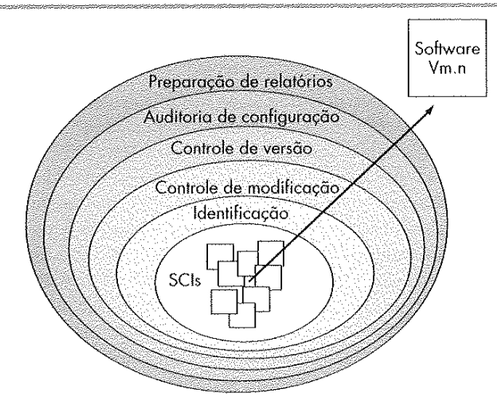
27.3.1 Identificação de Objetos na Configuração de Software

Para controlar e gerenciar itens de configuração de software, cada um deve receber um nome separadamente e depois ser organizado usando uma abordagem orientada a objetos. Dois tipos de objetos podem ser identificados [CHO89]: objetos básicos e objetos agregados.[3] Um *objeto básico* é uma unidade de informação criada por um engenheiro de software durante a análise, projeto, código ou teste. Por exemplo, um objeto básico poderia ser uma seção de uma especificação de requisitos, parte de um modelo de projeto, código-fonte de um componente ou uma seqüência de casos de testes, que é usada para exercitar o código. Um *objeto agregado* é uma coleção de objetos básicos e de outros objetos agregados. Observando a Figura 27.2, **EspecificaçãodeProjeto** é um objeto agregado. Conceitualmente, ela pode ser vista como uma lista de ponteiros que tem nome (identificada) e que especifica objetos básicos tais como **ModelodeDados** e **ComponenteN**.

Cada objeto tem um conjunto de características distintas que o identificam unicamente: um nome, uma descrição, uma lista de recursos e uma "realização". O nome do objeto é uma cadeia de

FIGURA 27.4

As camadas do processo de SCM



3 O conceito de objeto agregado [GUS89] foi proposto como um mecanismo para representar uma versão completa de uma configuração de software.

Document shared on www.docsity.com

caracteres que identifica o objeto de modo não-ambíguo. A descrição do objeto é uma lista de itens de dados que identifica: o tipo do SCI (por exemplo, documento, programa, dados) representado pelo objeto, um identificador de projeto, informação de modificação e/ou versão.

A identificação de um objeto de configuração pode também considerar os relacionamentos que existem entre objetos que receberam nome. Por exemplo, usando a notação simples

diagrama de classe <parte-de> modelo de análise;
modelo de análise < parte-de> especificação de requisitos;

criamos uma hierarquia de SCI.

Em muitos casos, objetos estão inter-relacionados entre ramos da hierarquia de objetos. Esses relacionamentos cruzados na estrutura podem ser representados da seguinte forma:

modelo de dados <inter-relacionado> modelo de fluxo de dados;
modelo de dados <inter-relacionado> caso de teste da classe m;

no primeiro caso, o inter-relacionamento é entre um objeto composto, enquanto o segundo relacionamento é entre um objeto agregado (**ModelodeDados**) e um objeto básico (**CasodeTestedaClasseM**).

O esquema de identificação para objetos de configuração precisa reconhecer que os objetos evoluem ao longo do processo de software. Antes que um objeto se torne referencial, pode ser modificado diversas vezes e, mesmo depois que o referencial for estabelecido, modificações poderão ser bastante freqüentes.

PONTO CHAVE

Os inter-relacionamentos estabelecidos para objetos de configuração permitem a um engenheiro de software avaliar o impacto de uma modificação.

AVISO

Mesmo se o banco de dados do projeto fornece a capacidade de estabelecer esses relacionamentos, eles tomam tempo para estabelecer e são difíceis de ser mantidos atualizados. Apesar de muito úteis para análise de impacto, eles não são essenciais para a gestão global de modificação.

27.3.2 Controle de Versão

Controle de versão combina procedimentos e ferramentas para gerir diferentes versões de objetos de configuração, que são criados durante o processo de software. Um sistema de controle de versão implementa ou está diretamente integrado com quatro capacidades principais: (1) um banco de dados de projeto (repositório) que guarda todos os objetos de configuração relevantes; (2) uma capacidade de *gestão de versão* que guarda todas as versões de um objeto de configuração (ou permite que qualquer versão seja construída usando diferenças de versões anteriores); (3) uma *facilidade de construir* que permite ao engenheiro de software coletar todos os objetos de configuração relevantes e construir uma versão específica do software. Além disso, sistema de controle de versão e controle de modificação freqüentemente implementam uma capacidade de *acompanhamento de tópicos* (também chamado de *acompanhamento de bugs*) que permite à equipe registrar e acompanhar o estado de todos os tópicos importantes associados a cada objeto de configuração.

PONTO CHAVE

Uma facilidade de "construir" (make) permite a um engenheiro de software extrair todos os objetos relevantes da configuração e construir uma versão específica do software.

> "Qualquer modificação, mesmo uma modificação para melhor, é acompanhada por desvantagens e desconfortos."
>
> **Arnold Bennett**

Um certo número de sistemas de controle de versão estabelece um ***conjunto de modificações*** — uma coleção de todas as modificações (em alguma configuração referencial) que são necessárias para criar uma versão específica do software. Dart [DAR91] observa que um conjunto de modificações "capta todas as modificações a todos os arquivos da configuração junto com a razão para as modificações e detalhes de quem fez as modificações e quando".

Um certo número de conjuntos de modificações que receberam denominação pode ser identificado para uma aplicação ou sistema. Isso permite a um engenheiro de software construir uma versão do software especificando os conjuntos de modificações (por nome) que precisam ser aplicados à configuração referencial. Para conseguir isso, uma abordagem de *modelagem de sistema* é aplicada. O modelo do sistema contém: (1) *um gabarito* que inclui uma hierarquia de componentes e uma "ordem de construção" para os componentes que descreve como o sistema deve ser construído, (2) regras de construção e (3) regras de verificação.[4]

4 É também possível consultar o modelo de sistema para avaliar como uma modificação em um componente terá impacto sobre outros componentes.

Um certo número de diferentes abordagens automatizadas para controle de versão foi proposto ao longo das duas últimas décadas. A diferença principal nas abordagens é a sofisticação dos atributos que são usados para construir versões e variantes específicas de um sistema e os mecanismos do processo de construção.

FERRAMENTAS DE SOFTWARE

O Sistema de Versões Concorrentes (Concurrent Versions System, CVS)

O uso de ferramentas para obter o controle de versão é essencial para gestão efetiva de modificação. *Concurrent Versions System* (CVS) é uma ferramenta amplamente usada para controle de versão. Originalmente projetada para código-fonte, mas útil para qualquer arquivo baseado em texto, o sistema CVS (1) estabelece um repositório simples, (2) mantém todas as versões de um arquivo em um único arquivo com nome, armazenando somente as diferenças entre as versões progressivas do arquivo original, e (3) protege contra modificações simultâneas de um arquivo, estabelecendo diferentes diretórios para cada desenvolvedor, isolando assim um do outro. CVS combina modificações quando cada desenvolvedor completa seu trabalho.

É importante observar que CVS não é um sistema "de construção", isto é, não constrói uma versão específica do software. Outras ferramentas (por exemplo, *Makefile*) precisam ser integradas ao CVS para conseguir isso. CVS não implementa um processo de controle de modificação (por exemplo, solicitações de modificação, relatórios de modificação, acompanhamento de *bugs*).

Mesmo com essas limitações, CVS "é um sistema predominante de código aberto para controle de versão transparente à rede [que] é útil para todos, desde desenvolvedores individuais até grandes equipes distribuídas" [CVS02]. Sua arquitetura cliente/servidor permite aos usuários ter acesso a arquivos via conexões de Internet, e sua filosofia de código aberto torna-o disponível na maioria das plataformas populares.

CVS está disponível sem custo para os ambientes Windows, Macintosh e UNIX. Veja *www.cvshome.org* para mais detalhes.

27.3.3 Controle de Modificação

A realidade do controle de modificação no contexto da engenharia de software moderna foi muito bem resumida por James Bach [BAC98]:

> O controle de modificação é vital, mas as forças que o tornam necessário também o tornam desagradável. Preocupamo-nos com a modificação porque uma pequena perturbação no código pode criar uma grande falha no produto. Mas ela também pode reparar uma falha grande ou proporcionar magníficas capacidades novas. Preocupamo-nos com as modificações porque um único desenvolvedor preguiçoso pode afundar o projeto; no entanto, idéias brilhantes originam-se nas mentes desses preguiçosos e um processo de controle de modificação complicado pode efetivamente desencorajá-los a fazer trabalho criativo.

Bach reconhece que estamos diante da necessidade de um balanceamento. Muito controle de modificação cria problemas. Pouco controle cria outros problemas.

> "A arte do progresso é preservar a ordem entre as modificações e preservar as modificações no meio da ordem."
>
> **Alfred North Whitehead**

PONTO CHAVE

Deve ser observado que um certo número de solicitações de modificação podem ser combinadas para resultar em uma única ECO e que ECOs em geral resultam em modificações a vários objetos de configuração.

Para um projeto grande de engenharia de software, modificações sem controle levam rapidamente ao caos. Para tais projetos, o controle de modificação combina procedimentos humanos e ferramentas automatizadas. O processo de controle de modificação é ilustrado esquematicamente na Figura 27.5. Um *pedido de modificação* é apresentado e avaliado para determinar o mérito técnico, os efeitos colaterais em potencial, ou o impacto geral em outros objetos da configuração e funções do sistema, e o custo projetado da modificação. Os resultados da avaliação são apresentados em um *relatório de modificação*, que é usado por uma *autoridade de controle de modificação* (*change control authority*, CCA) — uma pessoa ou grupo que toma a decisão final sobre o estado e a prioridade da modificação. Uma *ordem de modificação de engenharia* (*engineering change order* — ECO) é gerada para cada modificação aprovada. A ECO descreve a modificação a ser feita, as restrições que precisam ser respeitadas e os critérios de revisão e auditoria.

Document shared on www.docsity.com

AVISO

A confusão leva a erros — alguns dos quais muito sérios. Controle de acesso e sincronização evita confusão. Use ferramentas de controle de versão e modificação que implementem ambos.

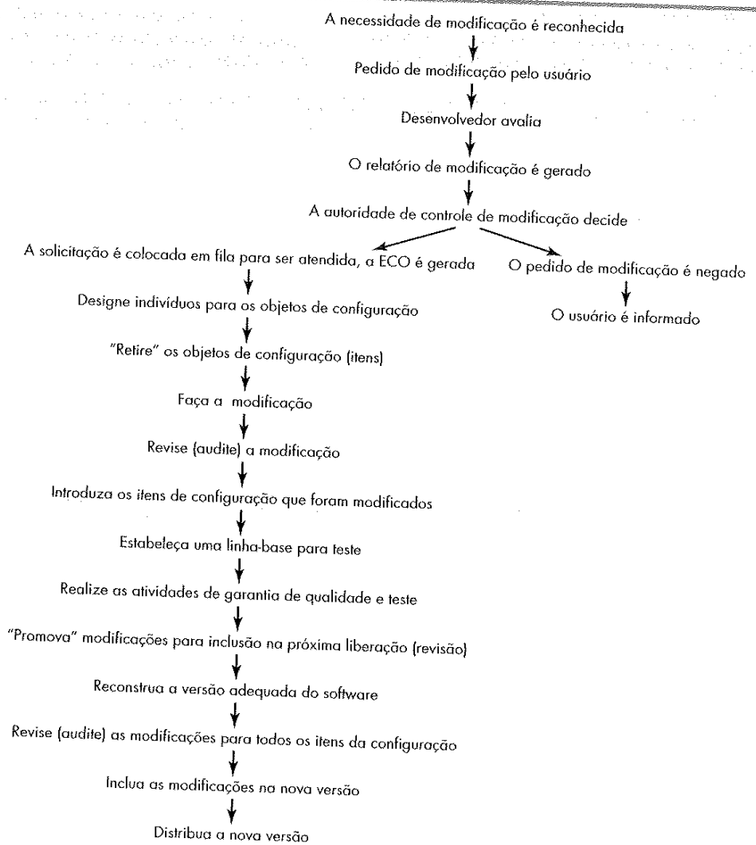
Os objetos a ser modificados podem ser colocados em um diretório que é controlado apenas pelo engenheiro de software que está fazendo a modificação. Um sistema de controle de versão (veja a moldura CVS) atualiza o arquivo original quando a modificação estiver feita. Como alternativa, o(s) objeto(s) a ser modificado(s) pode(m) ser "retirado(s)" do banco de dados do projeto (repositório), a modificação é feita e atividades adequadas de SQA são aplicadas. O(s) objeto(s) é (são) então "introduzido(s)" no banco de dados e os mecanismos adequados de controle de versão (Seção 27.3.2) são usados para criar a versão seguinte do software.

Esses mecanismos de controle de versão, integrados com o processo de controle de modificação, implementam dois elementos importantes do controle de modificação — controle de acesso e controle de sincronização. *Controle de acesso* determina quais engenheiros de software têm autoridade para ter acesso e modificar um determinado objeto de configuração. *Controle de sincronização* ajuda a garantir que modificações paralelas, realizadas por duas pessoas diferentes, não se superpõem [HAR89].

Alguns leitores podem começar a se sentir desconfortáveis com o nível de burocracia implicado na descrição do processo de controle de modificação, mostrado na Figura 27.5. Essa sensação não é incomum. Sem as salvaguardas adequadas, o controle de modificação pode retardar o progresso e criar burocracia desnecessária. A maioria dos desenvolvedores de software, que têm mecanismos de controle de modificação (infelizmente muitos não têm nenhum), tem criado algumas camadas de controle para ajudar a evitar os problemas aqui mencionados.

FIGURA 27.5

O processo de controle de modificação



AVISO

Opte por um pouco mais de controle de modificação do que você pensa que vai precisar. É provável que "demais" seja a quantidade adequada.

Antes que um SCI se torne um referencial, apenas o *controle de modificação informal* precisa ser aplicado. O desenvolvedor do objeto de configuração (SCI) em questão pode fazer quaisquer modificações justificáveis pelo projeto e pelos requisitos técnicos (desde que as modificações não afetem os requisitos mais amplos do sistema, que ficam fora do âmbito de trabalho do desenvolvedor). Quando o objeto tiver passado por revisão técnica formal e for aprovado, será criado um referencial.[5] Quando um SCI se torna referencial, o *controle de modificação para o nível de projeto* é implementado. Então, para fazer uma modificação, o desenvolvedor precisa obter aprovação do gerente de projeto (se a modificação é "local") ou da CCA, se a modificação afeta outros SCIs. Em alguns casos, a geração formal de pedidos de modificação, relatórios de modificação e ECO é dispensada. No entanto, a avaliação de cada modificação é conduzida e todas as modificações são acompanhadas e revisadas.

Quando o produto de software é entregue ao cliente, o *controle de modificação formal* é instituído. O procedimento formal de controle de modificação foi esboçado na Figura 27.5.

> "Troca é inevitável, exceto em máquinas de refrigerante."
>
> **Letreiro de pára-choque**

A autoridade de controle de modificação desempenha um papel ativo na segunda e na terceira camadas de controle. Dependendo do tamanho e caráter de um projeto de software, a CCA pode ser composta de uma pessoa — o gerente do projeto — ou de várias pessoas (por exemplo, representantes de software, hardware, engenharia de banco de dados, suporte e marketing). O papel da CCA é assumir um ponto de vista global, isto é, avaliar o impacto da modificação além do SCI em questão. Como a modificação vai afetar o hardware? Como a modificação vai afetar o desempenho? Como a modificação vai alterar a percepção que o cliente tem do produto? Como a modificação vai afetar a qualidade e a confiabilidade do produto? Essas e muitas outras questões são enfrentadas pela CCA.

CASASEGURA

Tópicos de SCM

A cena: Escritório de Doug Miller quando o projeto de software do *CasaSegura* começa.

Os personagens: Doug Miller (gerente da equipe de engenharia de software do *CasaSegura*) Vinod Raman, Jamie Lazar e outros membros da equipe de engenharia de software do produto.

A conversa:

Doug: Eu sei que é cedo, mas nós precisamos conversar sobre gestão de modificação.

Vinod (rindo): Sem dúvida. O departamento de marketing telefonou esta manhã com algumas "segundas idéias". Nada importante, mas é apenas o começo.

Jamie: Nós temos sido bastante informais quanto à gestão de modificação nos projetos anteriores.

Doug: Eu sei, mas esse é maior e mais visível, e eu lembro.

Vinod (acenando afirmativamente com a cabeça): Nós quase morremos por causa das modificações descontroladas do projeto de controle de iluminação residencial... lembra os atrasos que...

Doug (franzindo a testa): Um pesadelo que eu preferia não reviver.

Jamie: Então, o que fazemos?

Doug: Eu acho que três coisas. Primeiro, temos que desenvolver — ou tomar emprestado — um processo de controle de modificação.

Jamie: Você quer dizer, como as pessoas solicitam modificação?

Vinod: É, mas também como avaliamos a modificação, decidimos quando fazê-la (se isso é o que nós decidimos), e como manter registros do que é afetado pela modificação.

Doug: Segundo, precisamos ter uma ferramenta SCM realmente boa para controle de modificação e versão.

Jamie: Podemos construir um banco de dados para todos os nossos produtos de trabalho.

5 Um referencial pode ser criado por outras razões também. Por exemplo, quando "construções de áreas" são criadas, todos os componentes introduzidos até uma certa hora tornam-se o referencial para o trabalho do dia seguinte.

Document shared on www.docsity.com

Vinod: Eles são chamados de SCIs nesse contexto, e a maioria das boas ferramentas fornece algum apoio para isso.

Doug: Isso é um bom início, agora nós temos que...

Jamie: Ehhh, Doug, você disse que eram três coisas!

Doug (sorrindo): Terceira: temos que nos comprometer a seguir o processo de gestão de modificação e usar as ferramentas — não importa o que aconteça, certo?

27.3.4 Auditoria de Configuração

Identificação, controle de versão e controle de modificação ajudam o desenvolvedor de software a manter a ordem, o que de outro modo seria uma situação caótica e fluida. No entanto, mesmo os mecanismos de controle mais bem-sucedidos acompanham a modificação apenas até que uma ECO seja gerada. Como podemos garantir que a modificação foi adequadamente implementada? A resposta envolve dois aspectos: (1) revisões técnicas formais e (2) auditoria da configuração de software.

A revisão técnica formal (apresentada em detalhe no Capítulo 26) enfoca a correção técnica do objeto de configuração que foi modificado. Os revisores avaliam o SCI para determinar sua consistência com outros SCI, omissões ou efeitos colaterais em potencial. Uma revisão técnica formal deve ser conduzida para todas as modificações, exceto as mais triviais.

Uma *auditoria de configuração de software* complementa a revisão técnica formal, tratando das seguintes questões:

? Quais são as principais questões feitas durante uma auditoria de configuração?

1. A modificação especificada na ECO foi feita? Foi incorporada alguma modificação adicional?
2. Uma revisão técnica formal foi conduzida para avaliar a correção técnica?
3. O processo de software foi seguido e as normas de engenharia de software foram adequadamente aplicadas?
4. A modificação foi "destacada" no SCI? A data da modificação e o autor foram especificados? Os atributos do objeto de configuração refletem a modificação?
5. Os procedimentos SCM para anotar a modificação, registrá-la e relatá-la foram seguidos?
6. Todos os SCI relacionados foram adequadamente atualizados?

Em alguns casos, as questões de auditoria são formuladas como parte de uma revisão técnica formal. No entanto, quando a SCM é uma atividade formal, a auditoria SCM é conduzida separadamente pelo grupo de garantia de qualidade. Tais auditorias formais também garantem que os SCIs corretos (por versão) foram incorporados em uma construção específica em que toda a documentação está atualizada e consistente com a versão que foi construída.

AVISO

Desenvolva uma "lista de quem precisa saber" para cada objeto de configuração e a mantenha atualizada. Quando uma modificação é feita, certifique-se de que todos da lista sejam informados.

27.3.5 Preparação de Relatórios de Estado

A *preparação de relatórios de estado da configuração* (algumas vezes chamada de *contabilidade de estado*) é uma tarefa de SCM que responde às seguintes questões: (1) O que aconteceu? (2) Quem fez? (3) Quando aconteceu? (4) O que mais será afetado?

O fluxo de informação para a preparação de relatórios de estado da configuração (*configuration status reporting*, CSR) é ilustrado na Figura 27.5. Cada vez que um SCI recebe uma identificação nova ou atualizada, é feita uma entrada na CSR. Cada vez que uma modificação é aprovada pela CCA (isto é, que uma ECO é emitida) uma entrada na CSR é feita. Cada vez que uma auditoria de configuração é conduzida, os resultados são relatados como parte da tarefa de CSR. A saída de CSR pode ser colocada em um banco de dados on-line ou em um site Web, de modo que os desenvolvedores ou mantenedores de software possam ter acesso à informação de modificação por categoria de palavra-chave. Além disso, um relatório CSR é gerado regularmente e se destina a manter a gerência e os profissionais informados de modificações importantes.

FERRAMENTAS DE SOFTWARE

Apoio a SCM

Objetivo: Ferramentas SCM fornecem apoio a uma ou mais atividades do processo discutidas na Seção 27.3.

Mecânica: As mais modernas ferramentas SCM trabalham em conjunto com um repositório (um sistema de banco de dados) e fornecem mecanismos para identificação, controle de versão e modificação, auditoria e preparação de relatórios.

Ferramentas Representativas[6]

CCC/Harvest, distribuída por Computer Associates (www.cai.com), é um sistema SCM multiplataforma.

ClearCase, desenvolvida pela Rational (www.rational.com), fornece uma família de funções de SCM.

Concurrent Versions System (CVS), uma ferramenta de código aberto (www.cvshome.org), é um dos sistemas de controle de versão mais amplamente usado pela indústria (veja moldura anterior).

PVCS, distribuída por Merant (*www.merant.com*), fornece um conjunto completo de ferramentas SCM que são aplicáveis tanto para software convencional quanto para WebApps.

SourceForge, distribuída por VA Software (*sourceforge.net*), fornece gestão de versão, capacidades de construção, rastreamento tópicos/bug e muitas outras características de gestão.

SurroundSCM, desenvolvida por Seapine Software (www.seapine.com), fornece capacidade de gestão de modificação completa.

Vesta, distribuída por Compac (www.vestasys.org), é um sistema SCM de domínio público que pode apoiar tanto projetos pequenos (<10 KLOC) quanto maiores (10.000 KLOC).

Uma lista abrangente de ferramentas e ambientes SCM comerciais pode ser encontrada em *www.cmtoday.com/yp/commercial.html*.

27.4 GESTÃO DE CONFIGURAÇÃO PARA ENGENHARIA DA WEB

Na Parte 3 deste livro, discutimos a natureza especial das aplicações Web e o processo de engenharia da Web necessário para construí-las. Entre as muitas características que diferenciam WebApps de software convencional está a onipresente natureza da modificação.

Engenharia da Web usa um modelo de processo iterativo, incremental (Capítulo 16), que aplica muitos princípios derivados do desenvolvimento ágil de software (Capítulo 4). Usando essa abordagem, uma equipe de engenharia desenvolve um incremento de WebApp em um prazo muito curto por meio de uma abordagem dirigida ao cliente. Incrementos subseqüentes adicionam funcionalidade e conteúdo, e é provavel que cada um implemente modificações que levem a conteúdo aperfeiçoado, melhor usabilidade, estética aprimorada, melhor navegação, aumento de desempenho e maior segurança. Assim, no mundo ágil de engenharia da Web, modificação é vista de modo um tanto diferente.

Engenheiros da Web precisam acolher modificação e, no entanto, uma equipe tipicamente ágil evita todas as coisas que parecem tornar o processo pesado, burocrático e formal. Gestão de configuração de software é freqüentemente vista (ainda que incorretamente) como tendo essas características. Essa aparente contradição é sanada não pela rejeição dos princípios, práticas e ferramentas SCM, mas, em vez disso, pela adaptação deles para satisfazer as necessidades especiais de projetos de engenharia da Web.

27.4.1 Tópicos de Gestão de Configuração para WebApps

? Que impacto modificações não controladas têm em uma WebApp?

À medida que WebApps tornam-se crescentemente importantes para a sobrevivência e o crescimento do negócio, a necessidade de gestão de configuração cresce. Por quê? Porque sem controles efetivos, modificações impróprias em uma WebApp (relembre que imediatismo e evolução contínua são atributos dominantes de muitas WebApps) podem levar a uma colocação não autorizada de

6 As ferramentas mencionadas aqui não representam uma recomendação, mas, em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

Document shared on www.docsity.com

nova informação do produto; funcionalidade erroneamente ou pobremente testada que frusta os visitantes de um site Web; furos de segurança que colocam em perigo sistemas internos da empresa e outras conseqüências economicamente desagradáveis ou mesmo desastrosas.

As estratégias gerais para gestão de configuração de software (SCM) descritas neste capítulo são aplicáveis, mas táticas e ferramentas devem ser adaptadas para se adequarem à natureza única de WebApps. Quatro tópicos [DAR99] devem ser considerados quando se desenvolvem táticas para gestão de configuração de WebApp — conteúdo, pessoal, escalabilidade e políticas.

Conteúdo. Uma WebApp típica contém uma ampla variedade de conteúdo — texto, gráficos, applets, scripts, arquivos de áudio/vídeo, formulários, elementos ativos de página, tabelas, dados encadeados e muitos outros. O desafio é organizar esse mar de conteúdo em um conjunto racional de objetos de configuração (Seção 27.1.4) e depois estabelecer mecanismos de controle de configuração adequados para esses objetos.

Pessoal. Como uma porcentagem significante de desenvolvimento de WebApp continua a ser conduzida de modo *ad hoc*, qualquer pessoa envolvida na WebApp pode (e freqüentemente o faz) criar conteúdo. Muitos criadores de conteúdo não têm conhecimento de engenharia de software e são completamente alheios à necessidade de gestão de configuração. Como conseqüência, a aplicação cresce e muda de modo não controlado.

Escalabilidade. As técnicas e controles aplicados a pequenas WebApps não são bem escaláveis. Não é incomum que uma simples WebApp cresça significantemente à medida que interconexões com sistemas de informação, bancos de dados, armazéns de dados e portais *gateway* existentes são implementadas. À medida que tamanho e complexidade crescem, pequenas modificações podem ter efeitos indesejados de longo alcance e podem ser problemáticos. Assim, o rigor dos mecanismos de controle de configuração deve ser diretamente proporcional à escala da aplicação.

Política. Quem é o "dono" de uma WebApp? Essa questão é feita em grandes e pequenas empresas, e sua resposta tem um significante impacto nas atividades de gestão e controle associados a WebE. Em algumas instâncias, desenvolvedores Web estão alojados fora da organização de TI, criando potenciais dificuldades de comunicação. Dart [DAR99] sugere as seguintes questões para ajudar a entender a política associada à WebE:

? Como determinar quem tem responsabilidade na Gestão de Configuração da WebApp?

- Quem assume responsabilidade pela precisão da informação do site Web?
- Quem garante que os processos de controle de qualidade foram seguidos antes que a informação seja publicada pelo site?
- Quem é responsável por fazer modificações?
- Quem assume o custo de modificação?

As respostas a essas questões ajudam a determinar as pessoas dentro da organização que precisam adotar um processo de gestão de configuração para WebApps.

27.4.2 Objetos de Configuração de WebApp

WebApps englobam uma extensa faixa de objetos de configuração — objetos de conteúdo (por exemplo, texto, gráficos, imagens, vídeo, áudio), componentes funcionais (por exemplo, scripts, applets) e objetos de interface (por exemplo, COM ou CORBA). Objetos de WebApp podem ser identificados (ter nomes de arquivos a eles atribuídos) de qualquer modo que seja adequado para a organização. No entanto, as seguintes convenções são recomendadas para garantir que a compatibilidade entre plataformas seja mantida: nomes de arquivos devem ser limitados a 32 caracteres de comprimento, nomes com todas as letras maiúsculas ou misturadas devem ser evitados, e o uso de *underscores* em nomes de arquivos deve ser evitado. Além disso, referências (ligações) URL dentro de um objeto de configuração devem sempre usar caminhos relativos (por exemplo, /products/alarmsensors.html).

Todo conteúdo de WebApp tem formato e estrutura. Formatos de arquivo interno são ditados pelo ambiente computacional em que o conteúdo é armazenado. No entanto, o formato de apresentação [*rendering format*] (freqüentemente chamado de formato de exibição [*display format*]) é definido pelo estilo estético e regras de projeto estabelecidos para a WebApp. *Estrutura de conteúdo* define uma arquitetura de conteúdo, isto é, define o modo pelo qual os objetos de conteúdo são montados para apresentar informação significativa ao usuário final. Boiko [BOI02] define estrutura como "mapas que você coloca sobre um conjunto de fragmentos de conteúdo [objetos] para organizá-los e torná-los acessíveis às pessoas que precisam deles".

27.4.3 Gestão de Conteúdo

Gestão de conteúdo está relacionada à gestão de configuração no sentido de que um sistema de gestão de conteúdo (*Content Management System*, CMS) estabelece um processo (apoiado por ferramentas adequadas) que adquire o conteúdo existente (de uma ampla variedade de objetos de configuração de WebApp), estrutura-o de um modo que lhe permita ser apresentado a um usuário final e depois o fornece ao ambiente do lado do cliente para exibição.

> "Gestão de conteúdo é um antídoto à atual fúria de informação."
>
> **Bob Boiko**

PONTO CHAVE

O subsistema de coleção engloba todas as ações necessárias para criar, adquirir e/ou converter conteúdo de uma forma que possa ser apresentado no lado do cliente.

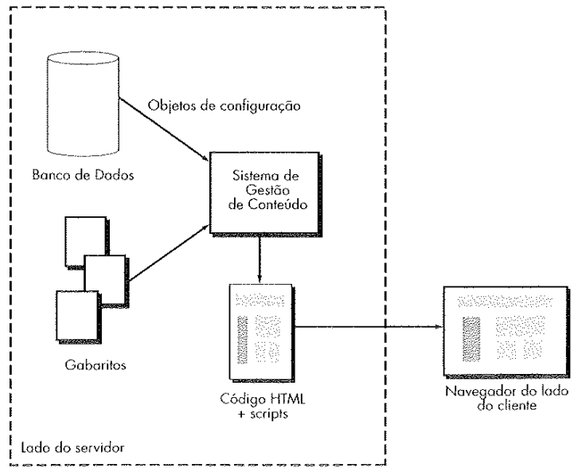
O uso mais comum de sistema de gestão de conteúdo ocorre quando uma WebApp dinâmica é construída. WebApps dinâmicas criam páginas da Web "na hora", isto é, o usuário consulta a WebApp solicitando informação específica. A WebApp consulta um banco de dados, formata a informação convenientemente, e a apresenta ao usuário. Por exemplo, uma empresa musical fornece uma biblioteca de CDs para venda. Quando o usuário solicita um CD ou sua e-música equivalente, um banco de dados é consultado e uma variedade de informação sobre o artista, o CD (por exemplo, sua ilustração de capa ou gráficos), o conteúdo musical, e amostra de áudio são todos baixados e configurados em gabarito padronizado de conteúdo. A página Web resultante é construída do lado do servidor e passada para o navegador do lado do cliente para exame pelo usuário final. Uma representação genérica disso é mostrada na Figura 27.6.

Em um sentido mais geral, um CMS "configura" o conteúdo para o usuário final pela invocação de três subsistemas: um subsistema de coleção, um subsistema de gestão e um subsistema de publicação [BOI02].

O subsistema de coleção. Conteúdo é derivado de dados e informação que precisam ser criados ou adquiridos por um desenvolvedor de conteúdo. O subsistema de coleção engloba todas as

FIGURA 27.6

Sistema de gestão de conteúdo (CMS)



Document shared on www.docsity.com

ações necessárias para criar e/ou adquirir conteúdo, e as funções técnicas que são necessárias para (1) converter conteúdo em uma forma que pode ser representada por uma linguagem de marcação (por exemplo, HTML, XML), e (2) organizar o conteúdo em pacotes que podem ser exibidos efetivamente do lado do cliente.

PONTO CHAVE

O subsistema de gestão implementa um repositório para todo o conteúdo. Gestão de configuração é realizada neste subsistema.

O subsistema de gestão. Uma vez que o conteúdo exista, ele precisa ser armazenado em um repositório, catalogado para aquisição e uso subseqüentes, e rotulado para definir (1) estado corrente (por exemplo, o objeto de conteúdo está completo ou em desenvolvimento), (2) a versão adequada do objeto de conteúdo, e (3) objetos de conteúdo relacionados. Assim, o *subsistema de gestão* implementa um repositório que engloba os seguintes elementos:

- *Banco de dados de conteúdo* — a estrutura de informação que foi estabelecida para armazenar todos os objetos de conteúdo.
- *Capacidades de banco de dados* — funções que permitem ao CMS procurar objetos de conteúdo específico (ou categorias de objetos), armazenar e recuperar objetos e gerir a estrutura de arquivo que foi estabelecida para o conteúdo.
- *Funções de gestão de configuração* — os elementos funcionais e fluxo de trabalho associado que apóiam a identificação de objeto de conteúdo, controle de versão, gestão de modificação, auditoria de modificação e preparação de relatórios.

Além desses elementos, o subsistema de gestão implementa uma função administrativa que engloba os metadados e regras que controlam a estrutura global do conteúdo e o modo pelo qual ela é apoiada.

PONTO CHAVE

O subsistema de publicação extrai conteúdo do repositório e o entrega a navegadores do lado do cliente.

O subsistema de publicação. O conteúdo precisa ser extraído do repositório, convertido para uma forma que é amena para publicação e formatado de modo que possa ser transmitido aos navegadores do lado do cliente. O *subsistema de publicação* realiza essas tarefas usando uma série de gabaritos. Cada *gabarito* é uma função que constrói uma publicação usando um de três diferentes componentes [BOI02]:

- *Elementos estáticos* — texto, gráficos, mídia e scripts que não precisam de processamento adicional são transmitidos diretamente para o lado do cliente.
- *Serviços de publicação* — chamadas de função a serviços específicos de recuperação e formatação que personalizam conteúdo (usando regras predefinidas) executam a conversão de dados e constroem ligações de navegação adequadas.
- *Serviços externos* — fornecem acesso à infra-estrutura de informação externa da empresa tais como aplicações de dados ou retaguarda da empresa.

Um sistema de gestão de conteúdo que engloba cada um desses subsistemas é aplicável a projetos importantes de engenharia da Web. No entanto, a filosofia básica e a funcionalidade associada com um CMS são aplicáveis a todas as WebApps dinâmicas.

FERRAMENTAS DE SOFTWARE

Gestão de Conteúdo

Objetivo: Apoiar engenheiros de software e desenvolvedores de conteúdo na gestão de conteúdo que é incorporada a WebApps.

Mecânica: Ferramentas dessa categoria permitem a engenheiros da Web e fornecedores de conteúdo atualizar conteúdo de WebApp de modo controlado. A maioria estabelece um simples sistema de gestão de arquivo que atribui permissões de atualização e edição página a página para vários tipos de conteúdo de WebApp. Algumas mantêm um sistema de versão de modo que versões anteriores de conteúdo possam ser obtidas para fins históricos.

Ferramentas Representativas[7]

Content Management Tools Suite, desenvolvida por interactivetools.com (*www.interactivetools.com/*), é um conjunto de ferramentas de gestão de conteúdo que enfocam a gestão de conteúdo para domínios de aplicação específicos (p. ex., artigos de notícias, anúncios classificados, imóveis).

ektron-CMS300, desenvolvida por ektron (*www.ektron.com*), é um conjunto de ferramentas que fornece capacidades de gestão de conteúdo bem como ferramentas de desenvolvimento na Web.

OmniUpdate, desenvolvida por WebsiteASP, Inc. (*www.omniupdate.com*), é uma ferramenta que permite a fornecedores de conteúdo autorizado desenvolver atualizações controladas a conteúdo de WebAPP especificados.

Tower IDM, desenvolvida por Tower Technologies (*www.towertech.com*), é um sistema de processamento de documentos e repositório de conteúdo para gerir todas as formas de informação não estruturada de negócio – imagens, formulários, relatórios gerados por computador, extratos e faturas, documentos de escritório, e-mail e conteúdo da Web.

Informação adicional sobre ferramentas de SCM e gestão de conteúdo para engenharia da Web pode ser encontrada em um ou mais dos seguintes sites Web:

Developer's Virtual Encyclopedia (*www.wdlv.com*), *WebDeveloper* (*www.webdeveloper.com*),

Developer Shed (*www.devshed.com*), *webknowhow.net* (*www.webknowhow.net*) ou

WebReference (*www.webreference.com*).

27.4.4 Gestão de Modificação

O fluxo de trabalho associado com o controle de modificação para software convencional (Seção 27.3.3) é geralmente muito pesado para engenharia da Web. É improvável que a seqüência solicitação de modificação, relatório de modificação e ordem de engenharia de modificação possa ser conseguida de um modo ágil que seja aceitável pela maioria dos projetos de desenvolvimento de WebApp. Como então gerimos um fluxo contínuo de modificações solicitadas para conteúdo e funcionalidade de WebApp?

Para implementar gestão de modificação efetiva dentro da filosofia de "codifique e vá em frente" que continua a dominar o desenvolvimento de WebApp, o processo de controle de modificação convencional deve ser modificado. Cada modificação deve ser categorizada em uma de quatro classes:

Classe 1 — uma modificação de conteúdo ou função que corrige um erro ou aperfeiçoa conteúdo ou funcionalidade local.

Classe 2 — uma modificação de conteúdo ou função que tem impacto em outros objetos de conteúdo ou componentes funcionais.

Classe 3 — uma modificação de conteúdo ou função que tem impacto espalhado por uma WebApp (por exemplo, extensões importantes de funcionalidade, aperfeiçoamento significativo ou redução de conteúdo, importantes modificações solicitadas na navegação).

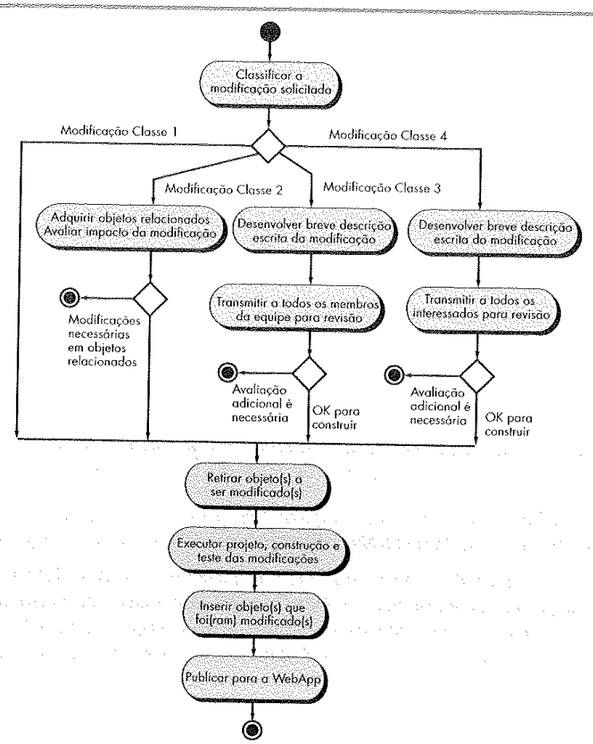
Classe 4 — uma importante modificação de projeto (por exemplo, uma modificação em projeto de interface ou abordagem de navegação) que será imediatamente notada por uma ou mais categorias de usuário.

7 As ferramentas mencionadas aqui não representam uma recomendação, mas, em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

Document shared on www.docsity.com

FIGURA 27.7

Gestão de modificações para WebApps



Uma vez categorizada a solicitação de modificação, ela pode ser processada de acordo com o algoritmo mostrado na Figura 27.7.

Com referência à figura, modificações classes 1 e 2 são tratadas informalmente e cuidadas de modo ágil. Para uma modificação classe 1, o engenheiro da Web avalia o impacto da modificação, mas nenhuma revisão externa ou documentação é necessária. À medida que a modificação é feita, os procedimentos-padrão de retirada e inserção são impostos pelas ferramentas do repositório de configuração. Para modificações classe 2, é incubência do engenheiro da Web revisar o impacto da modificação nos objetos relacionados (ou pedir a outros desenvolvedores responsáveis por aqueles objetos para fazê-lo). Se pode ser feita sem exigir modificações significativas em outros objetos, ela ocorre sem revisão ou documentação adicional. Se substanciais modificações são necessárias, avaliação e planejamento adicionais também são necessários.

Modificações classes 3 e 4 são também tratadas de modo ágil, mas alguma documentação descritiva e procedimentos de revisão mais formais são necessários. Uma *descrição de modificação* — descrevendo a modificação e fornecendo uma breve avaliação do impacto da modificação — é desenvolvida para modificações classe 3. A descrição é distribuída a todos os membros da equipe de engenharia da Web que a revisam para melhor avaliar seu impacto. Uma descrição de modificação é também desenvolvida para modificações classe 4, mas nesse caso a revisão é conduzida por todos os interessados.

FERRAMENTAS DE SOFTWARE

Gestão de Modificação

Objetivo: Apoiar engenheiros da Web e desenvolvedores de conteúdo na gestão de modificações à medida que elas são feitas em objetos de configuração de WebApp.

Mecânica: Ferramentas dessa categoria foram originalmente desenvolvidas para software convencional, mas podem ser adaptadas por engenheiros da Web para fazer modificações controladas em WebApps.

Ferramentas Representativas[8]

ChangeMan WCM, desenvolvida por Serena (*www.serena.com*), é uma de um conjunto de ferramentas de gestão de modificação que fornece capacidades de SCM.

ClearCase, desenvolvida por Rational (*www.rational.com*), é um conjunto de ferramentas que fornece capacidades de gestão de configuração completas para WebApps.

PVCS, desenvolvida por Merant (*www.merant.com*), é um conjunto de ferramentas que fornece capacidades de gestão de configuração completas para WebApps.

Source Integrity, desenvolvida por mks (*www.mks.com*), é uma ferramenta de SCM que pode ser integrada a ambientes de desenvolvimento selecionados.

27.4.5 Controle de Versão

À medida que uma WebApp evolui por meio de uma série de incrementos, um certo número de versões diferentes pode existir ao mesmo tempo. Uma versão (a WebApp operacional corrente) está disponível via Internet a usuários finais; outra versão (o incremento seguinte da WebApp) pode estar nos estágios finais de teste antes da implantação; uma terceira versão está em desenvolvimento e representa uma atualização importante em conteúdo, estética da interface e funcionalidade.

Objetos de configuração precisam ser claramente definidos de modo que cada um possa ser associado à versão adequada. Além disso, mecanismos de controle devem ser estabelecidos. Dreilinger [DRE99] discute a importância de controle de versão (e modificação) quando escreve:

> Em site *não controlado*, em que vários autores têm acesso para editar e contribuir, o potencial para conflitos e problemas aumenta — mais ainda quando esses autores trabalham em diferentes escritórios, em diferentes horas do dia e da noite. Você pode passar o dia melhorando o arquivo *index.html* para um cliente. Depois que fez suas modificações, outro desenvolvedor que trabalha em casa depois do expediente ou em outro escritório pode passar a noite carregando sua própria nova versão revisada do arquivo *index.html*, escrevendo completamente por cima do seu trabalho sem maneira de recuperá-lo.

Essa situação deve parecer familiar a todo engenheiro de software bem como a todo engenheiro da Web. Para evitá-lo, um processo de controle de versão deve ser estabelecido.

1. *Um repositório central para o projeto de WebApp deve ser estabelecido.* O repositório conterá a versão corrente de todos os objetos de configuração da WebApp (conteúdo, componentes funcionais e outros).
2. *Cada engenheiro da Web cria sua própria pasta de trabalho.* A pasta contém os objetos que estão sendo criados ou modificados em um determinado momento.
3. *Os relógios em todas as estações de trabalho de desenvolvedores devem ser sincronizados.* Isso é feito para evitar conflitos de sobrescrita quando dois desenvolvedores fazem atualizações que são muito próximas umas das outras em tempo.
4. *À medida que novos objetos de configuração são desenvolvidos ou objetos existentes são modificados, eles são importados para o repositório central.* A ferramenta de controle de versão (veja anteriormente neste capítulo discussão de CVS) vai gerir todas as funções de inserção e retirada das pastas de trabalho de cada engenheiro da Web.
5. *À medida que objetos são importados ou exportados pelo repositório, uma mensagem de registro com hora registrada é feita.* Isso fornece informação útil para auditoria e pode tornar-se parte de um esquema efetivo de relatório.

8 As ferramentas mencionadas aqui não representam uma recomendação, mas, em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

Document shared on www.docsity.com

A ferramenta de controle de versão mantém diferentes versões da WebApp e pode reverter a uma versão mais antiga se necessário.

27.4.6 Auditoria e Preparação de Relatório

No interesse da agilidade, as funções de auditoria e preparação de relatórios são desenfatizadas no trabalho de engenharia da Web. No entanto, elas não são totalmente eliminadas. Todos os objetos inseridos ou retirados do repositório são gravados em um registro que pode ser revisto a qualquer momento. Um relatório de registro completo pode ser criado de modo que todos os membros da equipe de engenharia da Web tenham uma cronologia das modificações durante um período definido de tempo. Além disso, uma notificação automatizada por e-mail (endereçada a todos os desenvolvedores e interessados) pode ser enviada toda vez que um objeto é inserido ou retirado do repositório.

INFO

Normas SCM

A seguinte lista de normas SCM (extraída em parte de www.12207.com) é razoavelmente abrangente:

IEEE Standards	**standards.ieee.org/catalog/olis/**
IEEE 828	Software Configuration Management Plans
IEEE 1042	Software Configuration Management
ISO Standards	**www.iso.ch/iso/en/ISOOnline.frontpage**
ISO 10007-1995	Quality Management, Guidance for CM
ISO/IEC 12207	Information Technology — Software Life Cycle Processes
ISO/IEC TR 15271	Guide for ISO/IEC 12207
ISO/IEC TR 15846	Software Engineering — Software Life Cycle Process — Configuration Management for Software
EIA Standards	**www.eia.org/**
EIA 649	National Consensus Standard for Configuration Management
EIA CMB4-1	A Configuration Management Definitions for Digital Computer Programs
EIA CMB4-2	Configuration Identification for Digital Computer Programs
EIA CMB4-3	Computer Software Libraries
EIA CMB4-4	Configuration Change Control for Digital Computer Programs
EIA CMB6-1C	Configuration and Data Management References
EIA CMB6-3	Configuration Identification
EIA CMB6-4	Configuration Control
EIA CMB6-5	Textbook for Configuration Status Accounting
EIA CMB7-1	Electronic Interchange of Configuration Management Data
U.S. Military	**www-library.itsi.disa.mil tandards**
DoD MIL STD-973	Configuration Management
MIL-HDBK-61	Configuration Management Guidance
Other standards	
DO-178B	Guidelines for the Development of Aviation Software
ESA PSS-05-09	Guide to Software Configuration Management
AECL CE-1001-STD	Standard for Software Engineering rev.1 of Safety Critical Software
DOE SCM checklist	cio.doe.gov/ITReform/sqse/download/cmcklst.doc
BS-6488 British Std.,	Configuration Management of Computer-Based Systems
Best Practice –UK	Office of Government Commerce: www.ogc.gov.uk
CMII	Institute of CM Best Practices: www.icmhq.com

Um Guia de Recursos de Gestão de Configuração fornece informação complementar para os interessados em processos e prática de gestão de configuração. Ele está disponível em www.quality.org/config/cm-guide.html.

27.5 Resumo

Gestão de configuração de software é uma atividade guarda-chuva é aplicada ao longo de todo o processo de software. A SCM identifica, controla, audita e relata modificações que invariavelmente ocorrem enquanto o software está sendo desenvolvido e depois de ele ter sido entregue ao cliente. Toda informação produzida como parte da engenharia de software torna-se parte de uma configuração de software. A configuração é organizada de modo que permita controle ordenado de modificação.

A configuração de software é composta de um conjunto de objetos inter-relacionados, também chamados de itens de configuração de software, que são produzidos como resultado de alguma atividade de engenharia de software. Além dos documentos, programas e dados, o ambiente de desenvolvimento usado para criar software pode também ser colocado sob o controle de configuração. Todos os SCIs são armazenados em um repositório que implementa mecanismos e estruturas de dados para garantir integridade dos dados, fornece apoio de integração com outras ferramentas de software, apóia o compartilhamento da informação entre todos os membros da equipe de software e implementa funções de apoio de controle de versão e modificação.

Quando um objeto de configuração tiver sido desenvolvido e revisado, tornar-se-á um referencial. Modificações em objetos que se tornaram referenciais resultam na criação de uma nova versão daquele objeto. A evolução de um programa pode ser rastreada examinando o histórico de revisão de todos os objetos de configuração. Objetos básicos e compostos formam um repositório de objetos a partir do qual são criadas versões. O controle de versão é um conjunto de procedimentos e ferramentas para administrar o uso desses objetos.

O controle de modificação é uma atividade procedimental que garante qualidade e consistência à medida que modificações são feitas em um objeto de configuração. O processo de controle de modificação começa com um pedido de modificação, leva a uma decisão de aceitar ou rejeitar o pedido de modificação e culmina com uma atualização controlada do SCI que deve ser modificado.

A auditoria de configuração é uma atividade de SQA que ajuda a garantir que a qualidade seja mantida à medida que as modificações são feitas. A preparação de relatórios de estado fornece informação sobre cada modificação para aqueles que precisam saber.

Gestão de configuração para engenharia da Web é semelhante em muitos pontos a SCM para software convencional. No entanto, cada uma das tarefas centrais de SCM deve ser encadeada para torná-la tão simples quanto possível e provisões especiais para gestão de conteúdo devem ser implementadas.

Referências Bibliográficas

[BAB86] Babich, W. A., *Software Configuration Management*, Addison-Wesley, 1986.

[BAC98] Bach, J., "The Highs and Lows of Change Control", *Computer*, v. 31, n. 8, ago. 1998, p. 113-115.

[BER80] Bersoff, E. H., Henderson, V. D. e Siegel, S. G. *Software Configuration Management*, Prentice-Hall, 1980.

[BOI02] Boiko, B., *Content Management Bible*, Hungry Minds Publishing, 2002.

[CHO89] Choi, S. C. e Scacchi, W., "Assuring the Correctness of a Configured Software Description", *Proc. 2nd Intl. Workshop on Software Configuration Management*, ACM, Princeton, NJ, out. 1989, p. 66-75.

[CVS02] Concurrent Versions System Web site, www.cvshome.org, 2002.

[DAR91] Dart, S., "Concepts in Configuration Management Systems", *Proc. Third International Workshop on Software Configuration Management*, ACM SIGSOFT, 1991, disponível em http://www.sei.cmu.edu/legacy/scm/abstracts/abscm_concepts.html.

[DAR99] "Change Management: Containing the Web Crisis", *Proc. Software Configuration Management Symposium*, Toulouse, France, 1999, disponível em http://www.perforce.com/perforce/conf99/dart.html.

[DAR01] Dart, S., *Spectrum of Functionality in Configuration Management Systems*, Software Engineering Institute, 2001, disponível em http://www.sei.cmu.edu/legacy/scm/tech_rep/TR11_90/TOC_TR11_90.html.

[DRE99] Dreilinger, S., "CVS Version Control for Web Site Projects", 1999, disponível em http://www.durak.org/cvswebsites/howto-cvs/howto-cvs.html.

Document shared on www.docsity.com

[FOR89] Forte, G., "Rally Round the Repository", *CASE Outlook*, dez. 1989, p. 5-27.

[GRI95] Griffen, J., "Repositories: Data Dictionary Descendant Can Extend Legacy Code Investment", *Application Development Trends*, abr. 1995, p. 65-71.

[GUS89] Gustavsson, A., "Maintaining the Evolution of Software Objects in an Integrated Environment", *Proc. 2nd Intl. Workshop on Software Configuration Management*, ACM, Princeton, NJ, out. 1989, p. 114-117.

[HAR89] Harter, R., "Configuration Management", *HP Professional*, v. 3, n. 6, jun. 1989.

[IEE94] *Software Engineering Standards*, IEEE Computer Society, 1994.

[JAC02] Jacobson, I., "A Resounding 'Yes' to Agile Processes — But Also More", *Cutter IT Journal*, v. 15, n. 1., jan. 2002, p. 18-24.

[REI89] Reichenberger, C., "Orthogonal Version Management", *Proc. 2nd Intl. Workshop on Software Configuration Management*, ACM, Princeton, NJ, out. 1989, p. 137-140.

[SHA95] Sharon, D. e Bell, R., "Tools That Bind: Creating Integrated Environments", *IEEE Software*, mar. 1995, p. 76-85.

[TAY85] Taylor, B., "A Database Approach to Configuration Management for Large Projects", *Proc. Conf. Software Maintenance — 1985*, IEEE, nov. 1985, p. 15-23.

Problemas e Pontos a Considerar

27.1. Por que a Primeira Lei da Engenharia de Software é verdadeira? Forneça exemplos específicos de cada uma das quatro razões fundamentais para modificação.

27.2. Quais são os quatro elementos existentes quando um sistema efetivo de SCM é implementado? Discuta brevemente cada um.

27.3. Discuta as razões para referenciais com suas próprias palavras.

27.4. Considere que você é o gerente de um pequeno projeto. Que referenciais você definiria para o projeto e como iria controlá-los?

27.5. Use agregações ou compostos UML (Capítulo 8) para descrever os inter-relacionamentos entre os SCIs (objetos de configuração listados na Seção 27.1.4).

27.6. Projete um sistema de projeto (repositório) que permita a um engenheiro de software armazenar, estabelecer referências cruzadas, rastrear, atualizar e modificar todos os itens de configuração de software importantes. Como esse banco de dados manipularia versões diferentes do mesmo programa? O código-fonte seria tratado diferentemente da documentação? Como dois desenvolvedores seriam impedidos de fazer modificações diferentes no mesmo SCI, ao mesmo tempo?

27.7. Pesquise uma ferramenta de SCM existente e descreva como ela implementa controle de versões, e objetos de configuração em geral.

27.8. As relações <parte-de> e < inter-relacionado> representam relacionamentos simples entre objetos de configuração. Descreva cinco outros relacionamentos que poderiam ser úteis no contexto de um repositório SCM.

27.9. Pesquise uma ferramenta existente de SCM e descreva como ela implementa a mecânica de controle de versão. Alternativamente, leia dois ou três dos trabalhos sobre SCM e descreva as diferentes estruturas de dados e mecanismos de referência que são usados para controle de versão.

27.10. Usando a Figura 27.5 como guia, desenvolva uma subdivisão de trabalho ainda mais detalhada para controle de modificação. Descreva o papel da CCA e sugira formatos para o pedido de modificação, o relatório de modificação e o ECO.

27.11. Desenvolva uma checklist para uso durante auditorias de configuração.

27.12. Qual a diferença entre uma auditoria de SCM e uma revisão técnica formal? As suas funções podem ser unidas em uma mesma revisão? Quais são os prós e os contras?

27.13. Descreva rapidamente as diferenças entre SCM para software convencional e SCM para WebApps.

27.14. O que é gestão de conteúdo? Use a Web para pesquisar as características de uma ferramenta de gestão de conteúdo e forneça um breve resumo.

Leituras e Fontes de Informação Adicionais

Lyon (*Practical CM*, Raven Publishing, 2003, disponível em www.configuration.org) escreveu um guia abrangente para profissionais de GC (*Configuration Management*, CM) que inclui diretrizes pragmáticas para implementação de todos os aspectos de um sistema de gestão de configuração (atualizado anualmente). Hass (*Configuration Management: Principles and Practice*, Addison-Wesley, 2002) e Leon (*A Guide to Software Configuration Management*, Artech House, 2000) fornecem um conjunto útil sobre o assunto. White e Clemm (*Software Configuration Management Strategies and Rational ClearCase*, Addison-Wesley, 2000) apresentam SCM (*Software Configuration Management*) no contexto de uma das mais populares ferramentas de SCM.

Mikkelsen e Pherigo (*Practical Software Configuration Management: The Latenight Developer's Handbook*, Allyn & Bacon, 1997) e Compton e Callahan (*Configuration Management for Software*, VanNostrand-Reinhold, 1994) fornecem tutorais pragmáticos sobre importantes práticas de SCM. Ben-Menachem (*Software Configuration Management Guidebook*, McGraw-Hill, 1994) e Ayer e Patrinnostro (*Software Configuration Management*, McGraw-Hill, 1992) apresentam uma boa visão geral para aqueles que precisam de introdução adicional sobre o assunto. Berlack (*Software Configuration Management*, Wiley, 1992) apresenta um levantamento útil de conceitos SCM, enfatizando a importância de repositório e ferramentas de gestão de modificação. Babich [BAB86] fornece uma abreviada, mas importante abordagem de tópicos pragmáticos de gestão de configuração de software. Arnold e Bohner (*Software Change Impact Analysis*, IEEE Computer Society Press, 1996) editaram uma antologia que discute como analisar o impacto de modificações em sistemas complexos baseados em software.

Berczuk e Appleton (*Software Configuration Management Patterns*, Addison-Wesley, 2002) apresentam uma variedade de padrões úteis para apoiar o entendimento de SCM e implementar sistemas SCM eficazes. Brown *et al.* (*Anti-Patterns and Patterns in Software Configuration Management*, Wiley, 1999) discutem as coisas a não fazer (antipadrões) ao implementar um processo de SCM e depois consideram como remediá-las.

Buckley (*Implementing Configuration Management*, IEEE Computer Society Press, 1993) considera abordagens de gestão de configuração para todos os elementos do sistema — hardware, software e firmware — com discussão detalhada das principais atividades de CM. Rawlings (*SCM for Network Development Environments*, McGraw-Hill, 1994) considera o impacto de SCM no desenvolvimento de software em um ambiente de redes. Bays (*Software Release Methodology*, Prentice-Hall, 1999) apresenta uma coleção das melhores práticas para todas as atividades que ocorrem depois que modificações são feitas em uma aplicação.

À medida que WebApps ficaram mais dinâmicas, gestão de conteúdo tornou-se um tópico essencial para engenheiros da Web. Livros de Addey e seus colegas (*Content Management Systems*, Glasshaus, 2003), Boiko [BOI02], Hackos (*Content Management for Dynamic Web Delivery*, Wiley, 2002), Nakano (*Web Content Management*, Addison-Wesley, 2001) apresentam abordagens valiosas para o assunto.

Uma ampla variedade de fontes de informação sobre gestão de configuração de software está disponível na Internet. Uma lista atualizada de referências da World Wide Web pode ser encontrada no site deste livro: **http://www.mhhe.com/pressman**.

Document shared on www.docsity.com

PARTE

TÓPICOS AVANÇADOS EM ENGENHARIA DE SOFTWARE

Nesta parte de *Engenharia de Software* vamos considerar um certo número de tópicos avançados que vão ampliar seu conhecimento sobre a engenharia de software. As questões abaixo são tratadas nos capítulos seguintes:

- Que notação e preliminares matemáticas ("métodos formais") são necessárias para especificar formalmente software?
- Que atividades técnicas chave são conduzidas durante o processo de engenharia de software sala limpa ?
- Como a engenharia de software baseada em componente é usada para criar sistemas a partir de componentes reusáveis?
- Que atividades técnicas são necessárias para reengenharia de software?
- Quais são as direções futuras da engenharia de software?

Uma vez respondidas essas questões, você compreenderá tópicos que podem ter um profundo impacto em engenharia de software na próxima década.

Document shared on www.docsity.com

CAPÍTULO

28 MÉTODOS FORMAIS

CONCEITOS-CHAVE

especificação construtiva .. 631
especificação formal 635
esquemas 640
estados 629
invariante de dados 629
linguagem Z 640
OCL 637
operações 629
operadores de conjuntos .. 632
operadores lógicos 634
pré e pós-condições 629

Os métodos de engenharia de software podem ser categorizados em um espectro de "formalidade" que está frouxamente ligado ao grau de rigor matemático aplicado durante a análise e o projeto. Por essa razão, os métodos de análise e projeto discutidos anteriormente neste livro situam-se no extremo informal do espectro. Uma combinação de diagramas, texto, tabelas e notação simples é usada para criar modelos de análise e projeto, mas pouco rigor matemático foi aplicado.

Agora vamos considerar o outro extremo do espectro da formalidade. Aqui, uma especificação e projeto são descritos usando uma sintaxe e uma semântica formais, que especificam a função e o comportamento do sistema. A especificação é matemática na forma (por exemplo, cálculos de proposições podem ser usados como base para uma linguagem de especificação formal).

Em sua discussão introdutória aos métodos formais, Anthony Hall [HAL90] afirma:

> Os métodos formais são controversos. Seus advogados alegam que eles podem revolucionar o desenvolvimento [de software]. Seus opositores pensam que eles são impossivelmente difíceis. Enquanto isso, para a maioria das pessoas, os métodos formais são tão pouco familiares que é difícil julgar as alegações competitivas.

Neste capítulo, exploramos métodos formais e examinamos seu possível impacto na engenharia de software nos próximos anos.

PANORAMA

O que é? Os métodos formais permitem a um engenheiro de software criar uma especificação mais completa, consistente e não-ambígua do que aquelas produzidas por métodos convencionais. A teoria dos conjuntos e a notação lógica são usadas para criar um enunciado claro dos fatos (requisitos). Essa especificação matemática pode então ser analisada para melhorar (ou mesmo provar) correção e consistência. Como a especificação é criada usando uma notação matemática, ela é inerentemente menos ambígua que os métodos informais de representação.

Quem faz? Uma especificação formal é criada por um engenheiro de software especialmente treinado.

Por que é importante? Falhas podem ter um alto preço em sistemas de segurança crítica ou de missão crítica. Vidas podem ser perdidas ou severas conseqüências econômicas podem surgir quando falha o software de computador. Em tais situações, é essencial que os erros sejam descobertos antes que o software seja colocado em operação. Os métodos formais reduzem drasticamente os erros de especificação e, como conseqüência, servem de base para um software com poucos erros, quando o cliente começa a usá-lo.

Quais são os passos? A notação e heurística de conjuntos e especificação construtiva — operadores de conjunto, operadores lógicos e seqüências — formam a base de métodos formais. Métodos formais definem o invariante de dados, estado e operações de uma função do sistema pela tradução dos requisitos informais do problema para uma representação mais formal.

Qual é o produto do trabalho? Uma especificação, representada em uma linguagem formal tal como OCL ou Z, é produzida quando métodos formais são aplicados.

Como tenho certeza de que fiz corretamente? Como os métodos formais usam matemática discreta como mecanismo de especificação, provas lógicas podem ser aplicadas a cada função do sistema para demonstrar que a especificação está correta. No entanto, mesmo se provas lógicas não são usadas, a estrutura e disciplina de uma especificação formal vai melhorar a qualidade do software.

28.1 CONCEITOS BÁSICOS

A *Enciclopédia de Engenharia de Software* [MAR94] define métodos formais do seguinte modo:

> O método é formal se tem uma sólida base matemática, dada tipicamente por uma linguagem formal de especificação. Essa base fornece um meio para definir precisamente noções como consistência e completeza, e mais relevantemente, especificação, implementação e correção.

As propriedades desejáveis de uma especificação formal — consistência, completeza e falta de ambigüidade — são os objetivos de todos os métodos de especificação. No entanto, o uso de métodos formais resulta em uma probabilidade muito mais alta para atingir esses ideais. A sintaxe formal de uma linguagem de especificação (Seção 28.4) permite que os requisitos e projetos sejam interpretados apenas de um modo, eliminando a ambigüidade, que freqüentemente ocorre quando uma linguagem natural (por exemplo, inglês) ou uma notação gráfica precisam ser interpretadas por um leitor. As facilidades descritivas da teoria dos conjuntos e da notação lógica (Seção 28.2) permitem o claro enunciado de fatos (requisitos). Para ser consistente, fatos declarados em uma parte da especificação não devem ser contraditos em outra parte. A consistência é garantida pela prova matemática de que os fatos iniciais podem ser formalmente aplicados (usando regras de inferência) nas últimas declarações da especificação.

> "Os métodos formais têm um potencial fremendo para melhorar a clareza e a precisão da especificação dos requisitos e de encontrar erros importantes e sutis."
>
> **Steve Easterbrook *et al.***

A completeza é difícil de atingir, mesmo quando são usados métodos formais. Alguns pontos de um sistema podem ser deixados indefinidos à medida que a especificação está sendo criada; outras características podem ser omitidas propositadamente para dar aos projetistas alguma liberdade na escolha de uma abordagem de implementação; e, finalmente, é impossível considerar todo o cenário operacional em um sistema grande e complexo. As coisas podem simplesmente ser omitidas por erro.

Apesar de o formalismo fornecido pela matemática ter algum apelo para alguns engenheiros de software, outros (alguns diriam, a maioria) olham de esguelha para uma visão matemática de desenvolvimento de software. Para entender por que uma abordagem formal tem mérito, precisamos primeiro considerar as deficiências associadas com as abordagens menos formais.

28.1.1 Deficiências de Abordagens Menos Formais[1]

Os métodos discutidos para análise e projeto nas Partes 2 e 3 deste livro fizeram forte uso de linguagem natural e de uma variedade de notações gráficas. Apesar de a aplicação cuidadosa de métodos de análise e projeto, acoplados a rigorosas revisões, poder e efetivamente conduzir a software de alta qualidade, um descuido na aplicação desses métodos pode criar uma variedade de problemas. Uma especificação de sistema pode conter contradições, ambigüidades, pontos vagos, declarações incompletas e níveis de abstração misturados.

Contradições são conjuntos de afirmações em desacordo uns com os outros. Por exemplo, uma parte da especificação do sistema pode declarar que o sistema deve monitorar todas as temperaturas em um reator químico, enquanto a outra parte, talvez escrita por outra pessoa, pode declarar que apenas temperaturas dentro de um certo intervalo devem ser monitoradas.

Ambigüidades são declarações que podem ser interpretadas de diferentes maneiras. Por exemplo, a seguinte declaração é ambígua:

> A identidade do operador consiste no nome do operador e na senha; a senha consiste de seis dígitos. Deve ser mostrada no monitor de vídeo de segurança e depositada no arquivo de registro de entradas, quando um operador entra no sistema.

AVISO

Apesar de um bom índice de um documento não poder eliminar contradições, pode ajudar a descobri-las. Considere criar um índice abrangente para especificações e outros documentos.

1 Esta seção e outras na primeira parte deste capítulo foram adaptadas da contribuição de Darrel Ince para a quinta edição européia de *Engenharia de Software*.

Document shared on www.docsity.com

Nesse resumo, a palavra *deve* se refere à senha ou à identidade do operador?

Termos vagos freqüentemente ocorrem porque especificação de sistema é um documento muito volumoso. Conseguir um alto nível de precisão consistentemente é uma tarefa quase impossível.

> "Cometer erros é humano, repeti-los também é."
>
> **Malcolm Forbes**

Incompleteza é provavelmente um dos problemas que ocorrem mais freqüentemente com especificações de sistemas. Por exemplo, considere o requisito funcional:

> O sistema deve manter o nível do reservatório de hora em hora, com base em sensores de profundidade situados no reservatório. Esses valores devem ser armazenados pelos últimos seis meses.

AVISO

Revisões técnicas formais eficazes podem eliminar muitos desses problemas. No entanto, alguns não serão descobertos. Fique atento a deficiências durante o projeto, código e teste.

Isso descreve a parte do sistema correspondente ao principal armazenamento de dados. Se um dos comandos do sistema fosse:

> A função do comando MÉDIA é exibir em um PC o nível médio da água para um sensor particular, entre dois momentos.

Considerando que não houvesse mais detalhes para esse comando, o comando estaria seriamente incompleto. Por exemplo, a descrição do comando não inclui o que deve acontecer se um usuário do sistema especificar um instante ocorrido há mais de seis meses.

Níveis de abstração misturados ocorrem quando declarações muito abstratas estão misturadas aleatoriamente com declarações em um nível muito baixo de detalhe. Apesar de esses dois tipos de declarações serem importantes em uma especificação de sistema, os especificadores conseguem freqüentemente misturá-las tanto, que se torna muito difícil ver a arquitetura funcional global de um sistema.

28.1.2 Matemática no Desenvolvimento de Software

A matemática tem muitas propriedades úteis para os desenvolvedores de grandes sistemas. Uma das suas propriedades mais úteis é ser capaz de descrever sucinta e exatamente uma situação física, um objeto ou o resultado de uma ação. Uma especificação de um sistema baseado em computador pode ser desenvolvida usando matemática especializada de modo bastante análogo ao que um engenheiro eletricista pode usar matemática para descrever um circuito[2].

Matemática dá suporte à abstração e, assim, é um excelente meio de modelagem. Como é um meio exato há pouca possibilidade de ambigüidade. Especificações podem ser validadas matematicamente para contradições e incompleteza, e os pontos vagos que podem ser eliminados. Além disso, a matemática pode ser usada para representar níveis de abstração em uma especificação de sistema, de um modo organizado.

Finalmente, a matemática fornece um alto nível de validação, quando é usada como meio de desenvolvimento de software. É possível usar uma prova matemática para demonstrar que um projeto satisfaz uma especificação e que algum código de programa é um rêflexo correto de um projeto.

28.1.3 Conceitos de Métodos Formais

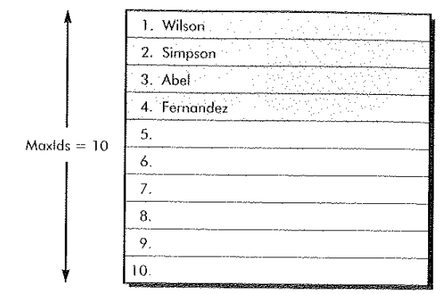
O objetivo desta seção é apresentar os conceitos principais envolvidos na especificação matemática de sistemas de software sem sobrecarregar o leitor com demasiado detalhe matemático. Usamos alguns exemplos simples para conseguir isso.

Exemplo 1: uma tabela de símbolos. Um programa é usado para manter uma tabela de símbolos. Tal tabela é usada freqüentemente em muitos tipos diferentes de aplicações. Ela consiste em uma coleção de itens sem nenhuma duplicação. Um exemplo de uma típica tabela de símbolos

2 Uma palavra de cautela é adequada neste ponto. As especificações matemáticas de sistema, apresentadas neste capítulo, não são tão sucintas quanto a especificação matemática de um simples circuito. Sistemas de software são notoriamente complexos e seria irreal especificá-los em uma única linha de matemática.

FIGURA 28.1

Uma tabela de símbolos



é mostrado na Figura 28.1. Representa a tabela utilizada por um sistema operacional para guardar os nomes dos usuários do sistema. Outros exemplos de tabelas incluem a coleção de nomes de pessoal em um sistema de pagamento, ou a coleção de nomes de computadores em um sistema de rede de comunicações. Considere que a tabela apresentada nesse exemplo consiste em não mais que *MaxIds* membros do pessoal. Essa declaração, que coloca uma restrição na tabela, é o componente de uma condição conhecida como *invariante de dados* — uma idéia importante à qual vamos voltar ao longo deste capítulo.

PONTO CHAVE

Invariante de dados é um conjunto de condições que são verdadeiras ao longo da execução do sistema que contém uma coleção de dados.

Um invariante de dados é uma condição que é verdadeira ao longo da execução do sistema que contém uma coleção de dados. O invariante de dados que prevalece para a tabela de símbolos que acabou de ser discutida tem dois componentes: (1) a tabela não conterá mais do que *MaxIds* nomes e (2) não haverá nomes duplicados na tabela. No caso do programa da tabela de símbolos, isso significa que independentemente de quando a tabela de símbolos é examinada durante a execução do sistema, ela não contém mais que *MaxIds* identificadores de pessoal e não conterá duplicatas.

Outro conceito importante é o de *estado*. Muitas linguagens formais, tais como OCL (Seção 28.5), usam a noção de um estado como foi discutido nos Capítulos 7 e 8; isto é, um sistema pode estar em um de vários estados, cada um representando um modo de comportamento externamente observável. No entanto, uma definição diferente do termo *estado* é usada na linguagem Z (Seção 28.6). Em Z (e linguagens relacionadas), o estado de um sistema é representado por dados armazenados no sistema (assim, Z sugere um número muito maior de estados, representando cada possível configuração de dados). Usando essa última definição no exemplo do programa da tabela de símbolos, o estado é a tabela de símbolo. O conceito final é o de *operação*. Trata-se de uma ação que tem lugar no sistema e lê ou escreve dados em um estado. Se o programa da tabela de símbolos diz respeito a adicionar e remover nomes de pessoal ou da tabela de símbolos, então ele estará associado com duas operações: uma operação para *adicionar* um nome especificado à tabela de símbolos e outra para *remover* um nome existente na tabela.[3] Se o programa oferece a possibilidade de verificar se um nome específico está contido na tabela, então haveria uma operação que retornaria alguma indicação informando se o nome está na tabela.

AVISO

Outro modo de olhar para a noção de estado é dizer que os dados determinam o estado. Isto é, você pode examinar os dados para ver em que estado o sistema está.

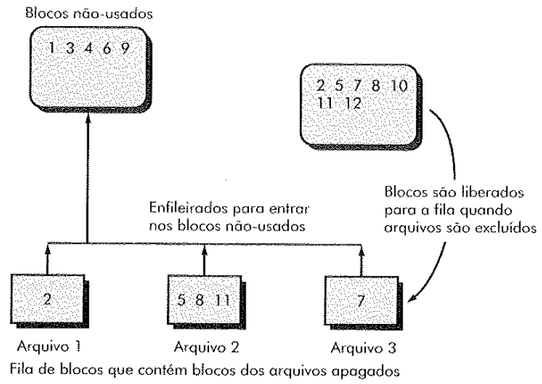
Três tipos de condições podem ser associadas com operações: invariantes, pré-condição e pós-condição. Um *invariante* define o que é garantido não mudar. Por exemplo, a tabela de símbolo tem um invariante que diz que o número de elementos é sempre menor ou igual a *MaxIds*. Uma *pré-condição* define as circunstâncias nas quais uma operação particular é válida. Por exemplo, a pré-condição para uma operação que adiciona um nome à tabela de símbolos de identificadores do pessoal é válida apenas se o nome a ser adicionado não estiver contido na tabela, e se houver também menos que MaxIds identificadores de pessoal na tabela. A *pós-condição* de uma operação define o que é garantido ser verdade quando uma operação é completada. Isso é definido pelo seu efeito sobre os dados. No exemplo de uma operação que adiciona um identificador à tabela de símbolos de identificadores de pessoal, a pós-condição especificaria matematicamente que a tabela foi aumentada com um novo identificador.

3 Deve-se observar que a adição de um nome não pode ocorrer no estado *completo*, e remover um nome é impossível em um estado *vazio*.

Document shared on www.docsity.com

FIGURA 28.2

Um manipulador de blocos



AVISO

Técnicas de brainstorming podem funcionar bem quando você precisa desenvolver um invariante de dados para uma função razoavelmente complexa. Peça a cada membro da equipe de software para escrever os limites, as restrições e as limitações da função e depois combine e edite o que fizeram.

Exemplo 2: um tratador de blocos. Uma das partes mais importantes do sistema operacional de computadores é o subsistema que mantém os arquivos criados pelo usuário. Parte do subsistema de arquivamento é o *tratador de blocos*. Os arquivos na memória são compostos de blocos de armazenagem, que são mantidos em um dispositivo de armazenagem de arquivos. Durante a operação do computador, arquivos serão criados e apagados, requerendo a aquisição e a liberação de blocos de armazenagem. Para fazer face a isso, o subsistema de arquivamento manterá um reservatório de blocos não-usados (livres) e o registro de blocos atualmente em uso. Quando são liberados blocos de um arquivo apagado, eles são normalmente adicionados a uma fila de blocos que estão aguardando para ser adicionados ao reservatório dos não-usados. Isso é mostrado na Figura 28.2. Nessa figura, são mostrados alguns componentes: o reservatório de blocos não-usados, os blocos que constituem os arquivos administrados atualmente pelo sistema operacional e os que estão esperando para ser adicionados ao reservatório. Os blocos que estão esperando são mantidos em uma fila, em que cada elemento da fila contém um conjunto de blocos de um arquivo apagado.

Para esse subsistema o estado é a coleção de blocos livres, a coleção de blocos usados e a fila de blocos devolvidos. O invariante de dados, expresso em linguagem natural, é:

- Nenhum bloco será marcado ao mesmo tempo como não-usado e usado.
- Todos os conjuntos de blocos mantidos na fila serão subconjuntos da coleção de blocos atualmente usados.
- Nenhum elemento da fila vai conter o mesmo número de blocos que outro elemento.
- A coleção de blocos usados e de não-usados será a coleção total de blocos que forma os arquivos.
- A coleção de blocos não-usados não terá número de blocos duplicado.
- A coleção de blocos usados não terá número de blocos duplicado.

Algumas das operações associadas com invariante de dados são: *add()* uma coleção de blocos ao fim da fila, *remove()* uma coleção de blocos usados da frente da fila e posicionados na coleção de blocos não usados, e *check()* se a fila de blocos está vazia.

A pré-condição da primeira operação é que os blocos a ser adicionados devem estar na coleção de blocos usados. A pós-condição é que a coleção de blocos deve ser adicionada ao fim da fila. A pré-condição da segunda operação é que a fila deve ter pelo menos um item. A pós-condição é que os blocos devem ser adicionados à coleção de blocos não-usados. A operação final — verificar se a fila de blocos devolvidos está vazia — não tem pré-condição. Isso significa que a operação está

sempre definida, independentemente do valor do estado. A pós-condição emprega o valor *verdadeiro* (*true*) se a fila está vazia e *falso* (*false*) no caso contrário.

Nos exemplos mencionados nesta seção, introduzimos conceitos importantes de especificação formal. Mas o fizemos sem enfatizar a matemática, que é necessária para fazer a especificação formal. Na Seção 28.2, consideraremos essa matemática.

28.2 PRELIMINARES MATEMÁTICAS

Para aplicar métodos formais efetivamente, um engenheiro de software deve saber trabalhar com notação matemática associada a conjuntos e seqüências e com notação lógica usada em cálculo de predicados. O objetivo desta seção é fornecer uma breve introdução. Para uma discussão mais detalhada, o leitor deve examinar livros dedicados a esses assuntos (por exemplo, [WIL87], [GRI93] e [ROS95]).

28.2.1 Conjuntos e Especificação Construtiva

Conjunto é uma coleção de objetos ou elementos, e é usado como a base dos métodos formais. Os elementos contidos em um conjunto são únicos (i. e., não são permitidas duplicatas). Conjuntos com um pequeno número de elementos são escritos dentro de chaves, com os elementos separados por vírgulas. Por exemplo, o conjunto

{C++, Smalltalk, Ada, COBOL, Java}

contém os nomes de cinco linguagens de programação.

A ordem em que os elementos aparecem no conjunto é irrelevante. O número de itens em um conjunto é conhecido como sua *cardinalidade*. O operador # retorna a cardinalidade de um conjunto. Por exemplo, a expressão

$$\#\{A, B, C, D\} = 4$$

implica que o operador de cardinalidade foi aplicado ao conjunto mostrado e o resultado indica o número de itens no conjunto.

? O que é especificação construtiva de um conjunto?

Há dois modos para definir um conjunto. Um conjunto pode ser definido pela enumeração dos seus elementos (é como acabaram de ser definidos os conjuntos mencionados). A segunda abordagem é criar uma *especificação construtiva do conjunto*. A forma geral dos membros de um conjunto é especificada usando uma expressão booleana. É preferível a especificação construtiva de um conjunto, em vez da enumeração, porque ela permite uma definição sucinta de grandes conjuntos. Também define explicitamente a regra que foi usada na construção do conjunto. Considere o seguinte exemplo de especificação construtiva:

$$\{n: \mathbb{N} \mid n < 3 \bullet n\}$$

Essa especificação tem três componentes, uma assinatura, $n : \mathbb{N}$, um predicado $n < 3$, e um termo n. A *assinatura* especifica o intervalo de valores que será considerado quando da formação do conjunto, o *predicado* (expressão booleana) define uma restrição do conjunto e, finalmente, o *termo* dá a forma geral do item do conjunto. Nesse exemplo, $\mathbb{N}$ representa os números naturais; conseqüentemente, os números naturais devem ser considerados. O predicado indica que apenas os números naturais menores do que 3 devem ser incluídos; e o termo especifica que cada elemento do conjunto será da forma n. Assim sendo, essa especificação define o conjunto

$$\{0, 1, 2\}$$

Quando a forma dos elementos de um conjunto é óbvia, o termo pode ser omitido. Por exemplo, o conjunto anterior pode ser especificado como

$$\{n : \mathbb{N} \mid n < 3\}$$

Document shared on www.docsity.com

AVISO

O conhecimento de operações de conjuntos é indispensável quando são desenvolvidas especificações formais. Familiarize-se com cada uma, se você pretende aplicar métodos formais.

Todos os conjuntos que foram descritos aqui têm elementos que são itens individuais. Também podem ser formados conjuntos de pares de elementos, de terno de elementos etc. Por exemplo, a especificação de conjunto

$$\{x, y : \mathbb{N} \mid x + y = 10 \bullet (x, y^2)\}$$

descreve o conjunto de pares de números naturais que têm a forma (x, y^2) e cuja a soma de x e y é 10. Trata-se então do conjunto

$$\{(1, 81), (2, 64), (3, 49), \ldots\}$$

Obviamente, uma especificação construtiva de um conjunto, necessária para representar algum componente de software de computador, pode ser consideravelmente mais complexa do que essas aqui mencionadas. No entanto, a forma básica e a estrutura permanecem as mesmas.

28.2.2 Operadores de Conjuntos

Uma simbologia especializada em conjuntos é usada para representar operações de conjuntos e lógicas. Esses símbolos precisam ser entendidos pelo engenheiro de software que pretende aplicar métodos formais.

O operador $\in$ é usado para indicar que um elemento pertence a um conjunto. Por exemplo, a expressão

$$X \in X$$

tem o valor *verdadeiro* se x é elemento do conjunto X, e o valor *falso* no caso contrário. Por exemplo, o predicado

$$12 \in \{6, 1, 12, 22\}$$

tem o valor *verdadeiro,* porque 12 é elemento desse conjunto.

O oposto do operador $\in$ é o operador $\notin$. A expressão

$$x \notin X$$

tem o valor *verdadeiro* se x não é elemento do conjunto X, e *falso* no caso contrário. Por exemplo, o predicado

$$13 \notin \{13, 1, 124, 22\}$$

tem o valor *falso.*

Os operadores $\subset$ e $\subseteq$ têm conjuntos como seus operandos. O predicado

$$A \subset B$$

tem o valor *verdadeiro* se os elementos do conjunto A pertencem ao conjunto B, e tem o valor *falso* no caso contrário. Assim, o predicado

$$\{1, 2\} \subset \{4, 3, 1, 2\}$$

tem o valor *verdadeiro.* No entanto, o predicado

$$\{HD1, LP4, RC5\} \subset \{HD1, RC2, HD3, LP1, LP4, LP6\}$$

tem o valor *falso,* porque o elemento RC5 não pertence ao conjunto à direita do operador.

O operador $\subseteq$ é semelhante a $\subset$. Mas se seus operandos são iguais, ele tem o valor verdadeiro. Assim, o valor do predicado

$$\{HD1, LP4, RC5\} \subseteq \{HD1, RC2, HD3, LP1, LP4, LP6\}$$

é *falso,* e o predicado

$$\{HD1, LP4, RC5\} \subseteq \{HD1, LP4, RC5\}$$

é *verdadeiro.*

> "As estruturas matemáticas estão entre as descobertas mais bonitas feitas pela mente humana."
>
> **Douglas Hofstadter**

Um conjunto especial é o conjunto vazio $\varnothing$. Ele corresponde ao zero na matemática normal. O conjunto vazio tem a propriedade de ser um subconjunto de qualquer outro conjunto. Duas identidades úteis envolvendo o conjunto vazio são

$$\varnothing \cup A = A \text{ e } \varnothing \cap A = \varnothing$$

para qualquer conjunto A, em que $\cup$ é conhecido como *operador união,* algumas vezes conhecido como *copo;* $\cap$ é o *operador interseção,* algumas vezes conhecido como *chapéu.*

O operador união toma dois conjuntos e forma um terceiro conjunto, que contém todos os elementos dos dois conjuntos com eliminação das duplicatas. Assim, o resultado da expressão

$$\{Arquivo1, Arquivo2, Impostos, Compilador\} \cap \{NovosImpostos, D2, D3, Arquivo2\}$$

é o conjunto

$$\{Arquivo\ 1, Arquivo\ 2, Impostos, Compilador, NovosImpostos, D2, D3\}$$

O operador interseção toma dois conjuntos e forma um terceiro conjunto, cujos elementos são comuns aos dois conjuntos. Assim, a expressão

$$\{12, 4, 99, 1\} \cap \{1, 13, 12, 77\}$$

resulta no conjunto $\{12, 1\}$.

O *operador diferença,* \, como o nome sugere, resulta em um conjunto formado pela remoção dos elementos do segundo operando no primeiro operando. Assim, o valor da expressão

$$\{Novo, Antigo, Arquivo\ de\ Impostos, ParamSist\} \setminus \{Antigo, ParamSist\}$$

resulta no conjunto {Novo, Arquivo de Impostos}.

O valor da expressão

$$\{a, b, c, d\} \cap \{x, y\}$$

é o conjunto vazio $\varnothing$. O operador sempre produz um conjunto; no entanto, neste caso, não há elementos comuns entre seus operandos, assim, o conjunto resultante não terá elementos.

O último operador é o *produto em cruz,* $\times$, algumas vezes conhecido como *produto cartesiano.* Ele tem dois operandos que formam um par de conjuntos. O resultado é um conjunto cujos elementos são pares ordenados, em que cada par é formado por um elemento do primeiro operando combinado com um elemento do segundo operando. Um exemplo de uma expressão envolvendo o produto em cruz é

$$\{1, 2\} \times \{4, 5, 6\}$$

O resultado dessa expressão é

$$\{(1, 4), (1, 5), (1, 6), (2, 4), (2, 5), (2, 6)\}$$

Note que cada elemento do primeiro operando é combinado com cada elemento do segundo operando.

Document shared on www.docsity.com

Um conceito importante para métodos formais é o de *conjunto potência*, que é formado pelos subconjuntos daquele conjunto. O símbolo usado para o operador conjunto potência nesse capítulo é $\mathbb{P}$. Trata-se de um operador unitário que, quando aplicado a um conjunto, produz o conjunto dos subconjuntos do seu operando. Por exemplo:

$$\mathbb{P}\{1, 2, 3\} = \{\varnothing, \{1\}, \{2\}\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

pois todos os elementos são subconjuntos de $\{1, 2, 3\}$.

28.2.3 Operadores Lógicos

Outro componente importante de um método formal é a lógica: a álgebra das expressões verdadeiras e falsas. O significado dos operadores lógicos comuns é bem entendido por todos os engenheiros de software. No entanto, os operadores lógicos associados com as linguagens de programação comuns são escritos usando os símbolos já disponíveis nos teclados. Os operadores matemáticos equivalentes a eles são:

$\wedge$	e
$\vee$	ou
$\neg$	não
$\Rightarrow$	implica

A *quantificação universal* é um modo de fazer uma declaração sobre os elementos de um conjunto, que é verdadeira para todos os elementos do conjunto. A quantificação universal usa o símbolo $\forall$. Um exemplo do seu uso é

$$\forall\, i, j: \mathbb{N} \bullet i > j \Rightarrow i^2 > j^2$$

que declara que para todo par de valores no conjunto dos números naturais, se i é maior do que j então i^2 é maior do que j^2.

28.2.4 Seqüências

Uma seqüência é uma estrutura matemática que modela o fato de que seus elementos são ordenados. Uma seqüência s é um conjunto de pares ordenados cujos elementos vão de 1 ao elemento de maior número. Por exemplo:

$$\{(1, \text{Jones}), (2, \text{Wilson}), (3, \text{Shapiro}), (4, \text{Estavez})\}$$

é uma seqüência. Os itens que formam os primeiros elementos dos pares ordenados são conhecidos coletivamente como *domínio* da seqüência, e a coleção dos segundos elementos é conhecida como *contradomínio* da seqüência. Neste livro, as seqüências são indicadas usando parênteses angulares. Por exemplo, a seqüência precedente normalmente seria escrita como $\langle$Jones, Wilson, Shapiro, Estavez$\rangle$.

Diferentemente de conjuntos, a duplicação é permitida em uma seqüência e a ordem de uma seqüência é importante. Assim,

$$\langle \text{Jones, Wilson, Shapiro} \rangle \neq \langle \text{Jones, Shapiro, Wilson} \rangle$$

A seqüência vazia é representada como $\langle\,\rangle$.

Vários operadores de seqüência são usados nas especificações formais. Concatenação, $\frown$, é um operador binário que forma uma seqüência construída adicionando-se o segundo operando ao fim do primeiro operando. Por exemplo:

$$\langle 2, 3, 34, 1 \rangle \frown \langle 12, 33, 34, 200 \rangle$$

resulta na seqüência $\langle 2, 3, 34, 1, 12, 33, 34, 200 \rangle$.

Outros operadores que podem ser aplicados a uma seqüência são *cabeça* (*head*), *cauda* (*tail*), *frente* (*front*) e *último* (*last*). O operador *cabeça* extrai o primeiro elemento de uma seqüência; *cauda* reproduz os últimos $n - 1$ elementos em uma seqüência de tamanho n; *último* extrai o elemento final de uma seqüência; e *frente* reproduz os primeiros $n - 1$ elementos de uma seqüência de tamanho n. Por exemplo:

cabeça $\langle 2, 3, 34, 1, 99, 100 = 2 \rangle$
cauda $\langle 2, 3, 34, 1, 99, 101 \rangle = \langle 3, 34, 1, 99, 101 \rangle$
último $\langle 2, 3, 34, 1, 99, 101 \rangle = 101$
frente $\langle 2, 3, 34, 1, 99, 101 \rangle = \langle 2, 3, 34, 1, 99 \rangle$

Como uma seqüência é um conjunto de pares, todos os operadores de conjunto descritos na Seção 28.2.2 são aplicáveis. Quando uma seqüência é usada em um estado, ela deve ser indicada pelo uso da palavra-chave *seq*. Por exemplo:

ListaArquivo : *seq* ARQUIVOS
NúmeroUsuários : $\mathbb{N}$

descreve um estado com dois componentes: uma seqüência de arquivos e um número natural.

28.3 Aplicação de Notação Matemática para Especificação Formal

Retomemos o exemplo do Tratador de blocos apresentado na Seção 28.1.3, para ilustrar o uso de notação matemática na especificação formal de um componente de software. Para lembrar, um componente importante do sistema operacional de um computador mantém arquivos que foram criados por usuários. O Tratador de blocos mantém um reservatório de blocos não-usados e também rastreia blocos que estão atualmente em uso. Quando são liberados blocos de um arquivo apagado, eles normalmente são adicionados à fila de blocos, aguardando para serem adicionados ao reservatório de blocos não-usados. Isso foi mostrado esquematicamente na Figura 28.2.[4]

Um conjunto denominado *BLOCOS* consiste em todos os números de blocos. *TodBlocos* é um conjunto de blocos que fica entre 1 e *MaxBlocos*. O estado será modelado por dois conjuntos e uma seqüência. Os dois conjuntos são *usado* e *livre*. Ambos contêm blocos — o conjunto *usado* contém os blocos atualmente usados em arquivos e o conjunto *livre* contém os blocos disponíveis para novos arquivos. A seqüência conterá conjuntos de blocos que estão prontos para ser liberados dos arquivos que foram apagados. O estado pode ser descrito como:

usado, livre: $\mathbb{P}$ *BLOCOS*
FilaBlocos: seq $\mathbb{P}$ *BLOCOS*

? Como posso representar estados e invariantes de dados usando operadores de conjuntos lógicos, que acabaram de ser introduzidos?

Isso se parece muito com a declaração das variáveis de um programa. Essa declaração afirma que *usado* e *livre* serão conjuntos de blocos e que *FilaBlocos* será uma seqüência, da qual cada elemento será um conjunto de blocos. O invariante de dados pode ser escrito como:

$usado \cup livre = \varnothing \wedge$
$usado \cap livre = TodBlocos \wedge$
$\forall i$: dom $FilaBlocos \bullet FilaBlocos\ i \subseteq usado \wedge$
$\forall\, i, j$: dom $FilaBlocos \bullet i \parallel j \Rightarrow FilaBlocos\ i \cap FilaBlocos\ j = \varnothing$

Veja na Web
Uma quantidade de informações sobre métodos formais pode ser encontrada em www.afm.sbu.ac.uk.

Os componentes matemáticos do invariante de dados coincidem com os quatro componentes de linguagem natural, indicados por bolinhas pretas descritas anteriormente. A primeira linha do invariante de dados declara que não existirão blocos comuns na coleção de blocos usados e na coleção de blocos livres. A segunda linha declara que a coleção de blocos usados e a de blocos livres será sempre igual a toda a coleção de blocos no sistema. A terceira linha indica que o i-ésimo elemento da fila de blocos será sempre um subconjunto dos blocos usados. A última linha declara que para dois elementos quaisquer da fila de blocos que não sejam os mesmos, não

4 Se sua lembrança do exemplo de manipulador de blocos não é clara, por favor, volte à Seção 28.1.3 para revisar o invariante de dados, operações, pré-condições e pós-condições associados ao manipulador de blocos.

Document shared on www.docsity.com

haverá blocos comuns nesses dois elementos. Os dois componentes finais em linguagem natural do invariante de dados são implementados em virtude do fato de *usado* e *livre* serem conjuntos e, conseqüentemente, não conterem duplicatas.

A primeira operação que vamos definir é a que remove o elemento da cabeça de uma fila de blocos. A pré-condição é de haver pelo menos um item na fila:

$$\#FilaBlocos > 0,$$

A pós-condição é que a cabeça da fila precisa ser removida e colocada na coleção de blocos livres e a fila ajustada para mostrar a remoção:

$$usado' = usado \setminus cabeça\ FilaBlocos \wedge$$
$$livre' = livre \cup cabeça\ FilaBlocos \wedge$$
$$FilaBlocos' = cauda\ FilaBlocos$$

Uma convenção usada em muitos métodos formais é de que o valor de uma variável após uma operação é indicado pelo sinal apóstrofo. Assim, o primeiro componente da expressão precedente declara que o novo bloco usado (*usado'*) será igual ao bloco usado menos os blocos que foram removidos. O segundo componente declara que o novo bloco livre (*livre'*) será o bloco livre com a cabeça da fila de blocos adicionada a ele. O terceiro componente declara que uma nova fila de blocos será igual à cauda do valor da fila de blocos, isto é, todos os elementos da fila menos o primeiro. Uma segunda operação adiciona uma coleção de blocos, *UmBloco*, à fila de blocos. A pré-condição é que *UmBloco* é atualmente um conjunto de blocos usados:

? Como represento pré e pós-condições?

$$UmBloco \subseteq usados$$

A pós-condição é que o conjunto de blocos é adicionado ao fim da fila de blocos e o conjunto de blocos usados e livres permanecem inalterados:

$$FilaBlocos' = FilaBlocos \frown \langle umBloco \rangle \wedge$$
$$usado' = usado \wedge$$
$$livre' = livre$$

Não há dúvida de que a especificação matemática da fila de blocos é consideravelmente mais rigorosa do que a narrativa em linguagem natural ou o modelo gráfico. O rigor adicional requer esforço, mas os benefícios conseguidos da consistência e completeza aperfeiçoadas podem ser justificados para muitos tipos de aplicações.

28.4 Linguagens de Especificação Formal

Uma linguagem de especificação formal é usualmente composta de três componentes principais: (1) uma *sintaxe*, que define a notação específica com a qual a especificação é representada, (2) uma *semântica*, para ajudar a definir um "universo de objetos" [WIN90] que será usado para descrever o sistema e (3) um conjunto de *relações*, que define as regras que indicam quais objetos satisfazem adequadamente a especificação.

O domínio sintático de uma linguagem de especificação formal é baseado em uma sintaxe derivada da notação padrão da teoria dos conjuntos e do cálculo de proposições. Por exemplo, variáveis como *x*, *y* e *z* descrevem um conjunto de objetos que se relacionam a um problema (algumas vezes chamado de *domínio do discurso*) e são usados em conjunto com os operadores descritos na Seção 28.2. Apesar de a sintaxe ser usualmente simbólica, ícones (por exemplo, símbolos gráficos como caixas, setas e círculos) podem também ser usados, se não forem ambíguos.

O *domínio semântico* de uma linguagem de especificação indica como a linguagem representa os requisitos do sistema. Por exemplo, uma linguagem de programação tem um conjunto de semântica formal que permite ao desenvolvedor de software especificar algoritmos que transformam as entradas em saídas. Uma gramática formal (como BNF) pode ser usada para descrever a sintaxe

da linguagem de programação. No entanto, uma linguagem de programação não constitui uma boa linguagem de especificação, porque pode representar apenas funções calculáveis. Uma linguagem de especificação deve ter um domínio semântico mais amplo, isto é, o domínio semântico de uma linguagem de especificação deve ser capaz de expressar idéias como "para todo *x* em um conjunto infinito *A*, existe *y* em um conjunto infinito *B* tal que a propriedade *P* é válida para todo *x* e *y*" [WIN90]. Outras linguagens de especificação aplicam uma semântica que permite a especificação do comportamento do sistema. Por exemplo, podem ser desenvolvidas uma sintaxe e semântica para especificar estados e transição de estados e eventos, junto com seus efeitos na transição, sincronização e temporização de estados.

É possível usar abstrações semânticas diferentes para escrever o mesmo sistema de diferentes maneiras. Fizemos isso de um modo menos formal no Capítulo 8. Classes, dados, funções e comportamento foram representados. Diferentes notações de modelagem podem ser usadas para representar o mesmo sistema. A semântica de cada representação fornece visões complementares do sistema. Para ilustrar essa abordagem, quando métodos formais são usados, considere que uma linguagem de especificação formal é usada para descrever o conjunto de eventos que faz que um estado particular ocorra em um sistema. Outra relação formal mostra todas as funções que ocorrem em um determinado estado. A intersecção dessas duas relações fornece uma indicação dos eventos que causarão a ocorrência de funções específicas.

Atualmente, uma variedade de linguagens de especificação formal está em uso. OCL [OMG03], Z ([ISO02], [SPI88], [SPI92]), LARCH [GUT93] e VDM [JON91] são linguagens representativas de especificação formal que exibem as características previamente mencionadas. Neste capítulo, apresentamos um resumo de OCL e Z.

28.5 Linguagem de Restrição de Objeto (Object Constraint Language, OCL)[5]

Linguagem de Restrição de Objetos (*Object Constraint Language*, OCL) é uma notação formal desenvolvida de modo que os usuários da UML possam adicionar mais precisão a suas especificações. Toda a potência da lógica e matemática discreta está disponível na linguagem. No entanto, os projetistas de OCL decidiram que somente caracteres ASCII (em vez de notação matemática convencional) devem ser usados em comandos OCL. Isso torna a linguagem mais amigável às pessoas que têm menos inclinação matemática, e mais facilmente processada por computador. Mas também torna OCL um pouco prolixa em certas ocasiões.

Veja na Web

Informação detalhada sobre OCL pode ser encontrada em www-3.ibm.com/software/awdtools/library/standards/ocl.html.

28.5.1 Um Breve Panorama da Sintaxe e Semântica de OCL

Para usar OCL, um engenheiro de software começa com um ou mais diagramas UML – mais comumente com diagramas de classes, de estados ou de atividades. Para esses, adicionamos expressões OCL que declaram fatos sobre elementos dos diagramas. Essas expressões são chamadas de *restrições* (*constraints*); qualquer implementação derivada do modelo deve garantir que cada uma das restrições permaneça sempre verdadeira.

Como uma linguagem de programação orientada a objetos, uma expressão OCL envolve operadores operando sobre objetos. No entanto, o resultado de uma expressão completa deve sempre ser um booleano, isto é, verdadeiro ou falso. Os objetos podem ser instâncias da classe **Collection** da OCL, das quais **Set** e **Sequence** são duas subclasses.

O objeto **self** é o elemento do diagrama UML no contexto do qual a expressão OCL está sendo avaliada. Outros objetos podem ser obtidos por navegação (*navigating*) usando o símbolo . (ponto, *dot*) partindo do objeto **self**. Por exemplo:

- Se **self** é da classe **C**, com atributo **a**, então **self.a** avalia para o objeto armazenado em **a**.

5 Esta seção teve a contribuição do prof. Timothy Lethbridge da Universidade de Ottawa e é apresentada aqui com permissão.

Document shared on www.docsity.com

- Se **C** tem uma associação um-para-muitos chamada de *assoc* com outra classe **D**, então **self.assoc** avalia para um **Set** cujos elementos são do tipo **D**.
- Finalmente (e um pouco mais sutilmente), se **D** tem atributo **b**, então a expressão **self.assoc.b** avalia para o conjunto de todos os **b**s que pertencem a todos os **D**s.

OCL fornece operações nativas (*built-in*) implementando a matemática descrita na Seção 28.2 e mais. Um pequeno exemplo delas é apresentado na Tabela 28.1.

TABELA 28.1 RESUMO DA NOTAÇÃO OCL CHAVE

Notação OCL	Significado
x.y	Obtém o propriedade y do elemento x. Uma propriedade pode ser um atributo, o conjunto de objetos no final de uma associação, o resultado da avaliação de uma operação, ou outras coisas dependendo do tipo de diagrama UML. Se x é um Set, então y é aplicado a todo elemento de x, os resultados são coletados em um novo Set.
c –> f()	Aplica-se a operação OCL *nativa f* à própria Collection c (em oposição a cada um dos objetos em c). Exemplos de operações *nativas* são listados abaixo.
and, or, =, < >	E lógico, ou lógico, igual, diferente.
p implies q	Verdadeiro se ou q é verdadeiro ou p é falso.
Exemplo de Operações sobre Collections (incluindo Sets e Sequences)	
c–> size()	O número de elementos na Collection c.
c–> isEmpty()	Verdadeiro se c não tem elementos, falso caso contrário.
c1–> includesAll(c2)	Verdadeiro se todo elemento de c2 está em c1.
c1–> excludesAll(c2)	Verdadeiro se nenhum elemento de c2 está em c1.
c–> forAll (elem | boolexpr)	Verdadeiro se boolexpr é verdadeira quando aplicado a todo elemento de c. Quando um elemento está sendo avaliado, ele está ligado à variável elem, que pode ser usada em boolexpr. Isso implementa quantificação universal, discutida anteriormente.
c–> forAll (elem1, elem2 | boolexpr)	Mesmo que a anterior, exceto que boolexpr é avaliada para todo possível *par* de elementos tirados de c, incluindo casos em que o par consiste do mesmo elemento.
c–>isUnique(elem | expr)	É verdadeira se a expressão avalia para um valor diferente quando aplicada a todo elemento de c.
Exemplos de Operações Específicas de Sets	
s1–> intersection(s2)	O conjunto daqueles elementos encontrados em s1 e também em s2.
s1–> union(s2)	O conjunto daqueles elementos encontrados em s1 ou em s2.
s1–> excluding(x)	O conjunto s1 com objeto x omitido.
Exemplo de Operação Específica de Sequences	
seq–> first()	O objeto que é o primeiro elemento em uma seqüência seq.

28.5.2 Um Exemplo Usando OCL

Nesta seção, OCL é usada para ajudar a formalizar a especificação do exemplo do Tratador de bloco, introduzido na Seção 28.1.3. O primeiro passo é desenvolver um modelo UML. Para esse exemplo, começamos com o diagrama de classe encontrado na Figura 28.3. Esse diagrama especifica muitos relacionamentos entre os objetos envolvidos, no entanto, precisamos adicionar expressões OCL para garantir que implementadores do sistema saibam exatamente o que eles devem garantir que permaneça verdadeiro quando o sistema roda.

As expressões OCL que adicionaremos correspondem a seis partes do invariante discutido na Seção 28.1.3. Para cada um, vamos repetir o invariante em português e depois dar a expressão OCL correspondente. É considerada boa prática fornecer o texto em português junto com a lógica formal, isso ajuda o leitor a entender a lógica, e também ajuda os revisores a descobrir erros, por exemplo, situações em que o português não corresponde à lógica.

1. Nenhum bloco será marcado ao mesmo tempo como não usado e usado.

```
context TratadorBlocos inv:
   (self.usado –> intersection(self.livre)) –> isEmpty()
```

   Note que cada expressão começa com a palavra-chave **context**. Isso indica o elemento do diagrama UML que a expressão restringe. Alternativamente, o engenheiro de software pode colocar a restrição diretamente no diagrama UML, entre chaves {}. A palavra-chave **self** refere-se aqui à instância de **TratadorBlocos**; a seguir, como é permitido em OCL, vamos omitir o **self**.

2. Todos os conjuntos de blocos mantidos na fila serão subconjuntos da coleção de blocos atualmente usados.

```
context TratadorBlocos inv:
   FilaBlocos–> forAll(umConjuntoBlocos | usado–>includesAll(umConjuntoBlocos))
```

3. Nenhum elemento da fila vai conter o mesmo número de blocos que outro elemento.

```
context TratadorBlocos inv:
   FilaBloco–>forAll(ConjuntoBlocos1, ConjuntoBlocos2|
      ConjuntoBlocos1<> ConjuntoBlocos2 implies
      ConjuntoBlocos1.elementos.número excludesAll(ConjuntoBlocos2.elementos.número))
```

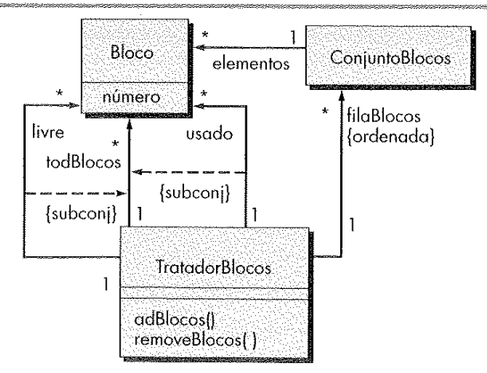
   A expressão antes de **implies** é necessária para garantir que ignoremos pares em que ambos os elementos são o mesmo bloco.

4. A coleção de blocos usados e de não-usados será a coleção total de blocos que formam os arquivos:

```
context TratadorBlocos inv:
   todBlocos = usados –> union(livre)
```

FIGURA 28.3

Diagrama de classe para um tratador de blocos



Document shared on www.docsity.com

5. A coleção de blocos não-usados não terá números de blocos duplicados.

```
context TratadorBlocos inv:
   livre -> isUnique(umBloco | umBloco.número)
```

6. A coleção de blocos usados não terá números de blocos duplicados.

```
context TratadorBlocos inv:
   usado -> isUnique(umBloco | umBloco.número)
```

OCL pode também ser usada para especificar pré-condições e pós-condições de operações. Por exemplo, considere operações que removem e adicionam conjuntos de blocos à fila. Observe que a notação **x@pre** indica o objeto **x** como existente anteriormente (*prior*) à operação; isso contraria a notação matemática discutida anteriormente, em que é o **x** depois da operação que é especialmente designado (como **x**´).

```
context TratadorBlocos::removeBlocos()
  pre: filaBlocos -> size()>0
  post: usado = usado@pre - filaBlocos@pre ->first() and
       livre = livre@pre -> union(filaBlocos@pre -> first()) and
       filaBlocos = filaBlocos@pre ->excluding(filaBlocos@pre -> first())

context TratadorBlocos::addBlocos(umConjuntoBlocos :ConjuntoBlocos)
  pre: usado -> includesAll(umConjuntoBlocos.elementos)
  post: (filaBlocos.elementos = filaBlocos.elementos@pre
       -> append(umConjuntoBlocos))and
       usado = usado@pre and
       livre = livre@pre
```

OCL é uma linguagem de modelagem, mas tem todos os atributos de uma linguagem formal. Ela permite a expressão de várias restrições, pré e pós-condições, guardas e outras características que se relacionam aos objetos representados em vários modelos UML.

28.6 A Linguagem de Especificação Z

Z (pronunciado adequadamente como "zed") é uma linguagem de especificação que evoluiu nas últimas duas décadas, tornando-se amplamente usada pela comunidade de métodos formais. A linguagem Z aplica conjuntos, relações e funções típicas no contexto de lógica de predicado de primeira ordem para construir *esquemas* – um modo de estruturar uma especificação formal.

28.6.1 Um Breve Panorama da Sintaxe e Semântica de Z

Veja na Web

Informação detalhada sobre linguagem Z pode ser encontrada em www-users.cs.york.ac.uk/~susan/abs/z.htm.

As especificações Z são organizadas como um conjunto de esquemas — uma estrutura semelhante a uma caixa que introduz variáveis e especifica os relacionamentos entre essas variáveis. Um *esquema* é essencialmente a especificação formal análoga ao componente da linguagem de programação. Do mesmo modo que componentes são usados para estruturar um sistema, esquemas são usados para estruturar uma especificação formal.

Um esquema descreve os dados armazenados a que um sistema tem acesso e altera. No contexto de Z, isso é chamado de "estado". Esse uso do termo *estado* em Z é levemente diferente do uso da palavra no restante deste livro.[6] Além disso, o esquema identifica as operações aplicadas para modificar o estado e os relacionamentos que ocorrem dentro do sistema. A estrutura genérica de um esquema tem a forma:

6 Lembramos que, em outros capítulos, *estado* foi usado para identificar o modo de comportamento de um sistema externamente observável.

NomeDoEsquema
declarações
———————————
invariante

em que declarações identificam as variáveis que estão contidas no estado do sistema e o invariante impõe restrições sobre o modo pelo qual o estado pode evoluir. Um resumo da notação da linguagem Z é apresentado na Tabela 28.2.

28.6.2 Um Exemplo Usando Z

Nesta seção, usamos a linguagem de especificação Z para modelar o exemplo do tratador de blocos introduzido anteriormente neste capítulo. O exemplo seguinte de um esquema descreve o estado do tratador de blocos e o invariante de dados:

Tratador de Blocos
usado, livre: $\mathbb{P}$ *BLOCOS*
FilaBlocos: seq $\mathbb{P}$ *BLOCOS*
usado $\cap$ *livre* = $\varnothing$ $\wedge$
usado $\cup$ *livre* = *TodosBlocos* $\wedge$
$\forall i$: **domínio** *FilaBlocos* • *FilaBlocos i* $\subseteq$ usados $\wedge$
$\forall$i, j : domínio FilaBlocos • i $\Vert$ j => FilaBlocos i $\cap$ FilaBlocos j = $\varnothing$
———————————

Como notamos, o esquema consiste de duas partes. A parte acima da linha central representa as variáveis do estado, enquanto a parte abaixo da linha central descreve o invariante de dados. Sempre que o esquema especifica operações que modificam o estado, é precedido pelo símbolo Δ. O exemplo seguinte descreve um esquema da operação que remove um elemento da fila de blocos:

RemoveBlocos
Δ *TratadorBlocos*
———————————
FilaBlocos > 0,
usado′ = *usado* \ *cabeça FilaBlocos* $\wedge$
livre′ = livre $\cup$ cabeça *FilaBlocos* $\wedge$
FilaBlocos′ = *cauda FilaBlocos*

A inclusão de Δ *FilaBlocos* resulta em todas as variáveis que constituem o estado em disponibilidade para o esquema *RemoveBloco* e garante que o invariante de dados valerá antes e depois que a operação foi executada.

A segunda operação, que adiciona uma coleção de blocos ao fim da fila, é representada como:

AdicionaBloco
Δ *TratadorBlocos*
UmBloco? : BLOCOS
———————————
UmBloco? $\subseteq$ *usado*
FilaBlocos′ = *FilaBlocos* ⌢ ⟨*UmBloco*?⟩ $\wedge$
usado′ = *usado* $\wedge$
livre′ = *livre*

Por convenção, em Z, uma variável de entrada que é lida do estado e não faz parte dele é encerrada com uma interrogação. Assim, UmBloco?, que age como um parâmetro de entrada, é encerrada com uma interrogação.

Document shared on www.docsity.com

TABELA 28.2 UM RESUMO DA NOTAÇÃO Z

A notação Z é baseada em uma teoria de conjuntos típica e na lógica de primeira ordem. Z fornece uma construção chamada de *esquema* para descrever um espaço de estado e operações de uma especificação. Um esquema agrupa declarações de variáveis com uma lista de predicados que restringem os possíveis valores de uma variável. Em Z, o esquema X é definido pela forma:

X

declarações

predicados

Funções globais e constantes são definidas pela forma

declarações

predicados

A declaração dá o tipo da função ou constante, enquanto os predicados dão o seu valor. Somente um conjunto resumido de símbolos de Z é apresentado nesta tabela.

Conjuntos:

$S : \mathbb{P}\, X$	S é declarado como um conjunto dos Xs.
$x \in S$	x é um elemento de S.
$x \notin S$	x não é um elemento de S.
$S \subseteq T$	S é um subconjunto de T: Cada elemento de S está também em T.
$S \cup T$	A união de S e T: Contém todo elemento de S ou T ou de ambos.
$S \cap T$	A intersecção de S e T: Contém todos os elementos de ambos S e T.
$S \setminus T$	A diferença de S e T: Contém todos os elementos de S exceto aqueles que estão também em T.
$\varnothing$	Conjunto vazio: Não contém elementos.
$\{x\}$	Conjunto unitário: Contém somente x.
$\mathbb{N}$	O conjunto dos números naturais 0, 1, 2, ...
$S : \mathbb{F}\, X$	S é declarado como um conjunto finito de Xs.
$\max\{S\}$	O máximo do conjunto não vazio de números S.

Funções:

$f : X \rightarrowtail\!\!\!\!\rightarrow Y$	f é declarado como uma injeção parcial de X em Y.
dom f	Domínio de f: o conjunto de valores x para os quais f(x) é definido.
ran f	Contradomínio de f: o conjunto de valores assumidos por f(x) à medida que x varia ao longo do domínio de f.
$f \oplus \{x \mapsto y\}$	Função que concorda com f exceto se x é mapeado em y.
$\{x\} \trianglelefteq f$	Uma função como f, exceto que x é removido de seu domínio.

Lógica:

$P \wedge Q$	P e Q: é verdade se tanto P quanto Q são verdades.
$P \Rightarrow Q$	P implica Q: É verdade se Q é verdade ou P é falso.
$\theta S' = \theta S$	Nenhum componente do esquema S se modifica na operação.

FERRAMENTAS DE SOFTWARE

Métodos Formais

Objetivo: O objetivo das ferramentas de métodos formais é apoiar uma equipe de software na especificação e verificação da correção.

Mecânica: A mecânica das ferramentas varia. Em geral, as ferramentas apóiam a especificação e a prova de correção automatizada, usualmente pela definição de uma linguagem especializada para prova de teorema. Muitas ferramentas não são comercializadas e foram desenvolvidas para fins de pesquisa.

Ferramentas Representativas[7]

ACL2, desenvolvida pela Universidade do Texas (www.cs.utexas.edu/users/moore/acl2/), é "tanto uma linguagem de programação, em que se pode modelar sistemas de computador, quanto uma ferramenta para auxiliar a provar propriedades desses modelos".

EVES, desenvolvida por ORA Canadá (www.ora.on.ca/eves.html), implementa a linguagem de especificação formal Verdi e um gerador de prova automatizado.

Uma lista abrangente de mais de 90 ferramentas de métodos formais pode ser encontrada em http://www.afm.sbu.ac.uk/.

28.7 OS DEZ MANDAMENTOS DOS MÉTODOS FORMAIS

A decisão de usar um método formal no mundo real não é tomada levianamente. Bowan e Hinchley [BOW95] cunharam "os dez mandamentos dos métodos formais" como diretriz para aqueles que estão em vias de aplicar essa importante abordagem de engenharia de software.[8]

1. *Escolher a notação adequada.* Para escolher eficientemente entre a ampla gama de linguagens de especificação formal, um engenheiro de software deve considerar o vocabulário da linguagem, o tipo de aplicação a ser especificado e o âmbito de uso da linguagem.
2. *Formalizar, mas não formalizar demais.* Geralmente, não é necessário aplicar métodos formais a todos os ângulos de um sistema principal. Os componentes que são críticos quanto à segurança são os primeiros a ser escolhidos, seguidos por componentes cuja falha não pode ser tolerada (por razões do negócio).
3. *Estimar custos.* Métodos formais têm altos custos iniciais. Treinamento do pessoal, aquisição de ferramentas de apoio e uso de consultores contratados resultam em altos custos iniciais que devem ser considerados quando do exame do retorno do investimento associado com métodos formais.
4. *Providenciar um especialista em métodos formais que possa ser chamado quando necessário.* Treinamento especializado e consultoria de acompanhamento é essencial para o sucesso, quando métodos formais são usados pela primeira vez.
5. *Não abandonar seus métodos tradicionais de desenvolvimento.* É possível, e em muitos casos desejável, integrar métodos formais com métodos convencionais orientados a objetos (Parte 2 deste livro). Cada um tem seus pontos fortes e fracos. Uma combinação, se adequadamente aplicada, pode produzir excelentes resultados.[9]
6. *Documentar suficientemente.* Métodos formais fornecem um método conciso, não-ambíguo e consistente para documentar requisitos do sistema. No entanto, é recomendado que um comentário em linguagem natural acompanhe a especificação formal para servir como mecanismo de reforço do entendimento do sistema pelo leitor.
7. *Não comprometer os padrões de qualidade.* "Não há nada mágico a respeito de métodos formais" [BOW95] e, por isso, outras atividades de SQA (Capítulo 26) precisam continuar a ser aplicadas à medida que o sistema é desenvolvido.

7 As ferramentas mencionadas aqui não representam uma recomendação, mas, em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

8 Este tratamento é uma versão bastante abreviada de [BOW95].

9 Engenharia de software sala limpa (Capítulo 29) é um exemplo de abordagem integrada que usa métodos formais e métodos de desenvolvimento mais convencionais.

Document shared on www.docsity.com

8. *Não ser dogmático*. O engenheiro de software deve reconhecer que métodos formais não são uma garantia de correção. É possível (alguns diriam provável) que o sistema final, mesmo quando desenvolvido usando métodos formais, possa ter pequenas omissões, defeitos sem importância e outros atributos que não satisfaçam as expectativas.
9. *Testar, testar e testar novamente*. A importância do teste de software foi discutida nos Capítulos 13 e 14. Métodos formais não absolvem o engenheiro de software da necessidade de conduzir testes bem planejados e rigorosos.
10. *Reusar*. No longo prazo, o único modo racional de reduzir os custos e melhorar a qualidade do software é por meio de reuso (Capítulo 30). Métodos formais não mudam essa realidade. Na verdade, pode ser que métodos formais sejam uma abordagem apropriada quando componentes para bibliotecas de reuso tiverem que ser criados.

28.8 Métodos Formais — A Estrada à Frente

Apesar de as técnicas de especificação formal baseadas em matemática não serem ainda usadas amplamente na indústria, elas oferecem vantagens substanciais sobre técnicas menos formais. Liskov e Bersins [LIS86] resumem isso da seguinte maneira:

> Especificações formais podem ser estudadas matematicamente, enquanto especificações informais não podem. Por exemplo, pode-se provar que um programa correto satisfaz suas especificações, ou pode-se provar que dois conjuntos alternativos de especificações são equivalentes... Certas formas de incompleteza ou inconsistência podem ser detectadas automaticamente.

Além disso, a especificação formal remove a ambigüidade e encoraja maior rigor nos primeiros estágios do processo de engenharia de software.

Mas ainda restam problemas. A especificação formal focaliza principalmente as funções e os dados. Os tópicos de tempo, controle e comportamento de um problema são mais difíceis de representar. Além disso, alguns elementos de um problema (por exemplo, interfaces humano/computador) são mais bem especificados usando técnicas gráficas ou protótipos. Finalmente, a especificação usando métodos formais é mais difícil de aprender do que métodos que incorporem notação UML, e representa um significativo "choque cultural" para alguns profissionais de software.

28.9 Resumo

Métodos formais propiciam a fundação para ambientes de especificação, que conduzem a modelos de análise mais completos, consistentes e não-ambíguos do que aqueles produzidos usando métodos convencionais ou orientados a objetos. As facilidades descritivas da teoria dos conjuntos e da notação lógica permitem a um engenheiro de software criar um enunciado claro dos fatos (requisitos).

Os conceitos subjacentes que direcionam os métodos formais são: (1) o invariante de dados, uma condição verdadeira durante toda a execução do sistema que contém uma coleção de dados; (2) o estado, uma representação do modo de comportamento externamente observável de um sistema ou (em Z e linguagens relacionadas) os dados armazenados a que um sistema tem acesso e altera e (3) a operação, uma ação que tem lugar no sistema e lê ou grava dados em um estado. Uma operação é associada a duas condições: uma pré-condição e uma pós-condição.

A matemática discreta — a notação e as heurísticas associadas com conjuntos e especificação construtiva, operadores de conjuntos, operadores lógicos e seqüências — forma a base de métodos formais. A matemática discreta é implementada no contexto de uma linguagem de especificação formal, tal como OCL e Z. Essas linguagens de especificação formal têm domínios tanto sintático quanto semântico. O domínio sintático usa uma simbologia que é aproximadamente semelhante à notação de conjuntos e cálculo de predicados. O domínio semântico permite à linguagem expressar requisitos de um modo conciso.

Decidir pelo uso de métodos formais leva em consideração os custos de implantação, bem como as modificações culturais associadas com uma tecnologia radicalmente diferente. Na maioria dos exemplos, métodos formais têm maior retorno em sistemas de segurança e de negócios críticos.

Referências Bibliográficas

[BOW95] Bowan, J. P. e Hinchley, M. G., "Ten Commandments of Formal Methods", *Computer*, v. 28, n. 4, abr. 1995.

[GRI93] Gries, D. e Schneider, F. B., *A Logical Approach to Discrete Math*, Springer-Verlag, 1993.

[GUT93] Guttag, J. V. e Horning, J. J., *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.

[HAL90] Hall, A., "Seven Myths of Formal Methods", *IEEE Software*, set. 1990, p. 11-20.

[HIN95] Hinchley, M. G. e Jarvis, S. A., *Concurrent Systems: Formal Development in CSP*, McGraw-Hill, 1995.

[HOR85] Hoare, C. A. R., *Communicating Sequential Processes*, Prentice-Hall International, 1985.

[ISO02] *Z Formal Specification Notation — Syntax, Type System and Semantics*, ISO/IEC 13568:2002, Intl. Standards Organization, 2002.

[JON91] Jones, C. B., *Systematic Software Development Using VDM*, 2ª ed., Prentice-Hall, 1991.

[LIS86] Liskov, B. H. e Berzins, V., "An Appraisal of Program Specifications", em *Software Specification Techniques*, N. Gehani e McKittrick A. T., (eds.), Addison-Wesley, 1986, p. 3.

[MAR94] Marciniak, J. J. (ed.), *Encyclopedia of Software Engineering*, Wiley, 1994.

[OMG03] "Object Constraint Language Specification", em *Unified Modeling Language*, v. 2.0, Object Management Group, set. 2003, disponível em www.omg.org.

[ROS95] Rosen, K. H., *Discrete Mathematics and Its Applications*, 3. ed., McGraw-Hill, 1995.

[SPI88] Spivey, J. M., *Understanding Z: A Specification Language and Its Formal Semantics*, Cambridge University Press, 1988.

[SPI92] _____, *The Z Notation: A Reference Manual*, Prentice-Hall, 1992.

[WIL87] Wiltala, S. A., *Discrete Mathematics: A Unified Approach*, McGraw-Hill, 1987.

[WIN90] Wing, J. M., "A Specifier's Introduction to Formal Methods", *Computer*, v. 23, n. 9, set.1990, p. 8-24.

[YOU94] Yourdon, E., "Formal Methods", *Guerrilla Programmer*, Cutter Information Corp., out. 1994.

Problemas e Pontos a Considerar

28.1. Revise os tipos de deficiência associados com abordagens menos formais para a engenharia de software na Seção 28.1.1. Dê três exemplos de cada uma partindo de sua própria experiência.

28.2. Os benefícios da matemática como um mecanismo de especificação foram longamente discutidos neste capítulo. Há contradições?

28.3. Você foi nomeado para uma equipe que está desenvolvendo um software para um modem de fax. Seu serviço é desenvolver a parte da aplicação correspondente à "lista telefônica". A função de lista telefônica permite que até *MaxNomes* de pessoas sejam armazenados junto com os correspondentes nomes das empresas, números de fax e outras informações relacionadas. Usando linguagem natural defina:

a. O invariante de dados.
b. O estado.
c. As operações prováveis.

28.4. Você foi designado para uma equipe de software que está desenvolvendo um software chamado *Expansor de Memória*, que proporciona memória aparentemente maior que a memória física para um PC. Isso é conseguido pela identificação, coleta e reatribuição de blocos de memória atribuídos a uma aplicação existente, porém não estão sendo usados. Os blocos não usados são redistribuídos a aplicações que requerem memória adicional. Fazendo as pressuposições adequadas e usando linguagem natural defina:

a. O invariante de dados.
b. O estado.
c. As operações prováveis.

28.5. Desenvolva uma especificação construtiva para um conjunto que contém n-uplas de números naturais da forma (x, y, z^2), tais que a soma de x e y é igual a z.

28.6. O instalador de uma aplicação baseada em PC define primeiro se um conjunto aceitável de recursos de hardware e de sistemas está presente. Verifica a configuração de hardware para determinar se os vários dispositivos (dos muitos dispositivos possíveis) estão presentes e define se as versões específicas do software e dos controla-

Document shared on www.docsity.com

dores do sistema já estão instaladas. Que operador de conjuntos poderia ser usado para conseguir isso? Dê um exemplo nesse contexto.

28.7. Tente desenvolver uma expressão usando operadores lógicos e de conjuntos para o seguinte enunciado: "Para todo *x* e *y*, se *x* é pai de *y* e *y* é pai de *z*, então *x* é avô de *z*. Todo mundo tem um pai". Sugestão: use a função *P(x, y)* e *G(x, z)* para representar as funções pai e avô, respectivamente.

28.8. Desenvolva uma especificação de conjunto construtiva do conjunto de pares em que o primeiro elemento de cada par é a soma de dois números naturais diferentes de zero e o segundo elemento é a diferença entre os mesmos números. Ambos os números devem estar entre 100 e 200, inclusive.

28.9. Desenvolva uma descrição matemática do estado e invariante de dados para o Problema 28.3. Refine essa descrição na linguagem de especificação OCL ou Z.

28.10. Desenvolva uma descrição matemática do estado e invariante de dados para o Problema 28.4. Refine essa descrição na linguagem de especificação OCL ou Z.

28.11. Usando a notação de OCL ou Z apresentada na Tabela 28.1 ou 28.2, selecione uma parte do sistema de segurança *CasaSegura*, descrito neste livro, e tente especificá-la com OCL ou Z.

28.12. Usando uma ou mais das fontes de informação mencionadas nas referências deste capítulo ou nas Leituras e Fontes de Informação Adicionais, desenvolva uma apresentação de meia hora sobre a sintaxe e a semântica básicas de uma linguagem de especificação formal diferente de OCL ou Z.

Leituras e Fontes de Informação Adicionais

Além dos livros usados como referência neste capítulo, um grande número de livros sobre assuntos de métodos formais tem sido publicado ao longo da última década. Uma listagem de alguns dos mais úteis é a seguinte:

Bowan, J., *Formal Specification and Documentation Using Z: A Case Study Approach*, International Thomson Computer Press, 1996.

Casey, C., *A Programming Approach to Formal Methods*, McGraw-Hill, 2000.

Cooper, D. e Barden, R., *Z in Practice*, Prentice-Hall, 1995.

Craigen, D., Gerhart, S. e Ralston, T., *Industrial Application of Formal Methods to Model, Design and Analyze Computer Systems*, Noyes Data Corp., 1995.

Harry, A., *Formal Methods Fact File: VDM and Z*, Wiley, 1997.

Hinchley, M. e Bowan, J., *Applications of Formal Methods*, Prentice-Hall, 1995.

Hinchley, M. e Bowan, J., *Industrial Strenght Formal Methods*, Academic Press, 1997.

Hussmann, H., *Formal Foundations for Software Engineering Methods*, Springer-Verlag, 1997.

Jacky, J., *The Way of Z: Practical Programming with Formal Methods*, Cambridge University Press, 1997.

Monin, F. e Hinchley, M., *Understanding Formal Methods*, Springer-Verlag, 2003.

Rann, D., J. Turner e Whitworth, J., *Z: A Beginner's Guide*, Chapman and Hall, 1994.

Ratcliff, B., *Introducing Specification Using Z: A Practical Case Study Approach*, McGraw-Hill, 1994.

Sheppard, D., *An Introduction to Formal Specification with Z and VDM*, McGraw-Hill, 1995.

Warmer, J. e Kleppe, A., *Object Constraint Language*, Addison-Wesley, 1998.

Dean (*Essence of Discrete Mathematics*, Prentice-Hall, 1996), Gries e Schneider [GRI93] e Lipschultz e Lipson (*2000 Solved Problems in Discrete Mathematics*, McGraw-Hill, 1991) apresentam informação útil para aqueles que precisam aprender mais sobre o fundamento de métodos formais.

Uma ampla variedade de fontes de informação sobre métodos formais está disponível na Internet. Uma lista atualizada de referências da World Wide Web que são relevantes pode ser encontrada no site deste livro: **http://www.mhhe.com/pressman**.

Engenharia de Software Sala Limpa

CAPÍTULO 29

Conceitos-Chave

certificação 659
especificação caixa-clara .. 652
especificação caixa de estado 652
especificação caixa-preta .. 652
especificação funcional ... 651
estratégia sala limpa 648
estrutura de caixas 652
prova de correção 654
refinamento do projeto ... 653
subprovas 655
teste estatístico de uso 658
verificação 653

O uso integrado da modelagem da engenharia de software convencional (e possivelmente de métodos formais), da verificação de programas (provas de correção) e da SQA estatística foram combinados em uma técnica que pode conduzir a um software de qualidade extremamente alta. A *engenharia de software sala limpa* é uma abordagem que enfatiza a necessidade de fazer a correção no software à medida que vai sendo desenvolvido. Em vez do ciclo clássico de análise, projeto, código, teste e depuração, a abordagem sala limpa sugere um ponto de vista diferente [LIN94]:

> A filosofia por trás da engenharia de software sala limpa é evitar a dependência de processos de remoção de defeitos dispendiosos escrevendo os incrementos de código corretos da primeira vez e verificando sua correção antes do teste. Seu modelo de processo incorpora a certificação estatística da qualidade dos incrementos de código, à medida que se acumulam em um sistema.

De vários modos, a abordagem sala limpa eleva a engenharia de software para outro nível. Como os métodos formais apresentados no Capítulo 25, o processo sala limpa enfatiza o rigor na especificação de projeto e na verificação formal de cada elemento de projeto, usando provas de correção baseadas na matemática. Estendendo a abordagem adotada nos métodos formais, a abordagem sala limpa enfatiza também técnicas estatísticas para controle de qualidade, incluindo um teste baseado no uso antecipado do software por clientes.

Quando o software falha no mundo real, riscos imediatos e de longo prazo abundam. Os riscos podem estar relacionados à segurança humana, à perda econômica ou à operação efetiva de negócios e infra-estruturas sociais. A engenharia de software sala limpa é um modelo de processo que remove defeitos antes que eles possam precipitar riscos sérios.

Panorama

O que é? Quantas vezes você ouviu alguém dizer "faça certo da primeira vez"? Essa é a filosofia prevalente da engenharia de software sala limpa – um processo que enfatiza a verificação matemática da correção antes que comece a construção do programa e a certificação da confiabilidade do software como parte da atividade de teste. O limite inferior é uma taxa de falhas extremamente baixa, que seria difícil ou impossível conseguir usando métodos menos formais.

Quem faz? Um engenheiro de software especialmente treinado.

Por que é importante? Erros criam retrabalho, o que leva tempo e aumenta os custos. Não seria bom se pudéssemos reduzir drasticamente o número de erros (defeitos, *bugs*) introduzidos, à medida que o software é projetado e construído? Essa é a premissa da engenharia de software sala limpa.

Quais são os passos? Modelos de análise e de projeto são criados usando a representação de estruturas de caixas. Uma "caixa" encapsula o sistema (ou algum ângulo do sistema) em um nível específico de abstração. A verificação de correção é aplicada assim que o projeto da estrutura de caixas é completado. Uma vez verificada a correção de cada estrutura de caixas, o teste estatístico de uso começa. O software é testado definindo um conjunto de cenários de uso, determinando a probabilidade de uso para cada cenário e depois definindo testes aleatórios de acordo com as probabilidades. Os registros de erros que resultam são analisados para permitir o cálculo matemático da confiabilidade projetada para o componente de software.

Qual é o produto do trabalho? Desenvolvimento de especificações caixa-preta, caixa de estado e caixa-clara. Registro dos resultados das provas formais de correção e dos testes estatísticos de uso.

Document shared on www.docsity.com

Como tenho certeza de que fiz corretamente? Uma prova formal de correção é aplicada à especificação da estrutura de caixas. Um teste estatístico de uso exercita os cenários de uso para garantir que os erros na funcionalidade definida pelo usuário sejam descobertos e corrigidos. Os dados de teste são usados para fornecer uma indicação da confiabilidade do software.

29.1 A Abordagem Sala Limpa

A filosofia "sala limpa" nas tecnologias de fabricação de hardware é realmente bastante simples: É providencial em termos de custo e em termos de prazo estabelecer uma abordagem de fabricação que evite a introdução de defeitos de produto. Em vez de fabricar um produto e depois trabalhar para remover defeitos, a abordagem sala limpa exige a disciplina necessária para eliminar defeitos na especificação e no projeto e depois fabricar de uma maneira "limpa".

A filosofia sala limpa foi proposta para engenharia de software inicialmente por Mills, Dyer e Linger [MIL87], durante os anos de 1980. Apesar de as primeiras experiências com essa abordagem disciplinada para o trabalho de software terem se mostrado significativamente promissoras [HAU94], ela não ganhou ampla utilização. Henderson [HEN95] sugere três razões possíveis:

1. Uma crença de que a metodologia sala limpa é demasiado teórica, demasiado matemática e demasiado radical para uso no desenvolvimento real de software.
2. Ela advoga que não seja feito teste de unidade pelos desenvolvedores, mas em vez disso o substitui por verificação de correção e controle estatístico de qualidade conceitos que representam um afastamento importante do modo pelo qual a maioria do software é desenvolvida atualmente.
3. A maturidade da indústria de desenvolvimento de software. O uso de processos sala limpa requer aplicação rigorosa de processos definidos em todas as fases do ciclo de vida. Como a maioria das indústrias ainda está operando em um nível relativamente baixo de maturidade de processo, engenheiros de software ainda não estão prontos para aplicar técnicas de sala limpa.

Apesar das verdades em cada uma dessas preocupações, os benefícios em potencial da engenharia de software sala limpa sobrepujam de longe o investimento necessário para vencer a resistência cultural que está no âmago dessas preocupações.

> "O único modo de erros ocorrerem em um programa é por terem sido postos lá pelo autor. Nenhum outro mecanismo é conhecido... A prática correta visa a prevenir inserção de erros e, não conseguindo isso, removê-los antes do teste ou de qualquer outra execução do programa."
>
> Harlan Mills

29.1.1 A Estratégia Sala Limpa

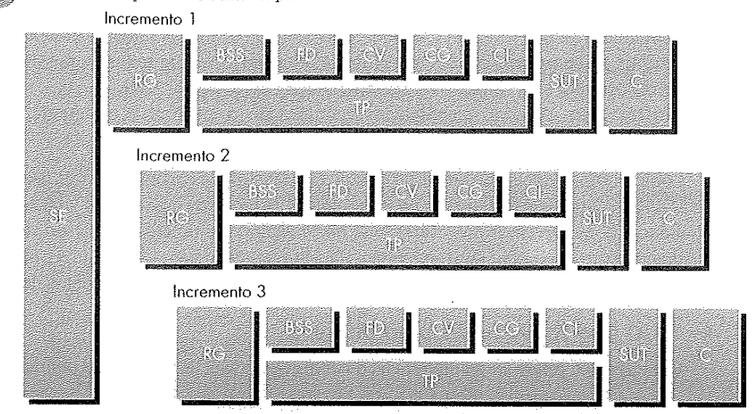
A abordagem sala limpa faz uso de uma versão especializada do modelo incremental de software (Capítulo 3). Uma "seqüência de incrementos de software" [LIN94] é desenvolvida por pequenas equipes independentes de engenharia de software. À medida que cada incremento é certificado, é integrado em um todo. Conseqüentemente, a funcionalidade do sistema cresce com o passar do tempo.

A seqüência de tarefas sala limpa para cada incremento é ilustrada na Figura 29.1. Os requisitos do sistema geral ou do produto são desenvolvidos usando os métodos de engenharia de sistemas discutidos no Capítulo 6. Uma vez tendo sido atribuída a funcionalidade ao elemento de software do sistema, a seqüência de incrementos sala limpa é iniciada. As seguintes tarefas ocorrem:

? Quais são as principais tarefas conduzidas como parte da engenharia de software sala limpa?

Planejamento incremental. É desenvolvido um plano de projeto que adota a estratégia incremental. A funcionalidade de cada incremento, seu tamanho projetado e um cronograma de desenvolvimento sala limpa são criados. Cuidado especial precisa ser tomado para garantir que incrementos certificados sejam integrados de modo oportuno.

FIGURA 29.1 O modelo de processo sala limpa



SE — engenharia de sistemas
RG — coleta de requisitos
BSS — especificação da estrutura de blocos
FD — projeto formal
CV — verificação de correção
CG — geração de código
CI — inspeção de código
SUT — teste estatístico de uso
C — certificação
TP — planejamento de teste

Coleta de requisitos. Usando técnicas similares àquelas introduzidas no Capítulo 7, é desenvolvida uma descrição mais detalhada dos requisitos em nível de cliente (para cada incremento).

Especificação da estrutura de caixa. Um método de especificação que faz uso de *estruturas de caixa* [HEV93] é usado para descrever a especificação funcional. Obedecendo aos princípios de análise operacional discutidos nos Capítulos 5 e 7, as estruturas de caixa "isolam e separam a definição criativa do comportamento, dados e procedimentos, em cada nível de refinamento".

Projeto formal. Usando a abordagem de estruturas de caixa, o projeto sala limpa é uma extensão natural, e sem emendas, da especificação. Apesar de ser possível fazer uma clara distinção entre as duas atividades, as especificações (chamadas de *caixas-pretas*) são refinadas iterativamente (dentro de um incremento) para tornarem-se análogas a projetos arquiteturais e em nível de componentes (chamados de *caixas de estado* e *caixas-claras*, respectivamente).

Veja na Web
Uma excelente fonte de informação e recursos para a engenharia de software sala limpa pode ser encontrada em www.cleansoft.com.

Verificação de correção. A equipe sala limpa conduz uma série de atividades de verificação de correção rigorosas sobre o projeto e depois sobre o código. A verificação (Seções 29.3 e 29.4) começa com a estrutura de caixa de mais alto nível (especificação) e se move em direção ao detalhe de projeto e de código. O primeiro nível de verificação de correção ocorre pela aplicação de um conjunto de "questões de correção" [LIN88]. Se isso não demonstrar que a especificação está correta, métodos mais formais de verificação (matemática) são usados.

Geração de código, inspeção e verificação. As especificações da estrutura de caixa, representadas em uma linguagem especializada, são traduzidas para a linguagem de programação adequada. Técnicas de inspeção ou *walkthrough* normalizados (Capítulo 26) são então usados para garantir a conformidade semântica do código e das estruturas de caixa, e a correção sintática do código. Depois, a verificação de correção é conduzida para o código-fonte.

> "A engenharia de software sala limpa consegue controle estatístico de qualidade sobre o desenvolvimento de software, pela estrita separação do processo de projeto do processo de teste, em uma seqüência de desenvolvimento incremental de software."
>
> Harlan Mills

Planejamento do teste estatístico. O uso projetado do software é analisado e uma seqüência de casos de teste que praticam uma "distribuição de probabilidade" de uso é planejada e projetada (Seção 29.4). Observando a Figura 29.1, essa atividade sala limpa é conduzida em paralelo com a especificação, verificação e geração de código.

Document shared on www.docsity.com

AVISO

Sala limpa enfatiza testes que exercitam o modo pelo qual o software é realmente usado. Casos de uso fornecem excelente entrada para o processo de planejamento do teste estatístico.

Teste estatístico de uso. Lembrando que o teste exaustivo do software de computador é impossível (Capítulo 14), é sempre necessário projetar um número finito de casos de teste. Técnicas estatísticas de uso [POO88] executam uma série de testes derivada de uma amostra estatística (a distribuição de probabilidades mencionada anteriormente) de todas as possíveis execuções do programa por todos os usuários de uma população-alvo (Seção 29.4).

Certificação. Uma vez completados a verificação, a inspeção e o teste de uso (e corrigidos todos os erros), o incremento é certificado como pronto para integração.

Como outros modelos de processo de software discutidos em outras partes deste livro, o processo sala limpa apóia-se fortemente na necessidade de produzir modelos de análise e projeto de alta qualidade. Como veremos posteriormente neste capítulo, a notação de estrutura de caixas é simplesmente outro modo para um engenheiro de software representar requisitos e projeto. A distinção real da abordagem sala limpa é a aplicação da verificação formal aos modelos de engenharia.

29.1.2 O Que Torna Sala Limpa Diferente?

Dyer [DYE92] refere-se às diferenças da abordagem sala limpa quando define o processo:

> Sala limpa representa a primeira tentativa prática de colocar o processo de desenvolvimento de software sob controle estatístico de qualidade, com uma estratégia bem definida para o contínuo aperfeiçoamento do processo. Para atingir essa meta, foi definido um ciclo de vida específico sala limpa, que focaliza a engenharia de software baseada em matemática para projetos de software corretos e teste de software baseado em estatística para certificação da confiabilidade do software.

PONTO CHAVE

As características de distinção mais importantes da sala limpa são a prova de correção e o teste estatístico de uso.

A engenharia de software sala limpa difere das visões convencional e orientada a objetos, apresentadas nas Partes 3 e 4 deste livro, porque:

1. Faz uso explícito de controle estatístico de qualidade.
2. Verifica a especificação de projeto usando uma prova de correção baseada em matemática.
3. Implementa técnicas de teste que têm alta probabilidade de descobrir erros de alto impacto.

Obviamente, a abordagem sala limpa aplica a maioria, se não todos, dos princípios e conceitos básicos de engenharia de software apresentados ao longo deste livro. Bons procedimentos de análise e projeto são essenciais se for para obter alta qualidade. Mas a engenharia sala limpa diverge das práticas convencionais de software pela desenfatização (alguns diriam eliminação) do papel do teste de unidade e depuração e pela redução drástica (ou eliminação) do volume de testes realizados pelo desenvolvedor do software.[1]

No desenvolvimento convencional de software, os erros são aceitos como fatos da vida. Como erros são considerados inevitáveis, cada módulo de programa deve ser submetido a teste de unidade (para descobrir erros) e depois depurado (para remover erros). Quando o software finalmente é liberado, o uso no campo descobre ainda mais defeitos e começa outro ciclo de teste e depuração. O retrabalho associado com essas atividades é dispendioso e demorado. Pior do que isso, pode ser degenerativo — a correção de erro pode (inadvertidamente) levar à introdução de mais erros ainda.

> "Uma coisa curiosa sobre a vida: você se recusa a aceitar qualquer coisa, mas a melhor você sempre quer."
>
> **W. Somerset Maugham**

Na engenharia de software sala limpa, teste de unidade e depuração são substituídos por verificação de correção e teste baseado em estatística. Essas atividades acopladas à manutenção de registros necessários para o aperfeiçoamento contínuo tornam a abordagem sala limpa singular.

1 Uma equipe de teste independente realiza os testes.

29.2 Especificação Funcional

Os princípios operacionais apresentados no Capítulo 7 se aplicam, independentemente do método de análise escolhido. Dados, função e comportamento são modelados. Os modelos resultantes devem ser particionados (refinados) para fornecer cada vez mais detalhes. O objetivo global é ir de uma especificação que capta a essência de um problema para uma especificação que fornece substanciais detalhes de implementação.

A engenharia de software sala limpa obedece aos princípios operacionais de análise usando um método chamado de *especificação de estrutura de caixas*. Uma "caixa" encapsula o sistema (ou alguma parte do sistema) em algum nível de detalhe. Por um processo de refinamento passo a passo, as caixas são refinadas em uma hierarquia em que cada caixa tem transparência referencial. Isto é, "o conteúdo de informação de cada caixa de especificação é suficiente para definir seu refinamento, sem depender da implementação de qualquer outra caixa" [LIN94]. Isso permite ao analista a partição hierárquica de um sistema, indo da representação essencial, no topo, para o detalhe específico de implementação, na parte inferior. Três tipos de caixas são usados:

? *Como é obtido o refinamento como parte de uma especificação de estrutura de caixas?*

Caixa-preta. A caixa-preta especifica o comportamento de um sistema ou parte de um sistema. O sistema (ou parte) responde a estímulos específicos (eventos) aplicando um conjunto de regras de transição que mapeiam o estímulo em uma resposta.

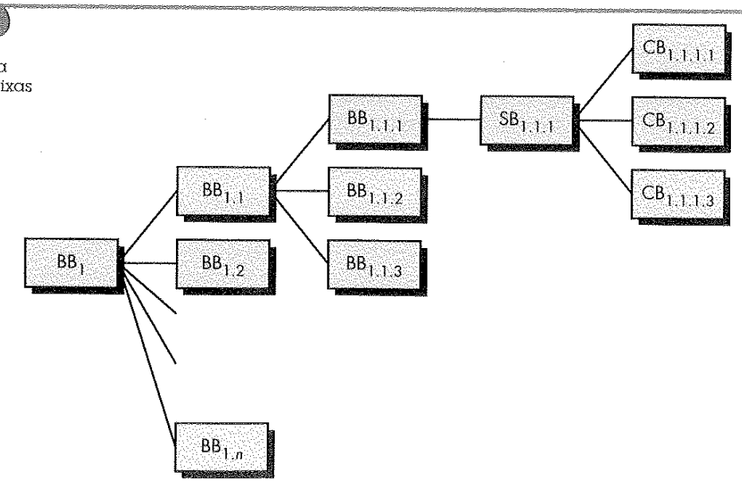
Caixa de estado. A caixa de estado encapsula dados de estado e serviços (operações) de um modo análogo aos objetos. Nessa visão da especificação, são representadas as entradas (estímulos) e as saídas (respostas) para a caixa de estado. A caixa de estado também representa o "histórico de estímulo" da caixa-preta; isto é, os dados encapsulados na caixa de estado que precisam ser retidos entre as transições implícitas.

Caixa-clara. As funções de transição implícitas na caixa de estados são definidas na caixa-clara. Isto é, uma caixa-clara contém o projeto procedimental da caixa de estado.

A Figura 29.2 ilustra a abordagem de refinamento usando a especificação de estrutura de caixas. Uma caixa-preta (*black box*-BB_1) define as respostas para um conjunto completo de estímulos. BB_1 pode ser refinado em um conjunto de caixas-pretas, $BB_{1.1}$ até BB_{1n}, cada uma delas cuida de uma classe de comportamento. O refinamento continua até que uma classe de comportamento coerente é identificada (por exemplo, $BB_{1.1.1}$). Então, uma caixa de estado (*statebox*, $SB_{1.1.1}$) é definida para a caixa-preta ($BB_{1.1.1}$). Neste caso, $SB_{1.1.1}$ contém todos os dados e serviços necessários para implementar o comportamento definido por $BB_{1.1.1}$. Finalmente, $SB_{1.1.1}$ é refinado em caixas-claras (*clear boxes*, $CB_{1.1.1.n}$) e são especificados detalhes de projeto procedimental.

FIGURA 29.2

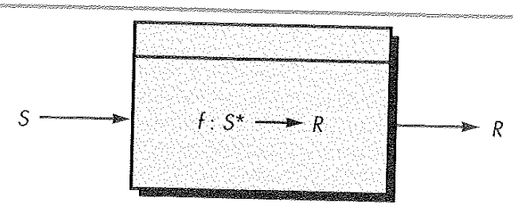
Refinamento da estrutura de caixas



Document shared on www.docsity.com

FIGURA 29.3

Uma especificação caixa-preta



PONTO CHAVE

O refinamento e a verificação de correção da estrutura de caixas ocorrem simultaneamente.

À medida que ocorre cada um desses passos de refinamento, a verificação de correção também ocorre. As especificações caixa de estado são verificadas para garantir a concordância de cada uma com o comportamento definido pela especificação caixa-preta pai. Analogamente, as especificações caixa-clara são verificadas contra a caixa de estado pai.

Deve-se notar que podem ser usados métodos de especificação baseados em linguagens tais como OCL ou Z (Capítulo 28) em conjunto com a abordagem de especificação de estrutura de caixas. O único requisito é que cada nível de especificação possa ser formalmente verificado.

29.2.1 Especificação Caixa-preta

Uma especificação *caixa-preta* descreve uma abstração, estímulos e respostas usando a notação mostrada na Figura 29.3 [MIL88]. A função f é aplicada a uma seqüência, S^*, de entradas (estímulos), S, e as transforma em uma saída (resposta), R. Para componentes de software simples, f pode ser uma função matemática, mas, em geral, f é descrita usando linguagem natural (ou uma linguagem de especificação formal).

Muitos dos conceitos introduzidos para sistemas orientados a objetos são também aplicáveis à caixa-preta. As abstrações de dados e as operações que manipulam essas abstrações são encapsuladas pela caixa-preta. Como uma hierarquia de classes, a especificação caixa-preta pode exibir hierarquias de uso nas quais as caixas de nível baixo herdam as propriedades das caixas mais altas na estrutura em árvore.

29.2.2 Especificação Caixa de Estado

A *caixa de estado* é "uma generalização simples de uma máquina de estado" [MIL88]. Lembrando a discussão da modelagem de comportamento e de diagramas de transição de estados, no Capítulo 8, um estado é algum modo observável do comportamento de um sistema. À medida que o processamento ocorre, um sistema responde a eventos (estímulos) fazendo uma transição do estado corrente para algum novo estado. Enquanto uma transição é feita, uma ação pode ocorrer. A caixa de estado usa uma abstração de dados para determinar a transição para o próximo estado e a ação (resposta) que vai ocorrer como conseqüência da transição.

Observando a Figura 29.4, a caixa de estado incorpora uma caixa-preta. O estímulo, S, que entra na caixa-preta chega de alguma fonte externa e de um conjunto de estados internos do sistema, T. Mills [MIL88] fornece uma descrição matemática da função f da caixa-preta contida na caixa de estado:

$$g: S^* \times T^* \rightarrow R \times T$$

em que g é uma subfunção ligada a um estado específico t. Quando considerados coletivamente, o par estado–subfunção (t, g) define a função f da caixa-preta.

29.2.3 Especificação Caixa-clara

A especificação caixa-clara é um tanto semelhante ao projeto procedimental e à programação estruturada (Capítulo11). Na verdade, a subfunção g dentro da caixa de estado é substituída pelas construções de programação estruturada que implementam g.

Como exemplo, considere a caixa-clara mostrada na Figura 29.5. A caixa-preta, g, mostrada na Figura 29.4, é substituída por uma seqüência de construções que incorpora uma condicional. Essas, por sua vez, podem ser refinadas em caixas-claras de nível mais baixo à medida que o refinamento passo a passo prossegue.

FIGURA 29.4

Uma especificação caixa de estado

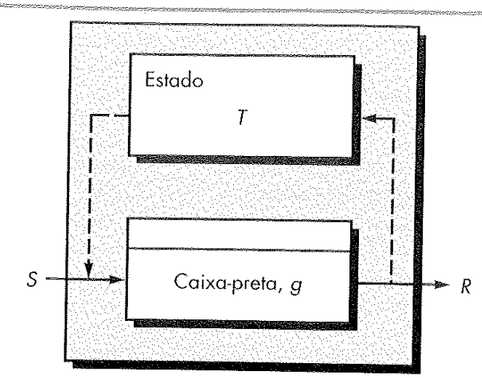
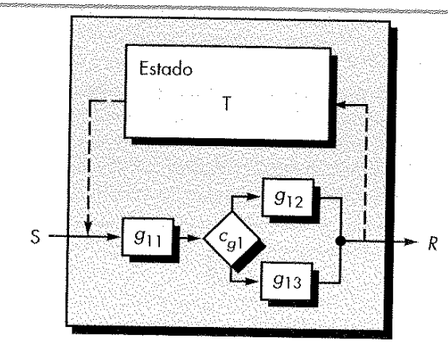


FIGURA 29.5

Uma especificação caixa-clara



É importante notar que a especificação procedimental descrita na hierarquia de caixas-claras pode ter sua correção provada. Esse tópico é considerado na seção seguinte.

29.3 PROJETO SALA LIMPA

A abordagem de projeto sala limpa utilizada na engenharia de software faz uso intenso da filosofia de programação estruturada. Neste caso, a programação estruturada é aplicada muito mais rigorosamente.

Funções básicas de processamento (descritas durante os primeiros refinamentos da especificação) são refinadas usando uma "expansão passo a passo de funções matemáticas em estruturas de conectivos lógicos [por exemplo, *se-então-senão*] e subfunções, em que a expansão [é] conduzida, até que todas as subfunções identificadas possam ser diretamente declaradas na linguagem de programação usada para implementação" [DYE92].

A abordagem de programação estruturada pode ser usada efetivamente para refinar uma função, mas e quanto ao projeto de dados? Aqui, alguns conceitos fundamentais de projeto (Capítulos 5 e 9) entram em jogo. Dados de programação são encapsulados como um conjunto de abstrações atendidas por subfunções. Os conceitos de encapsulamento de dados, ocultamento da informação e verificação de dados são usados para criar o projeto de dados.

29.3.1 Refinamento e Verificação de Projeto

Cada especificação caixa-clara representa o projeto de um procedimento (subfunção) necessário para conseguir uma transição de caixa de estado. Com a caixa-clara, são usadas as construções

Document shared on www.docsity.com

da programação estruturada e o refinamento passo a passo, como é ilustrado na Figura 29.6. Uma função de programa, *f*, é refinada em uma seqüência de subfunções *g* e *h*. Estas, por sua vez, são refinadas por construções condicionais (*se-então-senão* e *faça-enquanto*). Refinamento posterior ilustra o contínuo refinamento lógico.

? Que condições são aplicadas para provar a correção de construções estruturadas?

Em cada nível de refinamento, a equipe sala limpa[2] realiza uma verificação formal de correção. Para conseguir isso, um conjunto genérico de *condições de correção* é acoplado às construções de programação estruturada. Se uma função *f* é expandida em uma seqüência *g* e *h*, a condição de correção para todas as entradas de *f* é

- *g* seguido por *h* faz *f*?

AVISO

Se você se limita apenas às construções estruturadas, à medida que cria um projeto procedimental, a prova da correção é direta. Se você viola as construções, as provas da correção são difíceis ou impossíveis.

Quando uma função *p* é refinada em uma condicional da forma, se <c> então *q* senão *r*, a condição de correção para toda entrada em *p* é

- Sempre que a condição <c> é verdadeira, *q* faz *p*; e sempre que <c> é falsa, *r* faz *p*?

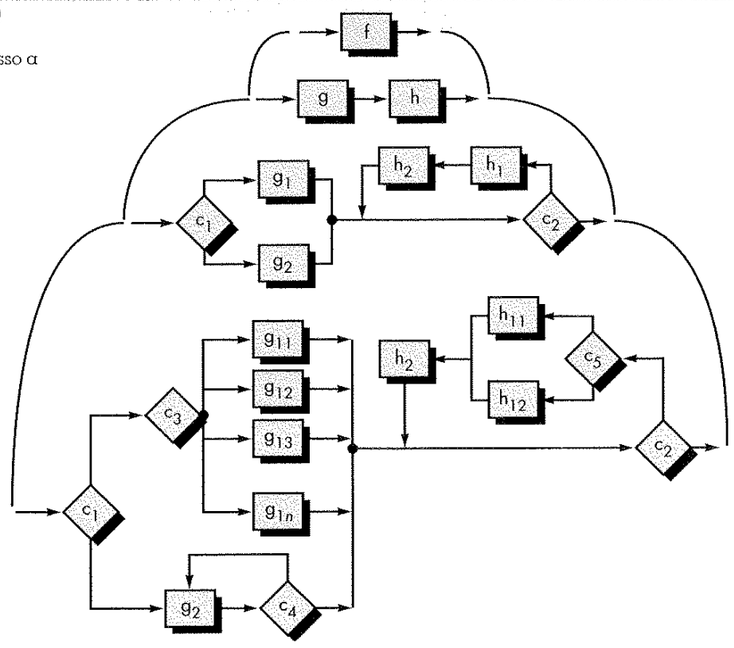
Quando a função *m* é refinada como um ciclo, as condições de correção para todas as entradas em *m* são

- O término está garantido?
- Sempre que <c> é verdadeiro, *n* seguido por *m* faz *m*; e sempre que <c> é falso, a saída do ciclo ainda faz *m*?

Cada vez que uma caixa-clara é refinada para o nível de detalhes seguinte, essas condições de correção são aplicadas.

FIGURA 29.6

Refinamento passo a passo



2 Como toda a equipe está envolvida no processo de verificação, é menos provável que um erro seja produzido na condução da própria verificação.

É importante notar que o uso das construções da programação estruturada restringe o número de testes de correção que precisam ser conduzidos. Uma única condição é verificada para seqüências; duas condições são testadas para *se-então-senão* e três condições são verificadas para ciclos.

Para ilustrar a verificação de correção para um projeto procedimental, usamos um exemplo simples, inicialmente introduzido por Linger, Mills e Witt [LIN79]. O objetivo é projetar e verificar um pequeno programa que calcula a parte inteira, *y*, da raiz quadrada de um determinado inteiro, *x*. O projeto procedimental é representado usando o fluxograma da Figura 29.7.

Para verificar a correção desse projeto, precisamos definir condições de entrada e de saída como apresentado na Figura 29.8. A condição de entrada menciona que *x* precisa ser maior ou igual a 0. A condição de saída exige que *x* permaneça imutável e assuma um valor dentro do intervalo mencionado na figura. Para provar que o projeto está correto, é necessário provar as condições *início, ciclo, cont, sim* e *saída*, mostradas na Figura 29.8, como sendo verdadeiras em todos os casos. Essas algumas vezes são chamadas de *subprovas*.

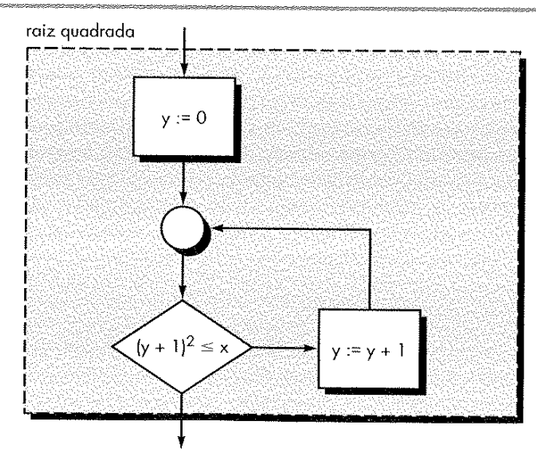
PONTO CHAVE

Para provar que um projeto está correto, você precisa primeiro identificar todas as condições e depois provar que cada uma assume um valor booleano adequado. Essas são as chamadas subprovas.

1. A condição *início* exige que [$x \geq 0$ e $y = 0$]. Com base nos requisitos do problema, a condição de entrada é considerada correta.[3] Assim, a primeira parte da condição *início*, $x \geq 0$, é satisfeita. Observando o fluxograma, a declaração imediatamente precedente da condição *início* faz $y = 0$. Conseqüentemente, a segunda parte da condição *início* também é satisfeita. Assim, *início* é verdadeira.
2. A condição *ciclo* pode ser encontrada em um dos dois modos: (1) diretamente a partir de *início* (nesse caso, a condição *ciclo* é satisfeita diretamente) ou (2) por meio do fluxo de controle que passa pela condição *cont*. Como a condição *cont* é idêntica à condição *ciclo*, *ciclo* é verdadeira independentemente do caminho do fluxo que leva a ela.
3. A condição *cont* é encontrada apenas depois de o valor de *y* ser incrementado de 1. Além disso, o caminho do fluxo de controle que leva a *cont* pode ser invocado apenas se a condição *sim* também é verdadeira. Conseqüentemente, se $(y + 1)^2 \leq x$, segue que $y^2 \leq x$. A condição *cont* é satisfeita.
4. A condição *sim* é testada na lógica condicional mostrada. Conseqüentemente, a condição *sim* deve ser verdadeira quando o fluxo de controle se move pelo caminho mostrado.
5. A condição *saída* exige primeiro que *x* permaneça inalterado. Um exame do projeto indica que *x* nunca aparece à esquerda de um operador de atribuição. Não há chamadas de função que usam *x*. Conseqüentemente, ele é inalterado. Como o teste da condicional $(y + 1)^2 \leq x$

FIGURA 29.7

Cálculo da parte inteira de uma raiz quadrada (LIN79)

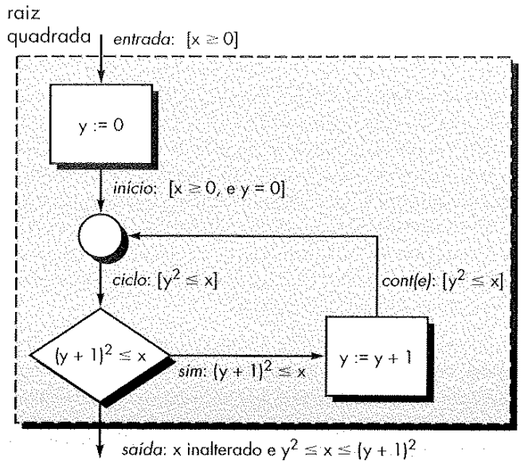


3 Um valor negativo para uma raiz quadrada não tem sentido neste contexto.

Document shared on www.docsity.com

FIGURA 29.8

Prova da correção do projeto (LIN79)



deve falhar para alcançar a condição *saída*, temos que $(y + 1)^2 \leq x$. Além disso, a condição de ciclo precisa ainda ser verdadeira (por exemplo, $y^2 \leq x$). Conseqüentemente, $(y + 1)^2 > x$ e $y^2 \leq x$ podem ser combinados para satisfazer a condição *saída*.

Além disso, devemos garantir que o ciclo termina. Um exame da condição do *ciclo* indica que como y é incrementado e $x \geq 0$, o ciclo deve certamente terminar.

Os cinco passos aqui mencionados são uma prova da correção do projeto do algoritmo mostrado na Figura 29.7. Agora, estamos certos de que o projeto vai de fato calcular a parte inteira de uma raiz quadrada.

Uma abordagem matemática mais rigorosa à verificação do projeto é possível. No entanto, uma discussão desse tópico foge do escopo deste livro. Leitores interessados devem ver [LIN79].

29.3.2 Vantagens da Verificação de Projeto[4]

A verificação rigorosa da correção de cada refinamento do projeto de caixa-clara tem várias vantagens distintas. Linger [LIN94] descreve-as da seguinte maneira:

? O que ganhamos fazendo provas de correção?

- *Reduz a verificação a um processo finito.* A maneira aninhada e seqüencial, na qual as estruturas de controle são organizadas em uma caixa-clara, define naturalmente uma hierarquia que revela as condições de correção que precisam ser verificadas. Um "axioma de substituição" [LIN79] nos permite substituir funções desejadas pelos refinamentos de sua estrutura de controle na hierarquia de subprovas. Por exemplo, a subprova para a função desejada f1, da Figura 29.9, exige a prova de que a composição de operações g1 e g2, com a função desejada f2, tenham o mesmo efeito nos dados que f1. Note que f2 substitui todos os detalhes do seu refinamento na prova. A substituição localiza o argumento da prova na estrutura de controle em mãos. De fato, ela permite ao engenheiro de software levar a efeito as provas em qualquer ordem.

PONTO CHAVE

Apesar do número extremamente grande de caminhos de execução em um programa, o número de passos para provar que o programa está correto é bastante pequeno.

- *É impossível superenfatizar o efeito positivo que a redução da verificação a um processo finito tem sobre a qualidade.* Apesar de todos os programas, com exceção dos mais triviais, exibirem um número essencialmente infinito de caminhos de execução, eles podem ser verificados em um número de passos finito.

4 Esta seção e as Figuras 29.7 a 29.9 foram adaptadas de [LIN94] e são usadas com permissão.

FIGURA 29.9

Um projeto com subprovas

```
                          Subprovas:
[f1]                      f1 = [FAÇA g1; g2; [f2] FIM] ?
 FAÇA
  g1
  g2
  [f2]                    f2 = [ENQUANTO p1 DO [f3] FIM] ?
   ENQUANTO
    p1
  FAÇA [f3]               f3 = [FAÇA g3; [f4]; g8 FIM] ?
    g3
    [f4]                  f4 = [SE p2; ENTÃO [f5] SENÃO [f6] FIM] ?
    SE
     p2
    ENTÃO [f5]            f5 = [FAÇA g4; g5 FIM] ?
     g4
     g5
    SENÃO [f6]            f6 = [FAÇA g6; g7 FIM] ?
     g6
     g7
    FIM
    g8
  FIM
FIM
```

- *Permite a equipes sala limpa verificarem cada linha do projeto e do código.* As equipes podem efetuar a verificação por meio de análise e discussão de grupo, com base no teorema da correção, e podem produzir provas escritas quando é requerida maior confiança em sistemas críticos com relação a vidas ou à missão desempenhada.
- *Resulta em um nível de defeito quase zero.* Durante uma revisão de equipe, a condição de correção de cada estrutura de controle é verificada na sua vez. Os membros da equipe precisam concordar que cada condição está correta, assim, um erro só é possível se todos os membros verificarem incorretamente uma condição. A exigência de concordância unânime, baseada na verificação individual, resulta em software que tem pouco ou nenhum defeito antes da primeira execução.
- *Aceita ampliação de escala.* Todo sistema de software, independentemente de quão grande seja, tem procedimentos de caixa-clara no topo dos níveis compostos de estruturas de seqüência, alternativa e iteração. Cada um deles tipicamente invoca um subsistema grande com milhares de linhas de código — e cada um desses subsistemas tem suas próprias funções e procedimentos no nível de topo. Assim, as condições de correção para essas estruturas de controle de alto nível são verificadas do mesmo modo que aquelas estruturas de nível mais baixo. A verificação nos níveis altos pode levar, e vai levar, mais tempo, mas não precisa de mais teoria.
- *Produz código melhor do que o teste de unidade.* O teste de unidade verifica os efeitos da execução apenas de caminhos selecionados entre os muitos caminhos possíveis. Baseando a verificação na teoria da função, a abordagem sala limpa pode verificar todo o efeito possível sobre todos os dados porque, apesar de um programa ter muitos caminhos de execução, tem apenas uma função. A verificação é também mais eficiente do que o teste de unidade. A maioria das condições de verificação pode ser examinada em uns poucos minutos, mas os testes de unidade levam tempo substancial para preparar, executar e examinar.

É importante notar que a verificação de projeto deve, em última análise, ser aplicada ao código-fonte propriamente dito. Neste contexto, é freqüentemente chamada de *verificação de correção*.

Document shared on www.docsity.com

29.4 Teste Sala Limpa

A estratégia e a tática do teste sala limpa são fundamentalmente diferentes das abordagens convencionais de teste. Os métodos convencionais derivam um conjunto de casos de teste para descobrir erros de projeto e de codificação. A meta do teste sala limpa é validar os requisitos do software, demonstrando que uma amostra estatística de casos de uso (Capítulo 7) foi executada de maneira bem-sucedida.

> "Qualidade não é um ato. É um hábito."
>
> Aristóteles

29.4.1 Teste Estatístico de Uso

AVISO

Mesmo se você decidir contra a abordagem de sala limpa, vale a pena considerar teste de uso estatístico como uma parte integral da sua estratégia de teste.

O usuário de um programa de computador raramente precisa entender os detalhes técnicos do projeto. O comportamento do programa visível ao usuário é determinado por entradas e eventos que são freqüentemente produzidos pelo usuário. Mas, em sistemas complexos, o espectro possível de entradas e eventos (por exemplo, os casos de uso) pode ser extremamente amplo. Que subconjunto de casos de uso vai verificar adequadamente o comportamento do programa? Essa é a primeira questão tratada pelo teste estatístico de uso.

O teste estatístico de uso "consiste em testar o software do modo que os usuários pretendem usá-lo" [LIN94]. Para conseguir isso, equipes de teste sala limpa (também chamadas de *equipes de certificação*) precisam determinar uma distribuição de probabilidade de uso do software. A especificação (caixa-preta) de cada incremento do software é analisada para definir um conjunto de estímulos (entradas ou eventos) que faz o software mudar seu comportamento. Com base nas entrevistas com usuários em potencial, na criação de cenários de uso e em uma compreensão geral do domínio de aplicação, uma probabilidade de uso é atribuída a cada estímulo.

Casos de testes são gerados para cada estímulo[5] de acordo com a distribuição de probabilidade de uso. Para ilustrar, considere o sistema de segurança *CasaSegura* discutido anteriormente neste livro. A engenharia de software sala limpa é usada para desenvolver um incremento de software que gere a interação do usuário com um teclado do sistema de segurança. Foram identificados cinco estímulos para esse incremento. A análise indica a distribuição percentual de probabilidade de cada estímulo. Para tornar a seleção de casos de teste mais fácil, essas probabilidades são mapeadas em intervalos numerados de 1 a 99 [LIN94] e ilustrados na seguinte tabela:

Estímulo de programa	Probabilidade	Intervalo
Armar/desarmar (AD)	50%	1-49
Ativar zona (ZS)	15%	50-63
Consulta (Q)	15%	64-78
Teste (T)	15%	79-94
Alarme de pânico	5%	95-99

Para gerar uma seqüência de casos de teste de uso, que segue a distribuição de probabilidades de uso, uma série de números aleatórios entre 1 e 99 é gerada. O número aleatório corresponde a um intervalo na distribuição de probabilidades precedente. Assim sendo, a seqüência dos casos de teste de uso é definida aleatoriamente, mas corresponde à probabilidade adequada de ocorrência do estímulo. Por exemplo, considere que as seguintes seqüências de números aleatórios são geradas:

13-94-22-24-45-56
81-19-31-69-45-9
38-21-52-84-86-4

5 Ferramentas automatizadas são usadas para conseguir isso. Para mais informação veja [DYE92].

Selecionando os estímulos adequados, com base no intervalo da distribuição mostrado na tabela, os seguintes casos de uso são derivados:

AD-T-AD-AD-AD-ZS
T-AD-AD-AD-Q-AD-AD
AD-AD-ZS-T-T-AD

A equipe de teste executa esses casos de uso e verifica o comportamento do software em função da especificação do sistema. O tempo dos testes é registrado, de modo que possam ser determinados intervalos de tempo. Usando esses intervalos de tempo, a equipe de certificação pode calcular o tempo médio de falha. Se uma longa seqüência de testes é conduzida sem falha, o tempo médio de falhas (MTTF) é baixo e a confiabilidade do software pode ser considerada alta.

29.4.2 Certificação

? Como certificamos um componente de software?

As técnicas de verificação e teste discutidas anteriormente neste capítulo levam a componentes de software (e incrementos completos) que podem ser certificados. No contexto da abordagem de engenharia de software sala limpa, a certificação implica que a confiabilidade (medida pelo tempo médio de falhas — MTTF) pode ser especificada para cada componente.

O impacto provável de componentes de software certificáveis vai muito além de um único projeto sala limpa. Componentes reusáveis de software podem ser armazenados junto com seus cenários de uso, estímulos de programa e distribuições de probabilidade. Cada componente teria uma confiabilidade certificada sob o cenário de uso e regime de teste descrito. Essa informação é de valor incalculável para outros que pretendam usar o componente.

A abordagem de certificação envolve cinco passos [WOH94]:

1. Cenários de uso devem ser criados.
2. Um perfil de uso é especificado.
3. Casos de testes são gerados a partir do perfil.
4. Testes são executados e dados de falhas são registrados e analisados.
5. A confiabilidade é calculada e certificada.

Os passos 1 a 4 já foram discutidos em seção anterior. Nesta seção, vamos nos concentrar na certificação da confiabilidade.

A certificação para a engenharia de software sala limpa exige a criação de três modelos [POO93]:

Modelo de amostragem. O teste de software executa m casos de testes aleatórios e é certificado se nenhuma falha ou um número de falhas especificado ocorre. O valor de m é derivado matematicamente para garantir que a confiabilidade desejada é atingida.

Modelo de componentes. Um sistema composto de n componentes deve ser certificado. O modelo de componentes permite ao analista determinar a probabilidade de o componente i falhar antes de ser completado.

Modelo de certificação. A confiabilidade total do sistema é projetada e certificada.

No final do teste estatístico de uso, a equipe de certificação tem a informação necessária para liberar um software que tem um MTTF certificado e calculado usando cada um desses modelos.

Uma discussão detalhada do cálculo dos modelos de amostragem, componentes e certificação foge do escopo deste livro. O leitor interessado deve ver [MUS87], [CUR86] e [POO93] para detalhes adicionais.

29.5 Resumo

A engenharia de software sala limpa é uma abordagem formal que pode levar ao desenvolvimento de software com uma qualidade extremamente alta. Usa especificação em estrutura de caixas (ou métodos formais) para a modelagem da análise e projeto, e enfatiza a verificação da correção,

Document shared on www.docsity.com

em vez do teste, como principal mecanismo para encontrar e remover erros. Teste estatístico de uso é aplicado para desenvolver a informação sobre a taxa de falhas necessária para certificar a confiabilidade do software entregue.

A abordagem sala limpa começa com modelos de análise e projeto que usam uma representação em estrutura de caixas. Uma "caixa" encapsula o sistema (ou algum tópico do sistema) em um nível de abstração específico. Caixas-pretas são usadas para representar o comportamento de um sistema externamente observável. Caixas de estado encapsulam dados e operações do estado. Uma caixa-clara é usada para modelar o projeto procedimental que está implícito nos dados e operações de uma caixa de estado.

A verificação de correção é aplicada quando o projeto da estrutura de caixas é completado. O projeto procedimental de um componente de software é particionado em uma série de funções. Para provar a correção das subfunções, são definidas condições de saída para cada subfunção e um conjunto de subprovas é aplicado. Se cada condição de saída é satisfeita, o projeto deve estar correto.

Uma vez completada a verificação de correção, começa o teste estatístico de uso. Diferentemente do teste convencional, a engenharia de software sala limpa não enfatiza o teste de unidade ou de integração. Em vez disso, o software é testado pela definição de um conjunto de cenários de uso, determinando a probabilidade de uso de cada cenário e definindo depois testes aleatórios que obedecem às probabilidades. Os registros de erros produzidos são combinados com os modelos de amostragem, de componentes e de certificação, para permitir o cálculo matemático da confiabilidade projetada para o componente de software.

A filosofia sala limpa é uma abordagem rigorosa de engenharia de software. É um modelo de processo de software que enfatiza a verificação matemática da correção e a certificação da confiabilidade do software. O referencial são taxas de falha extremamente baixas, que seriam difíceis ou impossíveis de alcançar usando métodos menos formais.

Referências Bibliográficas

[CUR86] Currittt, P. A., Dyer, M. e Mills, H. D., "Certifying the Reliability of Software", *IEEE Trans, Software Engineering*, v. SE-12, n. I, jan. 1994.

[DYE92] Dyer, M., *The Cleanroom Approach to Quality Software Development*, Wiley, 1992.

[HAU94] Hausler, P. A., Linger R. e Trammel, C., "Adopting Cleanroom Software Engineering with a Phased Approach", *IBM Systems Journal*, v. 33, n. I, jan. 1994, p. 89-109.

[HEN95] Henderson, J., "Why Isn't Cleanroom the Universal Software Development Methodology?", *Crosstalk*, v. 8, n. 5, maio 1995, p. 11-14.

[HEV93] Hevner, A. R. e Mills, H. D., "Box Structure Methods for System Development with Objects", *IBM Systems Journal*, v. 31, n. 2, fev. 1993, p. 232-251.

[LIN79] Linger, R. M., Mills H. D., e Witt, B. I., *Structured Programming: Theory and Practice*, Addison-Wesley, 1979.

[LIN88] Linger, R. M. e Mills, H. D., "A Case Study in Cleanroom Software Engineering: The IBM COBOL Structuring Facility", *Proc. COMPSAC 88*, Chicago, out. 1988.

[LIN94] Linger, R., "Cleanroom Process Model", *IEEE Software*, v. 11, n. 2, mar. 1994, p. 50-58.

[MIL87] Mills, H. D., Dyer, M. e Linger, R., "Cleanroom Software Engineering", *IEEE Software*, v. 4, n. 5, set. 1987, p. 19-24.

[MIL88] _____, "Stepwise Refinement and Verification in Box Structured Systems", *Computer*, v. 21, n. 6, jun. 1988, p. 23-35.

[MUS87] Musa, J. D., Iannino, A. e Okumoto, K., *Engineering and Managing Software with Reliability Measures*, McGraw-Hill, 1987.

[POO88] Poore, J. H. e Mills, H. D., "Bringing Software Under Statistical Quality Control", *Quality Progress*, nov. 1988, p. 52-55.

[POO93] _____, e Mutchler, D., "Planning and Certifying Software System Reliability", *IEEE Software*, v. 10, n. 1, jan. 1993, p. 88-99.

[WOH94] Wohlin, C. e Runeson, P., "Certification of Software Components", *IEEE Trans. Software Engineering*, v. SE-20, n. 6, jun. 1994, p. 494-499.

Problemas e Pontos a Considerar

29.1. Se tiver que escolher um aspecto da engenharia de software sala limpa que a torne radicalmente diferente das abordagens de engenharia de software convencional, qual seria ele?

29.2. Como um modelo de processo incremental e certificação trabalham juntos para produzir software de alta qualidade?

29.3. Desenvolva modelos de análise e projeto de "primeiro passo" para o sistema *CasaSegura* usando especificação de estrutura de caixas.

29.4. Desenvolva uma especificação em estrutura de caixas para uma parte do sistema SARB introduzido no Problema 8.10.

29.5. Um algoritmo para ordenação de bolha é definido da seguinte maneira:

```
procedimento de ordenação por bolha;
var i, f, inteiro;
início
 repita até que f = a[1]
        f: = a[1];
        para j:= 2 até n faça
            se a[j-1] > a[j] então início
            f: = a[j-1];
            a[j-1]:=a[j];
            a[j]:= f;
            fim
fim do rep
fim
```

Particione o projeto em subfunções e defina um conjunto em condições que lhe permitiriam provar que esse algoritmo é correto.

29.6. Documente uma prova de verificação de correção para a ordenação por bolha discutida no Problema 29.5.

29.7. Selecione um componente de programa que você projetou em outro contexto (ou atribuído a você pelo seu instrutor) e desenvolva uma prova de correção completa para ele.

29.8. Selecione um programa que você usa regularmente (por exemplo, um manipulador de *e-mail*, um processador de texto, um programa de planilha), e crie uma série de cenários de uso para o programa. Defina a probabilidade de uso de cada cenário e depois desenvolva uma tabela de estímulos de programa e de distribuição de probabilidades, semelhante àquela mostrada na Seção 29.4.1.

29.9. Para a tabela de estímulos de programa e distribuição de probabilidades desenvolvida no Problema 29.8, use um gerador de números aleatórios para desenvolver um conjunto de casos de teste para uso no teste estatístico de uso.

29.10. Descreva com suas próprias palavras o objetivo da certificação e o contexto da engenharia de software sala limpa.

Leituras e Fontes de Informação Adicionais

Prowell *et al.* (*Cleanroom Software Engineering: Technology and Process*, Addison-Wesley, 1999) fornecem um tratamento profundo de todos os aspectos importantes da abordagem sala limpa. Discussões úteis sobre tópicos sala limpa foram editadas por Poore e Trammell (*Cleanroom Software Engineering: A Reader*, Blackwell Publishing, 1996). Becker e Whittaker (*Cleanroom Software Engineering Practices*, Idea Group Publishing, 1996) apresentam uma excelente visão geral para aqueles que não estão familiarizados com as práticas sala limpa.

O *Cleanroom Pamphlet* (Software Technology Support Center, Hill AF Base, abril 1995) contém reproduções de vários artigos importantes. Linger [LIN94] produziu uma das melhores introduções ao assunto. O Data and Analysis Center for Software (DACS) (www.dacs.dtic.mil) fornece muitos artigos, guias e outras fontes de informação úteis sobre engenharia de software sala limpa.

Document shared on www.docsity.com

Linger e Trammell (*Cleanroom Software Engineering Reference Model*, SEI Technical Report CMU/SEI-96-TR-022, 1996) definiram um conjunto de 14 processos sala limpa e 20 produtos de trabalho que formam a base do SEI CMM para engenharia de software sala limpa (CMU/SEI-96-TR-023).

Michael Deck, da Cleanroom Software Engineering, preparou uma bibliografia sobre assuntos sala limpa. Muitas estão disponíveis em formato para serem baixadas.

Verificação do projeto por meio de prova de correção fica no âmago da abordagem sala limpa. Livros de Stavely (Toward Zero-Defect Software, Addison-Wesley, 1998), Baber (*Error-Free Software*, Wiley, 1991) e Schulmeyer (*Zero Defect Software*, McGraw-Hill, 1990) discutem prova de correção em considerável detalhe.

Uma ampla variedade de fontes de informação sobre engenharia de software sala limpa está disponível na Internet. Uma lista atualizada de referências da World Wide Web que são relevantes pode ser encontrada no site deste livro: **http://www.mhhe.com/pressman**.

ENGENHARIA DE SOFTWARE BASEADA EM COMPONENTES

CAPÍTULO 30

CONCEITOS-CHAVE

adaptação 669
ambiente de reuso 674
CBD 668
CBSE
  economia 675
  processo 665
classificação 673
engenharia de domínio ... 666
midleware 671
pontos estruturais 676
qualificação 669
tipos de componentes 664

O reuso, no contexto da engenharia de software, não é uma idéia velha nem nova. Os programadores têm usado idéias, abstrações e processos desde os primeiros dias da computação, mas as primeiras abordagens ao reuso eram as da prática atual. Hoje, sistemas complexos e de alta qualidade baseados em computador precisam ser construídos em períodos de tempo muito curtos e demandam uma abordagem mais organizada de reuso.

A engenharia de software baseada em componentes (Component-Based Software Engineering, CBSE) é um processo que enfatiza o projeto e a construção de sistemas baseados em computador usando "componentes" de software reusáveis. Clements [CLE95] descreve a CBSE do seguinte modo:

> [CBSE] está mudando o modo pelo qual grandes sistemas de software são desenvolvidos. [CBSE] incorpora a filosofia "comprar, em vez de construir", abraçada por Fred Brooks e outros. Do mesmo modo que as primeiras sub-rotinas liberaram o programador de pensar sobre detalhes, [CBSE] desloca a ênfase da programação de software para a composição de sistemas de software. A implementação deu lugar à integração como foco. Na sua fundamentação está a pressuposição de que há muitos pontos em comum, em vários sistemas grandes de software, para justificar o desenvolvimento de componentes reusáveis para explorar e satisfazer esses pontos em comum.

Mas surgem algumas questões. É possível construir sistemas complexos pela sua montagem a partir de um catálogo de componentes de software reusáveis? Isso pode ser realizado de um modo eficiente, em termos de custo e de tempo? Podem ser estabelecidos incentivos apropriados para encorajar os engenheiros de software a reusar, em vez de reinventar? A gerência aceita arcar com a despesa adicional associada com a criação de componentes de software

PANORAMA

O que é? Você compra um "sistema de entretenimento e o leva para casa. Cada componente foi projetado para se enquadrar em uma arquitetura de áudio-vídeo específica — as conexões são normalizadas e o protocolo de comunicação foi preestabelecido. A montagem é fácil, porque você não tem que construir o sistema a partir de centenas de partes discretas. A engenharia de software baseada em componentes (*Component Based Software Engineering*, CBSE) tenta conseguir a mesma coisa. Um conjunto de componentes de software normalizados, pré-construídos, é disponibilizado para se enquadrar em um estilo arquitetural específico para algum domínio de aplicação. Então, a aplicação é montada usando esses componentes, em vez de partes discretas de uma linguagem de programação convencional.

Quem faz? Engenheiros de software aplicam o processo CBSE.

Por que é importante? Leva apenas alguns minutos para montar o sistema de entretenimento residencial, porque os componentes são projetados para serem integrados com facilidade. Apesar de o software ser consideravelmente mais complexo, os sistemas baseados em componentes são mais fáceis de montar e, conseqüentemente, menos dispendiosos para construir do que sistemas confeccionados a partir de partes discretas. Além disso, a CBSE encoraja o uso de padrões arquiteturais previsíveis e infra-estrutura normalizada de software, levando conseqüentemente a um resultado de maior qualidade.

Quais são os passos? A CBSE abrange duas atividades paralelas de engenharia: engenharia de domínio e desenvolvimento baseada em componentes. A engenharia de domínio explora um domínio de aplicação com o objetivo específico de encontrar componentes funcionais, comportamentais e de dados, que são candidatos a reuso. Esses

Document shared on www.docsity.com

componentes são colocados em bibliotecas de reuso. O desenvolvimento baseado em componentes elicita os requisitos do cliente, seleciona um estilo arquitetural apropriado para atender aos objetivos do sistema a ser construído e depois (1) seleciona componentes potenciais para reuso, (2) qualifica os componentes para se certificar de que se encaixam adequadamente na arquitetura do sistema, (3) adapta os componentes se precisarem ser feitas modificações para integrá-los adequadamente e (4) integra os componentes para formar subsistemas e toda a aplicação. Além disso, componentes sob encomenda são trabalhados pela engenharia para cuidar dos ângulos do sistema que não podem ser implementados usando componentes existentes.

Qual o produto do trabalho? Software operacional, montado usando componentes de software existentes e de outros recém-desenvolvidos, é o resultado da CBSE.

Como tenho certeza de que fiz corretamente? Use as mesmas práticas de SQA que são aplicáveis em qualquer processo de engenharia de software — revisões técnicas formais avaliam os modelos de análise e projeto, revisões especializadas consideram os tópicos associados com os componentes adquiridos, teste é aplicado para descobrir erros no software recém-desenvolvido e nos componentes reusáveis que foram integrados na arquitetura.

reusáveis? Uma biblioteca de componentes, necessária para propiciar o reuso, pode ser criada de um modo que se torne acessível àqueles que dela necessitam? Componentes que já existem podem ser encontrados por aqueles que deles necessitam?

Mesmo hoje em dia, os engenheiros de software lutam com essas e outras questões sobre o reuso de componentes de software. Neste capítulo, enfocamos algumas das respostas.

30.1 Engenharia de Sistemas Baseada em Componentes

Veja na Web

Informação útil sobre CBSE de WebApps pode ser encontrada em www.cbd-hq.com.

Superficialmente, a CBSE parece bastante semelhante à engenharia de software convencional ou orientada a objetos. O processo começa quando uma equipe de software estabelece os requisitos para um sistema a ser construído usando técnicas de elicitação de requisitos convencionais (Capítulo 7). Um projeto arquitetural (Capítulo 10) é estabelecido, mas em vez de passar imediatamente a tarefas de projeto mais detalhadas, a equipe examina os requisitos para determinar que subconjunto é mais adequado à *composição* do que à construção. Isto é, a equipe formula as seguintes questões para cada requisito do sistema:

- Componentes comerciais prontos para uso* (*commercial off-the-shelf*, COTS) estão disponíveis para implementar o requisito?
- Componentes reusáveis, internamente desenvolvidos, estão disponíveis para implementar o requisito?
- As interfaces dos componentes disponíveis são compatíveis com a arquitetura do sistema a ser construído?

A equipe tenta modificar ou remover os requisitos do sistema que não podem ser implementados com COTS ou com componentes próprios[1]. Se os requisitos não puderem ser mudados ou descartados, os métodos de engenharia de software são aplicados para desenvolver aqueles componentes novos, que precisam ser trabalhados pela engenharia para satisfazer os requisitos. Mas para aqueles requisitos que podem ser atendidos com componentes disponíveis, um conjunto diferente de atividades de engenharia de software começa: qualificação, adaptação, composição e atualização. Cada uma dessas atividades de CBSE é discutida em mais detalhes na Seção 30.4.

Na primeira parte desta seção, o termo *componente* foi usado repetidamente, no entanto, um significado definitivo do termo é impreciso. Brown e Wallnau [BRO96] sugerem as seguintes possibilidades:

- *Componente* — uma parte não-trivial de um sistema, praticamente independente e substituível, que preenche uma função clara no contexto de uma arquitetura bem definida.

* N.R.T.: Este termo também é traduzido comumente por "componentes de prateleira".

1 A implicação é de que a organização ajuste seus requisitos de negócio ou produto, de modo que a implementação baseada em componentes possa ser realizada sem a necessidade de engenharia sob encomenda. Essa abordagem reduz o custo do sistema e melhora o tempo de colocação no mercado, mas nem sempre é possível.

- *Componente de software em execução* — um pacote dinamicamente constituído de um ou mais programas, geridos como uma unidade, ao qual se tem acesso por meio de interfaces documentadas, que podem ser descobertas durante a execução.
- *Componente de software* — uma unidade com dependências de contexto apenas explícita e contratualmente especificadas.
- *Componente de negócio* — implementação, em software, de um conceito de negócio ou processo de negócio "autônomo".

Além dessas descrições, os componentes de software podem também ser caracterizados com base no seu uso, no processo CBSE. Além de componentes COTS, o processo CBSE possibilita:

- *Componentes qualificados* — avaliados por engenheiros de software para garantir que não apenas a funcionalidade, mas também o desempenho, a confiabilidade, a usabilidade e outros fatores de qualidade (Capítulo 26) satisfazem os requisitos do sistema ou do produto a ser construído.
- *Componentes adaptados* — adaptados para modificar características não requeridas ou indesejáveis (também significando *mascarar* ou *empacotar*) [BRO96].
- *Componentes montados* — integrados no estilo arquitetural e interconectados com uma infra-estrutura adequada para permitir que os componentes sejam coordenados e geridos efetivamente.
- *Componentes atualizados* — para substituir o software existente à medida que novas versões de componentes se tornam disponíveis.

30.2 O Processo CBSE

O *processo* CBSE, no entanto, deve ser caracterizado de modo que não identifique apenas candidatos a componente, mas também qualifique a interface de cada componente, adapte componentes para remover discordâncias arquiteturais, monte componentes em um estilo arquitetural selecionado e os atualize, à medida que os requisitos do sistema se modificam [BRO96].

O modelo de processo para a engenharia de software baseada em componentes enfatiza caminhos paralelos, nos quais a engenharia de domínio (Seção 30.3) ocorre simultaneamente com o desenvolvimento baseado em componentes.

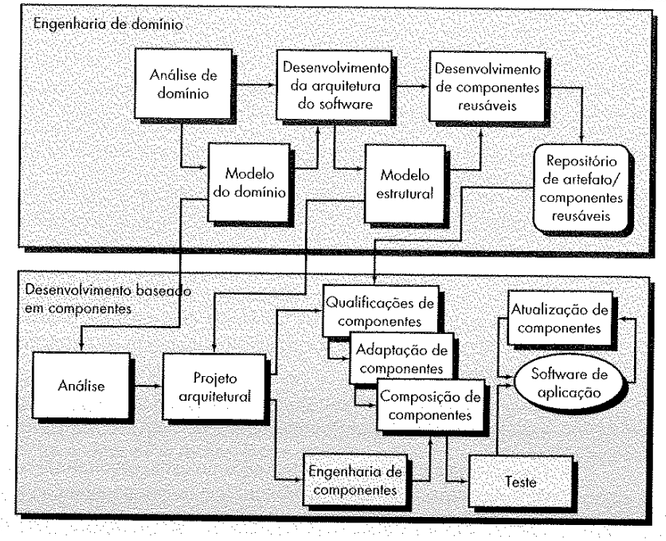
A Figura 30.1 ilustra um modelo típico de processo que acomoda explicitamente a CBSE [CHR95]. A *engenharia de domínio* cria um modelo do domínio de aplicação, que é usado como base para analisar os requisitos do usuário no curso da engenharia de software. Uma arquitetura de software genérica fornece a entrada para o projeto da aplicação. Finalmente, depois de terem sido adquiridos, selecionados de bibliotecas existentes, ou construídos (como parte da engenharia de domínio), os componentes reusáveis são colocados à disposição dos engenheiros de software durante o desenvolvimento baseado em componentes.

Os passos de *análise* e *projeto arquitetural* definidos como parte do desenvolvimento baseado em componentes (Figura 30.1) podem ser implementados dentro de um contexto do *paradigma de projeto abstrato* (*abstract design paradigm*, ADP) [DOG03]. Um ADP implica que o modelo global do software — representado como dados, função e comportamento (controle) — pode ser hierarquicamente decomposto. Quando a decomposição começa, o sistema é representado como uma coleção de frameworks arquiteturais, cada um composto de um ou mais padrões de projeto (Capítulo 10). Refinamento adicional identifica os componentes necessários para criar cada padrão de projeto. Em um contexto ideal, todos esses componentes deveriam ser adquiridos de um repositório (*as atividades de qualificação, adaptação e composição de componente se aplicam*). Quando componentes especializados são necessários, *engenharia de componente* é aplicada.

Document shared on www.docsity.com

FIGURA 30.1

Um modelo de processo que suporta CBSE



30.3 ENGENHARIA DE DOMÍNIO

O objetivo da *engenharia de domínio* é identificar, construir, catalogar e disseminar um conjunto de componentes de software que tem aplicabilidade a software existente e futuro, em um particular domínio de aplicação. A meta global é estabelecer mecanismos que permitam aos engenheiros de software compartilhar esses componentes — reusá-los — durante o trabalho com sistemas novos e existentes.

A engenharia de domínio inclui três principais atividades — análise, construção e disseminação.

> "Engenharia de domínio consiste em encontrar pontos comuns entre sistemas para identificar componentes que podem ser aplicados a muitos sistemas e identificar famílias de programas, que são posicionadas para tirar plena vantagem desses componentes."
>
> **Paul Clements**

AVISO

O processo de análise que discutimos nesta seção enfoca componentes reusáveis. No entanto, a completa análise de sistemas COTS (por exemplo, aplicações de e-commerce, aplicações de automação de força de venda) pode também ser uma parte da análise de domínio.

Pode-se alegar que "o reuso vai desaparecer, não por eliminação, mas por integração", do contexto da prática de engenharia de software [TRA95]. À medida que o reuso tem uma maior ênfase, alguns acreditam que a engenharia de domínio vai se tornar tão importante quanto a engenharia de software durante a próxima década.

30.3.1 O Processo de Análise de Domínio

A abordagem geral de análise de domínio é freqüentemente caracterizada no contexto da engenharia de software orientada a objetos. Os passos do processo são definidos como:

1. Defina o domínio a ser investigado.
2. Categorize os itens extraídos do domínio.
3. Colete uma amostra representativa das aplicações do domínio.
4. Analise cada aplicação da amostra e defina as classes de análise.
5. Desenvolva um modelo de análise para os objetos.

? Que componentes, identificados durante a análise de domínio, serão candidatos a reuso?

É importante notar que a análise de domínio é aplicável a qualquer paradigma de engenharia de software e pode ser aplicada tanto para o desenvolvimento convencional quanto para o orientado a objetos.

Apesar de os passos mencionados fornecerem um modelo útil para a análise de domínio, não fornecem diretrizes para decidir que componentes de software são candidatos a reuso. Hutchinson e Hindley [HUT88] sugerem o seguinte conjunto de questões pragmáticas como guia para identificar componentes de software reusáveis:

- A funcionalidade do componente é necessária em implementações futuras?
- Quão comum é a função do componente dentro do domínio?
- Há duplicação da função do componente dentro do domínio?
- O componente é dependente de hardware? Em caso afirmativo, o hardware permanece imutável entre implementações ou as especificidades do hardware podem ser removidas para outro componente?
- O projeto é suficientemente otimizado para a implementação seguinte?
- Podemos adicionar parâmetros a um componente não-reusável de modo que se torne reusável?
- O componente é reusável em várias implementações apenas com pequenas modificações?
- O reuso pela modificação é exeqüível?
- Um componente não-reusável pode ser decomposto para produzir componentes reusáveis?
- Quão válida é a decomposição de componente para reuso?

Para informação adicional sobre análise de domínio, veja [ATK01], [HEI01] e [PRI93].

30.3.2 Funções de Caracterização

Algumas vezes é difícil determinar se um componente possivelmente reusável é, de fato, aplicável a uma situação particular. Para estabelecer essa determinação, é necessário definir um conjunto de características do domínio compartilhadas por todo o software dentro do domínio. Uma característica do domínio define um atributo genérico de todos os produtos que existem dentro do domínio. Por exemplo, características genéricas poderiam incluir a importância da segurança/confiabilidade, a linguagem de programação, a simultaneidade no processamento e muitas outras.

Um conjunto de características de domínio para um componente reusável pode ser representado como $\{D_p\}$, em que cada item, D_{pi}, no conjunto representa uma característica específica do domínio. O valor atribuído a D_{pi} representa uma escala de ordenação que dá uma indicação da relevância da característica para o componente p. Uma escala típica [BAS94] poderia ser

1: Não relevante quanto ao reuso ser adequado.
2: Relevante apenas sob circunstâncias não usuais.
3: Relevante — o componente pode ser modificado de modo que possa ser usado, apesar das diferenças.
4: Claramente relevante, e se o novo software não tem essa característica, o reuso será ineficiente mas ainda pode ser possível.
5: Claramente relevante, e se o novo software não tem essa característica, o reuso será ineficaz e o reuso sem a característica não é recomendado.

Veja na Web

Informação útil sobre análise de domínio pode ser encontrada em www.sei.cmu.edu/str/descriptions/deda.html.

Quando um novo software, w, tem que ser construído no domínio da aplicação, um conjunto de características do domínio é criado para ele. É feita então uma comparação entre D_{pi} e D_{wi} para determinar se o componente p existente pode ser efetivamente reusado na aplicação w.

Mesmo quando o software a ser trabalhado pela engenharia existe claramente em um domínio de aplicação, os componentes reusáveis desse domínio devem ser analisados para determinar sua aplicabilidade. Em alguns casos (idealmente em número limitado), "reinventar a roda" pode ser ainda a escolha mais eficaz em termos de custo.

Document shared on www.docsity.com

30.3.3 Modelagem Estrutural e Pontos da Estrutura

Quando a análise de domínio é aplicada, o analista procura padrões repetidos nas aplicações que residem nesse domínio. A modelagem estrutural é uma abordagem de engenharia de domínio baseada em padrões que trabalha com a pressuposição de que todo o domínio de aplicação tem padrões repetidos (de função, de dados e de comportamento) que têm potencial de reuso.

Cada domínio de aplicação pode ser caracterizado por um modelo estrutural (por exemplo, sistemas de aviônica diferem bastante nas especificidades, mas todo o software moderno desse domínio tem o mesmo modelo estrutural). Conseqüentemente, o modelo estrutural é um estilo arquitetural (Capítulo 10) que pode e deve ser reusado pelas aplicações de um domínio.

McMahon [MCM95] descreve um *ponto de estrutura* como "uma construção distinta dentro de um modelo estrutural". Pontos de estrutura têm três características distintas:

? O que é um "ponto estrutural" e quais são suas características?

1. Um ponto de estrutura é uma abstração que deveria ter um número limitado de instâncias. Além disso, a abstração deve se repetir entre as aplicações do domínio. Caso contrário, o custo para verificar, documentar e disseminar o ponto de estrutura não pode ser justificado.
2. As regras que governam o uso do ponto de estrutura devem ser facilmente entendidas. Além disso, a interface com o ponto de estrutura deve ser relativamente simples.
3. O ponto de estrutura deveria implementar ocultamento da informação pelo isolamento de toda a complexidade contida no próprio ponto de estrutura. Isso reduz a complexidade perceptível do sistema global.

PONTO CHAVE

Um ponto de estrutura pode ser visto como um padrão de projeto que pode ser encontrado repetidamente em aplicações em um domínio específico.

A exemplo de pontos de estrutura como padrões arquiteturais para um sistema, considere o domínio de software para sistemas de alarme. Esse domínio pode abranger sistemas tão simples quanto o *CasaSegura* (discutido em capítulos anteriores) ou tão complexos quanto o sistema de alarme para um processo industrial. Em todos os casos, no entanto, é encontrado um conjunto de padrões estruturais previsíveis: uma *interface* que permite ao usuário interagir com o sistema, um *mecanismo de estabelecimento de limites* que permite ao usuário estabelecer limites nos parâmetros a ser medidos, um *mecanismo de gestão de sensores* que se comunica com todos os sensores de monitoramento, um *mecanismo de resposta* que reage a entradas fornecidas pelo sistema de gestão de sensores e um *mecanismo de controle* que permite ao usuário controlar a maneira pela qual o monitoramento é conduzido. Cada um desses pontos de estrutura é integrado em uma arquitetura do domínio.

É possível definir pontos de estrutura genéricos que transcendem alguns diferentes domínios de aplicação [STA94]:

- *Fachada de aplicação* — a IGU que inclui todos os menus, painéis e facilidades para edição de entradas e comandos.
- *Banco de dados* — repositório para todos os objetos relevantes ao domínio de aplicação.
- *Motor computacional* — os modelos numéricos e não-numéricos que manipulam os dados.
- *Recursos para relatórios* — função que produz saídas de todas as formas.
- *Editor de aplicações* — mecanismo para adaptar a aplicação às necessidades de usuários específicos.

Pontos de estrutura têm sido sugeridos como alternativa às linhas de código e pontos por função para estimativa do custo de software [MCM95]. Uma breve discussão de custos usando pontos de estrutura é apresentada na Seção 30.6.2.

30.4 Desenvolvimento Baseado em Componentes

O *desenvolvimento baseado em componentes* (*Component Based Development*, CBD) é uma atividade de CBSE que ocorre em paralelo à engenharia de domínio. Usando os métodos de análise e projeto arquitetural discutidos anteriormente neste livro, a equipe de software refina um estilo arquitetural adequado para o modelo de análise criado para a aplicação a ser construída[2].

Uma vez estabelecida a arquitetura, ela deve ser preenchida por componentes que (1) estão disponíveis em bibliotecas de reuso e/ou (2) são trabalhados por engenharia para satisfazer as necessidades do cliente. Assim, o fluxo de tarefas para o desenvolvimento baseado em componentes tem dois caminhos paralelos (Figura 30.1). Quando os componentes reusáveis estão disponíveis para possível integração na arquitetura, eles precisam ser qualificados e adaptados. Quando novos componentes são necessários, precisam ser construídos. Os componentes resultantes são então "compostos" (integrados) no gabarito da arquitetura e testados rigorosamente.

30.4.1 Qualificação, Adaptação e Composição de Componentes

Como já vimos, a engenharia de domínio fornece a biblioteca de componentes reusáveis que são necessários para a engenharia de software baseada em componentes. Alguns desses componentes reusáveis são desenvolvidos internamente, outros podem ser extraídos de aplicações existentes e outros ainda podem ser adquiridos de terceiros.

Infelizmente, a existência de componentes reusáveis não garante que esses possam ser integrados fácil ou efetivamente na arquitetura escolhida para uma nova aplicação. É por essa razão que uma seqüência de atividades de desenvolvimento baseado em componentes é aplicada quando um componente é proposto para uso.

? Que fatores são considerados durante a qualificação de componentes?

Qualificação de componentes. A qualificação de componentes garante que um componente candidato vai realizar a função necessária, ele "se encaixará" adequadamente no estilo arquitetônico especificado para o sistema, e exibirá as características de qualidade (por exemplo, desempenho, confiabilidade e usabilidade) necessárias para a aplicação.

A descrição da interface fornece informação útil sobre a operação e o uso de um componente de software, mas não fornece toda a informação necessária para determinar se o componente proposto pode de fato ser reusado efetivamente em uma nova aplicação. Entre os muitos fatores considerados durante a qualificação de componentes estão [BRO96]: a interface de programação da aplicação (*application programming interface*, API); ferramentas de desenvolvimento e integração exigidas pelo componente; requisitos de execução, incluindo o uso de recursos (por exemplo, memória ou armazenamento), tempo ou velocidade e protocolo de rede; exigências de serviço, incluindo interfaces do sistema operacional e apoio por outros componentes, características de segurança, incluindo controles de acesso e protocolo de autenticação; pressupostos de projeto embutidos, incluindo o uso de algoritmos específicos numéricos ou não-numéricos e tratamento de exceções.

AVISO

Além de avaliar se o custo de adaptação para reuso é justificável, a equipe de software também avalia se a obtenção da funcionalidade e desempenho requeridos pode ser feita de modo efetivo em termos de custo.

Cada um desses fatores é relativamente fácil de avaliar quando componentes reusáveis, que foram desenvolvidos internamente, são propostos. No entanto, é muito mais difícil determinar o funcionamento interno de COTS ou componentes de terceiros, porque a única informação disponível pode ser a própria especificação da interface.

Adaptação de componentes. Em condições ideais, a engenharia de domínio cria uma biblioteca de componentes que podem ser facilmente integrados na arquitetura de uma aplicação. A implicação de "fácil integração" é que (1) métodos consistentes de gestão de recursos tenham sido implementados para todos os componentes da biblioteca, (2) atividades comuns, como gestão de dados, existam para todos os componentes e (3) as interfaces dentro da arquitetura e com o ambiente externo tenham sido implementadas de maneira consistente.

Na realidade, mesmo depois de um componente ter sido qualificado para uso dentro da arquitetura de uma aplicação, ele pode conflitar com uma ou mais das áreas mencionadas. Para evitar esses conflitos, uma técnica de adaptação chamada de *empacotamento de componente* (*component wrapping*) [BRO96] é freqüentemente usada. Quando uma equipe de software tem pleno acesso ao projeto e código interno de um componente (não é freqüentemente o caso quando componentes COTS são usados), o *empacotamento caixa-branca* (*white box wrapping*) é aplicado. Como o seu

2 Deve-se notar que o estilo arquitetural é freqüentemente influenciado pelo modelo estrutural genérico, criado durante a engenharia do domínio (veja a Figura 30.1).

Document shared on www.docsity.com

correspondente no teste de software (Capítulo 14), o empacotamento caixa-branca examina os detalhes de processamento interno do componente e faz modificações no nível de código para remover qualquer conflito. O *empacotamento caixa-cinza* (*gray box wrapping*) é aplicado quando a biblioteca de componentes fornece uma linguagem para extensão de componentes ou API que permite a remoção ou o mascaramento dos conflitos. O *empacotamento caixa-preta* (*black box wrapping*) exige a introdução de pré e pós-processamento na interface do componente para remover ou mascarar conflitos. A equipe de software deve determinar se o esforço necessário para empacotar adequadamente um componente é justificado ou se um componente sob encomenda (projetado para eliminar os conflitos encontrados) deve, em vez disso, ser construído.

Composição de componentes. A tarefa de *composição de componentes* monta componentes qualificados, adaptados e construídos para compor a arquitetura estabelecida para uma aplicação. Para conseguir isso uma infra-estrutura deve ser estabelecida para aglutinar os componentes em um sistema em operação. A infra-estrutura (usualmente uma biblioteca de componentes especializados) fornece um modelo para a coordenação de componentes e serviços específicos, que permite aos componentes se coordenarem entre si e realizarem tarefas comuns.

Entre os muitos mecanismos para a criação de uma infra-estrutura efetiva está um conjunto de quatro "ingredientes arquiteturais" [ADL95], que devem estar presentes para obter a composição do componente:

? Que ingredientes são necessários para conseguir composição de componentes?

Modelo de intercâmbio de dados. Mecanismos que permitem que usuários e aplicações interajam e transfiram dados (por exemplo, arrastar e soltar, cortar e colar) devem ser definidos para todos os componentes reusáveis. Os mecanismos de intercâmbio de dados permitem não apenas a transferência de dados homem/software e componente/componente, mas também a transferência entre recursos do sistema (por exemplo, arrastar um arquivo para o ícone de uma impressora, para saída).

Automação. Diversas ferramentas, macros e *scripts* devem ser implementados para facilitar a interação entre componentes reusáveis.

Armazenamento estruturado. Dados heterogêneos (por exemplo, dados gráficos, voz/vídeo, texto e dados numéricos) contidos em um "documento composto" devem ser organizados e oferecer acesso como uma única estrutura de dados, em vez de como uma coleção de arquivos separados. "Dados estruturados mantêm um índice descritivo de estruturas aninhadas pelas quais as aplicações podem navegar livremente para localizar, criar ou editar conteúdos de dados individuais, como desejado pelo usuário final" [ADL95].

Modelo de objetos subjacente. O modelo de objetos garante que componentes desenvolvidos em diferentes linguagens de programação, que residem em diferentes plataformas, possam ser interoperacionais. Isto é, os objetos devem ser capazes de se comunicar por meio de uma rede. Para conseguir isso, o modelo de objetos define um padrão para a interoperabilidade de componentes.

Veja na Web
As informações mais recentes sobre CORBA podem ser obtidas em www.omg.org.

Como o impacto potencial de reuso e CBSE na indústria de software é enorme, várias companhias importantes e consórcios da indústria propuseram normas para componente de software:

OMG/CORBA. O Grupo de Gestão de Objetos (*Object Management Group*, OMG) publicou uma arquitetura comum para negociação entre objetos (*common object request broker architecture*, OMG/CORBA). Um negociador de solicitação entre objetos (*object request broker*, ORB) fornece diversos serviços para permitir que componentes reusáveis (objetos) se comuniquem com outros componentes, independentemente de sua localização no sistema.

Veja na Web
As informações mais recentes sobre COM podem ser obtidas em www.microsoft.com/COM.

Microsoft COM. A Microsoft desenvolveu um modelo de objeto de componentes (*component object model*, COM) que fornece uma especificação para usar componentes produzidos por diversos fornecedores em uma única aplicação, rodando sob o sistema operacional Windows. COM contém dois elementos: interfaces COM (implementadas como objetos COM) e um conjunto de mecanismos para registrar e passar mensagens entre interfaces COM.

Veja na Web
A informação mais recente sobre Javabeans pode ser obtida em java.sun.com/products/javabeans/docs/.

Sun JavaBeans Components. O sistema de componentes JavaBeans é uma infra-estrutura CBSE portátil, independente de plataforma, desenvolvida usando a linguagem de programação Java. O sistema de componentes JavaBeans inclui um conjunto de ferramentas, chamado de *Bean Development Kit* (BDK) que permite aos desenvolvedores (1) analisar como os (componentes) Beans existentes funcionam, (2) personalizar seu comportamento e aparência, (3) estabelecer mecanismos para coordenação e comunicação, (4) desenvolver Beans sob medida para uso em uma aplicação específica e (5) testar e avaliar o comportamento do Bean.

Quais dessas normas vão dominar a indústria? Não há resposta fácil no momento. Apesar de vários desenvolvedores terem adotado uma das normas, é provável que grandes organizações de software possam decidir usar todas as três, dependendo das categorias das aplicações e plataformas escolhidas.

INFO

Arquitetura de Negociador de Solicitação entre Objetos

Sistemas cliente/servidor são implementados usando componentes de software (objetos) que devem ser capazes de interagir uns com os outros dentro de uma única máquina (ou cliente ou servidor) ou através da rede. Um *negociador de solicitação entre objetos* (*object request broker*, ORB) é um "middleware" que permite que um objeto residente em um cliente envie uma mensagem para um método que está encapsulado em um objeto residente em um servidor. Em essência, o ORB intercepta a mensagem e manipula todas as atividades de comunicação e coordenação necessárias para encontrar o objeto para o qual a mensagem foi endereçada, invoca seu método, passa os dados adequados para o objeto e transfere os dados resultantes de volta para o objeto que gerou a mensagem inicialmente.

CORBA, COM e JavaBeans implementam uma filosofia de negociador de solicitação entre objetos. Nesta moldura, CORBA será usado para ilustrar o middleware ORB.

A estrutura básica de uma arquitetura CORBA é ilustrada na Figura 30.2. Quando CORBA é implementada em um sistema cliente/servidor, objetos tanto do cliente quanto do servidor são definidos usando uma *linguagem de descrição de interface* (*interface description language*, IDL), uma linguagem declarativa que permite a um engenheiro de software definir objetos, atributos, métodos, e as mensagens necessárias para invocá-los. Para acomodar uma solicitação de um método residente no servidor por um objeto residente no cliente, módulos pseudocontrolados IDL cliente e servidor são criados. Os pseudocontrolados fornecem o roteador (*gateway*) pelo qual solicitações de objetos através do sistema cliente/servidor são acomodadas.

Como solicitações de objetos pela rede ocorrem em tempo de execução, um mecanismo para armazenar a descrição do objeto deve ser estabelecido de modo que a informação pertinente sobre o objeto e sua localização estejam disponíveis quando necessário. O repositório de interface consegue isso.

Quando uma aplicação cliente precisa invocar um método contido em um objeto em outro lugar do sistema, CORBA usa invocação dinâmica para (1) obter informação pertinente sobre o método desejado do repositório da interface, (2) criar uma estrutura de dados com parâmetros a ser passados para o objeto, (3) criar uma solicitação do objeto, e (4) invocar a solicitação. A solicitação é então passada para o núcleo do ORB — uma parte específica da implementação do sistema operacional da rede que gerencia solicitações — e a solicitação é atendida.

A solicitação é passada através do núcleo e é processada pelo servidor. No local do servidor, um adaptador de objeto armazena informação da classe e do objeto em um repositório de interface residente no servidor, aceita e gerencia as solicitações que chegam do cliente, e executa uma variedade de outras funções de gestão de objeto. No servidor, pseudocontrolados IDL que são análogos aos definidos na máquina cliente são usados como interface da implementação do objeto real que reside no local do servidor.

30.4.2 Engenharia de Componentes

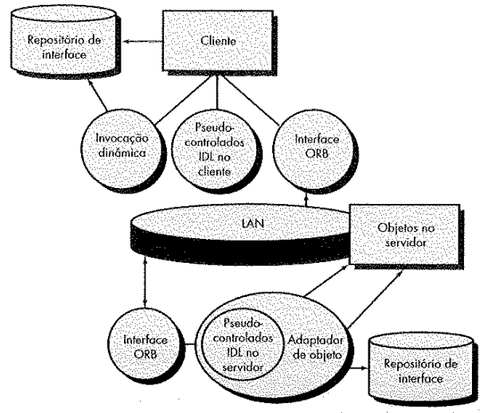
Como mencionamos anteriormente neste capítulo, o processo CBSE estimula o uso de componentes de software existentes. No entanto, há situações em que os componentes devem ser submetidos a engenharia, isto é, novos componentes de software devem ser desenvolvidos e integrados com componentes existentes COTS e internos. Como esses novos componentes tornam-se membros da biblioteca interna de componentes reusáveis, devem ser construídos para reuso.

Não há nada mágico a respeito da criação de componentes de software que podem ser reusados. Conceitos de projeto tais como abstração, ocultamento, independência funcional, refinamento e programação estruturada, junto com métodos de orientação a objetos, teste, SQA e métodos de

Document shared on www.docsity.com

FIGURA 30.2

A arquitetura CORBA básica



verificação de correção, todos contribuem para a criação de componentes de software reusáveis[3]. Nesta seção, não vamos rever esses tópicos. Em vez disso, consideramos tópicos específicos de reuso, que são complementares a sólidas práticas de engenharia de software.

30.4.3 Análise e Projeto para Reuso

O modelo de análise é avaliado para determinar os elementos do modelo que indicam componentes reusáveis existentes. O problema é extrair informação do modelo de requisitos com o objetivo de conseguir "satisfazer a especificação". Se satisfazer a especificação leva a componentes que atendem às necessidades da aplicação corrente, o projetista pode extrair esses componentes de uma biblioteca de reuso (repositório) e usá-los no projeto dos novos sistemas. Se componentes de projeto não podem ser encontrados, o engenheiro de software deve aplicar métodos de projeto convencional ou OO para criá-los. É nesse ponto — quando o projetista começa a criar um novo componente — que o *projeto para reuso* (*design for reuse*, DFR) deve ser considerado.

AVISO

DFR pode ser bastante difícil quando componentes precisam ser interfaceados ou integrados com sistemas legados ou com múltiplos sistemas, cuja arquitetura e protocolos de interface são inconsistentes.

Como já mencionamos, DFR exige que o engenheiro de software aplique sólidos conceitos e princípios de projeto de software (Capítulo 9). Mas as características do domínio de aplicação também precisam ser consideradas. Binder [BIN93] sugere alguns temas importantes[4] que formam a base do projeto para reuso:

Dados normalizados. O domínio da aplicação deve ser investigado e as estruturas de dados globais normalizadas (por exemplo, estruturas de arquivos ou um completo banco de dados) devem ser identificadas. Todos os componentes de projeto podem então ser caracterizados para fazer uso dessas estruturas de dados normalizados.

Protocolos de interface normalizados. Três níveis de protocolo de interface devem ser estabelecidos: a natureza de interfaces intramodulares, o projeto de interfaces externas técnicas (não humanas) e a interface homem/computador.

Gabaritos de programa. O modelo de estrutura (Seção 30.3.3) pode servir como gabarito para o projeto arquitetural de um novo programa.

Uma vez estabelecidos dados, interfaces e gabaritos de programa normalizados, o projetista tem um arcabouço no qual pode criar o projeto. Novos componentes que se adaptam a esse arcabouço têm uma alta probabilidade de reuso subseqüente.

3 Para aprender mais sobre esses tópicos, veja as Partes 2 e 5 deste livro.
4 Em geral, as preparações DFR devem ser realizadas como parte da engenharia de domínio (Seção 30.3).

30.5 Classificação e Recuperação de Componentes

Considere uma biblioteca universitária. Dezenas de milhares de livros, periódicos e outros recursos de informação estão disponíveis para uso. Mas para obter acesso a esses recursos, precisa ser desenvolvido um esquema de categorização. Para navegar por esse grande volume de informação, os bibliotecários definiram um esquema de classificação que inclui um código de classificação da biblioteca do Congresso Americano, palavras-chave, nomes de autor e outras entradas de indexação. Todas permitem ao usuário final encontrar o recurso desejado, rápida e facilmente.

Agora, considere um grande repositório de componentes. Dezenas de milhares de componentes de software reusáveis residem nele. Porém, como o engenheiro de software encontra aquele de que precisa? Para responder a essa questão, surgiu outra: Como podemos descrever componentes de software em termos classificáveis não-ambíguos? Essas são questões difíceis e até agora nenhuma resposta definitiva foi apresentada. Nesta seção exploramos os rumos atuais que permitirão aos futuros engenheiros de software navegar pelas bibliotecas de reuso.

30.5.1 Descrição de Componentes Reusáveis

Um componente de software reusável pode ser descrito de muitos modos, mas uma descrição ideal abrange o que Tracz [TRA90] chamou de *modelo 3C* — conceito, conteúdo e contexto.

O *conceito* de um componente de software é "uma descrição do que o componente faz" [WHI95]. A interface do componente é totalmente descrita e a semântica — representada no contexto de pré e pós-condições — é identificada. O conceito deve comunicar o objetivo do componente.

O *conteúdo* de um componente descreve como o conceito é realizado. Em essência, o conteúdo é uma informação que está oculta de usuários casuais e precisa ser conhecida apenas por aqueles que pretendem modificar ou testar o componente.

O *contexto* coloca um componente de software reusável no seu domínio de aplicabilidade. Isto é, especificando características conceituais operacionais e de implementação, o contexto permite ao engenheiro de software encontrar o componente adequado para satisfazer os requisitos da aplicação.

Para serem úteis em um local pragmático, o conceito, o conteúdo e o contexto precisam ser traduzidos em um esquema concreto de especificação. Dezenas de trabalhos e artigos foram escritos sobre esquemas de classificação para componentes de software reusáveis (por exemplo, [LUC01] e [WHI95] contêm uma extensa bibliografia). Os métodos propostos podem ser caracterizados em três áreas principais: métodos de biblioteca e ciência da informação, métodos de inteligência artificial e sistemas de hipertexto. A grande maioria dos trabalhos feitos até hoje sugere o uso de métodos de biblioteconomia para a classificação de componentes.

A Figura 30.3 apresenta uma taxonomia de métodos de indexação de biblioteconomia. *Vocabulários de indexação controlados* limitam os termos ou a sintaxe que podem ser usados para classificar um objeto (componente). *Vocabulários de indexação não-controlados* não colocam restrições na natureza da descrição. A maioria dos esquemas de classificação para componentes de software enquadra-se em três categorias:

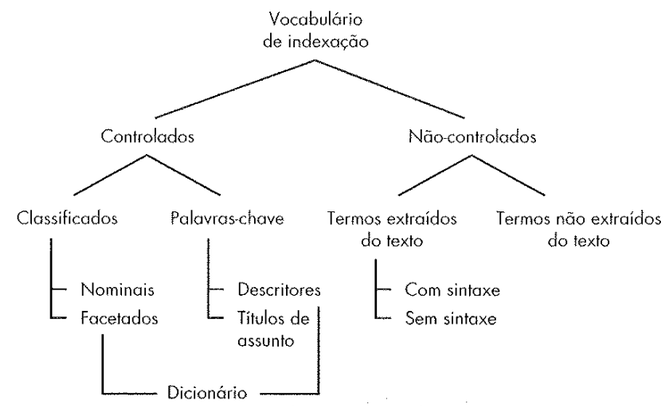
Classificação enumerada. Componentes são descritos por uma estrutura hierárquica em que as classes e os vários níveis de subclasses de componentes de software são definidos. A estrutura hierárquica de um esquema de classificação enumerado torna-o fácil de entender e usar. No entanto, antes que uma hierarquia possa ser construída, a engenharia de domínio precisa ser conduzida para que conhecimento suficiente das entradas adequadas na hierarquia se torne disponível.

Classificação facetada. Uma área de domínio é analisada e um conjunto de características descritivas básicas é identificado. Essas características, chamadas de *facetas*, são então ordenadas por importância e conectadas a um componente. Uma faceta pode descrever a função que o componente realiza, os dados que são manipulados, o contexto no qual são aplicados ou qualquer outra característica. O conjunto de facetas que descrevem o componente é chamado de *descritor facetado*. Geralmente a descrição facetada é limitada a não mais do que sete ou oito facetas.

Document shared on www.docsity.com

FIGURA 30.3

Uma taxonomia para métodos de indexação (FRA94)



Classificação de valores de atributos. Um conjunto de atributos é definido para todos os componentes em uma área de domínio. São então atribuídos valores a esses atributos, de modo muito análogo ao da classificação facetada. De fato, a classificação de valores de atributo é semelhante à classificação facetada com as seguintes exceções: (1) o número de atributos que pode ser usado não é limitado; (2) não é atribuída prioridade aos atributos e (3) a função de dicionário não é usada.

Com base no estudo empírico de cada uma dessas técnicas de classificação, Frakes e Pole [FRA94] indicam que não há uma técnica claramente "melhor" e que "nenhum método foi além de moderadamente bem, na eficiência das buscas...". Parece ser necessário realizar mais trabalho para o desenvolvimento de esquemas de classificação eficazes para bibliotecas de reuso.

30.5.2 O Ambiente de Reuso

O reuso de componentes de software deve ser apoiado por um ambiente que inclua os seguintes elementos:

- Um banco de dados de componentes capaz de guardar componentes de software e a informação de classificação necessária para recuperá-los.
- Um sistema de gestão de biblioteca que forneça acesso ao banco de dados.
- Um sistema de recuperação de componentes de software (por exemplo, um negociador de solicitação entre objetos) que permita a uma aplicação cliente recuperar componentes e serviços do servidor da biblioteca.
- Ferramentas CBSE que dão suporte à integração de componentes reusados a um novo projeto ou implementação.

Cada uma dessas funções interage com ou é incorporada aos limites de uma biblioteca de reuso.

Veja na Web

Uma abrangente coleção de recursos sobre CBSE pode ser encontrada em http://www.cbd-hq.com.

A biblioteca de reuso é o elemento de um repositório de software maior (Capítulo 27) e fornece facilidades para a guarda de componentes de software e uma grande variedade de artefatos reusáveis (por exemplo, especificações, projetos, padrões, frameworks, fragmentos de código, casos de teste, guias de usuário). A biblioteca inclui um banco de dados e as ferramentas que são necessárias para consultar o banco de dados e recuperar componentes dele. Um esquema de classificação de componentes (Seção 30.5.1) serve de base para as consultas à biblioteca.

As consultas são freqüentemente caracterizadas usando o elemento contexto do modelo 3C, descrito anteriormente nesta seção. Se uma consulta inicial produz uma lista volumosa de candidatos a componente, a consulta é refinada para diminuir a lista. É extraída então informação de

conceito e conteúdo (depois que os candidatos a componentes são encontrados) para assistir o desenvolvedor na seleção do componente adequado.

Uma discussão detalhada da estrutura de bibliotecas de reuso e das ferramentas que fazem a sua gestão é deixada a fontes dedicadas ao assunto. O leitor interessado deve ver [FIS00] e [LIN95] para informação adicional.

FERRAMENTAS DE SOFTWARE

Desenvolvimento Baseado em Componente

Objetivo: Apoiar modelagem, projeto, revisão e integração de componentes de software como parte de um sistema maior.

Mecânica: A mecânica das ferramentas varia. Em geral, ferramentas CBD apóiam em uma ou mais das seguintes capacidades: especificação e modelagem da arquitetura de software; navegação e seleção de componentes de software disponíveis, integração de componentes.

Ferramentas Representativas[5]

ComponentSource (*www.componentsource.com*) fornece uma ampla variedade de componentes (e ferramentas) de software COTS, apoiada em vários padrões de componente diferentes.

Component Manager, desenvolvida por Flashline (*www.flashline.com*), "é uma aplicação que permite, promove e mede reuso de componentes de software".

Select Component Factory, desenvolvida por Select Business Solutions (*www.selectbs.com/products*), "é um conjunto integrado de produtos para projeto de software, revisão de projeto, gestão de serviço/componente, gestão de requisitos e geração de código".

Software Through Pictures-ACD, distribuída por Aonix (www.aonix.com), permite modelagem abrangente usando UML para a arquitetura dirigida por modelo da OMG — uma abordagem aberta para CBSE, neutra em termos de fornecedor.

30.6 QUESTÕES ECONÔMICAS RELACIONADAS À CBSE

Veja na Web

Uma variedade de artigos fornecendo diretrizes para sistemas baseados em CBD e COTS pode ser encontrada em www.sei.cmu.edu.

A engenharia de software baseada em componentes tem um apelo intuitivo. Teoricamente deve fornecer vantagens em qualidade e cumprimento de prazos a uma organização de software. E essas deverão se traduzir em economias de custo. Mas existem dados concretos que apóiem nossa intuição?

Para responder a essa pergunta precisamos entender primeiro o que realmente pode ser reusado em um contexto de engenharia de software e quais são os custos realmente associados ao reuso. Como conseqüência, é possível desenvolver uma análise custo/benefício para reuso de componentes.

30.6.1 Impacto na Qualidade, Produtividade e Custo

Evidências de casos de estudo da indústria, em número considerável (por exemplo, [ALL02], [HEN95], [MCM95]), indicam que substanciais benefícios do negócio podem ser derivados de um reuso de software agressivo. A qualidade do produto, a produtividade de desenvolvimento e o custo global são melhorados.

Qualidade. Em condições ideais, um componente de software desenvolvido para reuso estaria verificado (ver Capítulo 29) e não conteria defeitos. Na realidade, a verificação formal não é realizada rotineiramente e defeitos podem ocorrer, e de fato ocorrem. No entanto, com cada reuso são encontrados e eliminados os defeitos e, como resultado, a qualidade do componente melhora. Ao longo do tempo, o componente torna-se virtualmente livre de defeitos.

Em um estudo conduzido na Hewlett Packard, Lim [LIM94] relata que a taxa de defeitos para código reusado é 0,9 defeito por KLOC, enquanto a taxa para software produzido a partir do zero é

5 As ferramentas mencionadas aqui não representam uma recomendação, mas, em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

Document shared on www.docsity.com

4,1 defeitos por KLOC. Para uma aplicação composta por 68% de código reusado, a taxa de defeitos foi de 2,0 defeitos por KLOC — uma melhora de 51% da taxa esperada, se a aplicação tivesse sido desenvolvida sem reuso. Henry e Faller [HEN95] relatam uma melhora de 35% na qualidade. Apesar de relatos anedóticos sobre percentual de melhora de qualidade se espalharem, é justo dizer que o reuso fornece um benefício significativo em termos de qualidade e confiabilidade para o software produzido.

Produtividade. Quando são aplicados componentes reusáveis ao longo do processo de software, é gasto menos tempo criando planos, modelos, documentos, código e dados necessários para produzir um sistema em condições de ser entregue. Daí decorre que é entregue ao cliente o mesmo nível de funcionalidade, com menos esforço de entrada. Assim sendo, a produtividade é aperfeiçoada. Apesar de os relatos de melhora percentual da produtividade serem notoriamente difíceis de interpretar[6], parece que 30% a 50% de reuso pode resultar em 25% a 40% de melhora de produtividade.

AVISO

O custo para desenvolver um componente reusável é freqüentemente maior que o custo para desenvolver um componente específico para uma aplicação. Certifique-se de que haverá necessidade para o componente reusável no futuro. É aí que o lucro é realizado.

Custo. A economia líquida de custo para reuso é estimada pela projeção do custo do projeto, se tivesse que ser desenvolvido a partir do zero, C_s, subtraindo depois a soma dos custos associados com o reuso, C_r, e os custos reais do software na hora de ser entregue, C_a.

C_s pode ser determinado pela aplicação de uma ou mais técnicas de estimação discutidas no Capítulo 23. Os custos associados com reuso, C_r, incluem [CHR95]: análise e modelagem de domínio, desenvolvimento da arquitetura do domínio, documentação ampliada para facilitar o reuso, suporte e aperfeiçoamento dos componentes de reuso, direitos autorais e licenças para componentes adquiridos externamente, criação ou aquisição e operação de um repositório de reuso, treinamento de pessoal em projeto e construção para reuso. Apesar de os custos associados com a análise de domínio (Seção 30.3) e com a operação do repositório de reuso poderem ser substanciais, muitos dos outros custos aqui mencionados referem-se a itens que fazem parte da boa prática de engenharia de software sendo ou não o reuso uma prioridade.

30.6.2 Análise de Custos Usando Pontos Estruturais

Na Seção 30.3.3, definimos um ponto estrutural como um padrão arquitetural que ocorre freqüentemente ao longo de um domínio de aplicação particular. Um projetista de software (ou engenheiro de sistemas) pode desenvolver uma arquitetura para uma nova aplicação, sistema ou produto definindo uma arquitetura de domínio e depois preenchendo-a com pontos estruturais. Ou esses pontos estruturais são componentes individuais reusáveis ou são pacotes de componentes reusáveis.

Apesar de os pontos estruturais serem reusáveis, sua qualificação, adaptação, integração e manutenção, em termos de custos, são significativos. Antes de prosseguir com o reuso, o gerente de projeto precisa entender os custos associados com o uso de pontos estruturais.

Como todos os pontos estruturais (e componentes reusáveis em geral) têm um histórico, podem ser coletados dados de custo para cada um. Em uma instalação ideal, a qualificação, adaptação, integração e manutenção, em termos de custos associados com cada componente em uma biblioteca de reuso, são mantidas para cada instância de uso. Esses dados podem então ser analisados para desenvolver os custos projetados para a próxima instância de reuso.

Considere, como exemplo, uma nova aplicação *X*, que requer 60% do código novo e o reuso de três pontos estruturais, SP_1, SP_2 e SP_3. Cada um desses componentes reusáveis foi usado em algumas outras aplicações e os custos médios de qualificação, adaptação, integração e manutenção estão disponíveis.

Para estimar o esforço necessário para entregar *X*, deve ser determinado:

$$\text{esforço global} = E_{novo} + E_{qual} + E_{adapt} + E_{int}$$

em que

6 Muitas circunstâncias atenuantes (por exemplo, domínio de aplicação, complexidade do problema, estrutura e tamanho da equipe, duração do projeto, tecnologia aplicada) podem ter profundo impacto na produtividade de uma equipe de projeto.

E_{novo}, = esforço necessário para projetar e construir novos componentes de software (usando as técnicas descritas no Capítulo 23).

E_{qual}, = esforço necessário para qualificar SP_1, SP_2 e SP_3.

E_{adapt}, = esforço necessário para adaptar SP_1, SP_2 e SP_3.

E_{int}, = esforço necessário para integrar SP_1, SP_2 e SP_3.

O esforço necessário para qualificar, adaptar e integrar SP_1, SP_2 e SP_3 é determinado tirando vantagem dos dados históricos coletados para qualificação, adaptação e integração de componentes reusáveis em outras aplicações.

30.7 Resumo

A engenharia de software baseada em componentes oferece benefícios inerentes em qualidade de software, produtividade dos desenvolvedores e custos globais do sistema. No entanto, muitos obstáculos precisam ser ultrapassados antes que o modelo de processo de CBSE seja amplamente usado pela indústria.

Além dos componentes de software, uma variedade de outros artefatos reusáveis pode ser adquirida por um engenheiro de software. Incluindo representações técnicas do software (por exemplo, especificações, modelos arquiteturais, projetos), documentos, padrões, frameworks, dados de teste e mesmo tarefas relacionadas a processo (por exemplo, técnicas de inspeção).

O processo CBSE abrange dois subprocessos simultâneos — engenharia de domínio e desenvolvimento baseado em componentes. O objetivo da engenharia de domínio é identificar, construir, catalogar e disseminar um conjunto de componentes de software em um domínio de aplicação particular. O desenvolvimento baseado em componentes qualifica, adapta e integra depois esses componentes para uso em um novo sistema. Além disso, o desenvolvimento baseado em componentes faz a engenharia de novos componentes, que são baseados nos requisitos do cliente, para um novo sistema.

Técnicas de análise e projeto para componentes reusáveis se apóiam nos mesmos princípios e conceitos, que são parte da boa prática de engenharia de software. Componentes reusáveis devem ser projetados em um ambiente que estabelece estruturas de dados normalizadas, bem como protocolos de interface e arquiteturas de programa para cada domínio de aplicação.

A engenharia de software baseada em componentes usa um modelo de intercâmbio de dados, ferramentas, armazenagem estruturada e um modelo de objetos subjacente para construir aplicações. O modelo de objetos geralmente segue uma ou mais normas de componentes (por exemplo, OMG/CORBA), que definem o modo pelo qual uma aplicação pode ter acesso a objetos reusáveis. Esquemas de classificação permitem ao desenvolvedor encontrar e recuperar componentes reusáveis, e seguir um modelo que identifica conceito, conteúdo e contexto. Classificação enumerada, classificação facetada e classificação por valor de atributos são representativas de muitos esquemas de classificação de componentes.

A economia do reuso de software é abordada em uma única questão: O custo efetivo para construir é menor do que para reusar? Em geral, a resposta é sim, mas um projetista de software deve considerar os custos significativos associados com a qualificação, a adaptação e a integração de componentes reusáveis.

Referências Bibliográficas

[ADL95] Adler, R. M., "Emerging Standards for Component Software", *Computer*, v. 28, n. 3, mar. 1995, p. 68-77.

[ALL02] Allen, P., "CBD Survey: The State of the Practice", *The Cutter Edge*, mar., 2002, disponível em http://www.cutter.com/research/2002/edge020305.html.

[ATK01] Atkinson, C. *et al.*, *Component-Based Product Line Engineering with UML*, Addison-Wesley, 2001.

[BAS94] Basili, V. R., Briand, L. C. e Thomas, W. M., "Domain Analysis for the Reuse of Software Development Experiences",

Document shared on www.docsity.com

Proc. of the 19th Annual Software Engineering Workshop, NASA/GSFC, Greenbelt, MD, dez. 1994.

[BIN93] Binder, R., "Design for Reuse Is for Real", *American Programmer*, v. 6, n. 8, ago. 1993, p. 30-37.

[BRO96] Brown, A. W. e Wallnau, K. C., "Engineering of Component Based Systems", *Component-Based Software Engineering*, IEEE Computer Society Press, 1996, p. 7-15.

[CHR95] Christensen, S. R., "Software Reuse Initiatives at Lockheed", *CrossTalk*, v. 8, n. 5, maio 1995, p. 26-31.

[CLE95] Clements, P. C., "From Subroutines to Subsystems: Component Based Software Development", *American Programmer*, v. 8, n. 11, nov. 1995.

[DOG03] Dogru, A. e Tanik, M., "A Process Model for Component-Oriented Software Engineering", *IEEE Software*, v. 20, n. 2, mar./abr. 2003, p. 34-41.

[FIS00] Fischer, B., "Specification-Based Browsing of Software Component Libraries", *J. Automated Software Engineering*, v. 7, n. 2, 2000, p. 179-200, disponível em http://asc.arc.nasa.gov/people/fischer/papers/ase-00.html.

[FRA94] Frakes, W. B. e Pole, T. P. "An Empirical Study of Representation Methods for Reusable Software Components", *IEEE Trans. Software Engineering*, v. SE-20, n. 8, ago. 1994, p. 617-630.

[HEI01] Heineman, G. e Councill, W. (eds.), *Component-Based Software Engineering*, Addison-Wesley, 2001.

[HEN95] Henry, E. e Faller, B., "Large Scale Industrial Reuse to Reduce Cost and Cycle Time", *IEEE Software*, set. 1995, p. 47-53.

[HUT88] Hutchinson, J. W. e Hindley, P. G., "A Preliminary Study of Large Scale Software Reuse", *Software Engineering Journal*, v. 3, n. 5, 1988, p. 208-212.

[LIA93] Liao, H. e Wang, F., "Software Reuse Based on a Large Object-Oriented Library", *ACM Software Engineering Notes*, v. 18, n. 1, jan. 1993, p. 74-80.

[LIM94] Lim, W. C., "Effects of Reuse on Quality, Productivity, and Economics", *IEEE Software*, set. 1994, p. 23-30.

[LIN95] Linthicum, D. S., "Component Development (a Special *Feature*)", *Application Development Trends*, jun. 1995, p. 57-78.

[LUC01] deLucena, Jr., V., "Facet-Based Classification Scheme for Industrial Software Components", 2001, disponível em http://research.microsoft.com/ users/cszypers/events/WCOP2001/Lucena.pdf.

[MCM95] McMahon, P. E., "Pattern-Based Architecture: Bridging Software Reuse and Cost Management", *Crosstalk*, v. 8, n. 3, mar. 1995, p. 10-16.

[ORF96] Orfali, R., Harkey, D. e Edwards, J., *The Essential Distributed Objects Survival Guide*, Wiley, 1996.

[PRI93] Prieto-Diaz, R., "Issues and Experiences in Software Reuse", *American Programmer*, v. 6, n. 8, ago. 1993, p. 10-18.

[POL94] Pollak, W. e Rissman, M., "Structural Models and Patterned Architectures", *Computer*, v. 27, n. 8, ago. 1994, p. 67-68.

[STA94] Staringer, W., "Constructing Applications from Reusable Components", *IEEE Software*, set. 1994, p. 61-68.

[TRA90] Tracz, W., "Where Does Reuse Start?", *Proc. Realities of Reuse Workshop*, Syracuse University CASE Center, jan. 1990.

[TRA95] _____, "Third International Conference on Software Reuse — Summary", *ACM Software Engineering Notes*, v. 20, n. 2, abr. 1995, p. 21-22.

[WHI95] Whittle, B., "Models and Languages for Component Description and Reuse", *ACM Software Engineering Notes*, v. 20, n. 2, abr. 1995, p. 76-89.

[YOU98] Yourdon, E. (ed.), "Distributed Objects", *Cutter IT Journal*, v. 11, n. 12, dez. 1998.

Problemas e Pontos a Considerar

30.1. Um dos principais obstáculos para o reuso é fazer os desenvolvedores de software considerarem o reuso de componentes existentes, em vez de reinventar novos componentes (apesar de tudo, é agradável construir as coisas!). Sugira três ou quatro modos diferentes pelos quais uma organização de software pode fornecer incentivos ao reuso para os engenheiros de software. Que tecnologias devem ser consideradas para apoiar o esforço de reuso?

30.2. Apesar de os componentes de software serem o "artefato" obviamente mais reusável, muitas outras entidades produzidas como parte da engenharia de software podem ser reusadas. Considere planos de projeto e estimativas de custo. Como podem ser reusados e qual o benefício conseguido?

30.3. Faça um pouco de pesquisa em engenharia de domínio e enriqueça o modelo de processo delineado na Figura 30.1. Identifique as tarefas necessárias para a análise de domínio e de desenvolvimento da arquitetura do software.

30.4. Em que as funções de caracterização para o domínio de aplicação e os esquemas de classificação de componentes se parecem? Em que diferem?

30.5. Desenvolva um conjunto de características de domínio para sistemas de informação que sejam relevantes ao processamento de dados sobre um estudante universitário.

30.6. Desenvolva um conjunto de características de domínio que sejam relevantes para o software de processamento de textos/publicação.

30.7. Desenvolva um modelo estrutural simples para um domínio de aplicação que lhe seja atribuído por seu instrutor ou um com o qual você está familiarizado.

30.8. O que é um ponto estrutural?

30.9. Obtenha informação sobre o padrão CORBA, COM ou JavaBeans mais recente, e prepare um trabalho de três a cinco páginas que discuta seus pontos principais. Informe-se sobre uma ferramenta de negociação de solicitações entre objetos e ilustre como a ferramenta segue a norma.

30.10. Desenvolva uma classificação enumerada para um domínio de aplicação que lhe seja atribuído pelo seu instrutor ou um com o qual você está familiarizado.

30.11. Desenvolva um esquema de classificação facetada para um domínio de aplicação que lhe seja atribuído pelo seu instrutor ou um com o qual você está familiarizado.

30.12. Pesquise a literatura para adquirir dados recentes sobre qualidade e produtividade que dão suporte ao uso de CBSE. Apresente-os para sua classe.

Leituras e Fontes de Informação Adicionais

Muitos livros sobre desenvolvimento baseado em componentes e reuso de componentes foram publicados nos anos recentes. Heineman e Councill [HEI01], Brown (*Large Scale Component-Based Development*, Prentice-Hall, 2000), Allen (*Realizing e-Business with Components*, Addison-Wesley, 2000), Herzum e Sims (*Business Component Factory*, Wiley, 1999) e Allen, Frost e Yourdon (*Component-Based Development for Enterprise Systems: Applying the Select Perspective*, Cambridge University Press, 1998) abrangem todos os tópicos importantes do processo de CBSE. Apperly e seus colegas (*Serviceand Component-Based Development*, Addison-Wesley, 2003), Atkinson [ATK01] e Cheesman e Daniels (*UML Components*, Addison-Wesley, 2000) discutem CBSE com ênfase em UML.

Leach (*Software Reuse: Methods, Models, and Costs*, McGraw-Hill, 1997) fornece uma análise detalhada de questões de custo associadas com CBSE e reuso. Poulin (*Measuring Software Reuse: Principles, Practices, and Economic Models*, Addison-Wesley, 1996) sugere vários métodos quantitativos para avaliar o benefício do reuso de software.

Dezenas de livros descrevendo as normas da indústria baseadas em componentes foram publicadas em anos recentes. Tratam do funcionamento interno das normas em si, mas também consideram muitos tópicos importantes de CBSE. Uma amostra das três normas discutidas neste capítulo é apresentada a seguir:

CORBA

Bolton, F., *Pure CORBA*, Sams Publishing, 2001.

Doss, G. M., *CORBA Networking with Java*, Wordware Publishing, 1999.

Hoque, R., *CORBA for Real Programmers*, Academic Press/Morgan Kaufmann, 1999.

Siegel, J., *CORBA Fundamentals and Programming*, Wiley, 1999.

Slama, D., Garbis, J. e Russell, P., *Enterprise CORBA*, Prentice-Hall, 1999.

COM

Box, D., Brown, K., Ewald, T., e Sells, C., *Effective COM: 50 Ways to Improve Your COM- and MTS-Based Applications*, Addison-Wesley, 1999.

Gordon, A., *The COM and COM Programming Primer*, Prentice-Hall, 2000.

Kirtland, M., *Designing Component-Based Applications*, Microsoft Press, 1999.

Tapadiya, P., *COM Programming*, Prentice-Hall, 2000.

Document shared on www.docsity.com

Muitas organizações aplicam uma combinação de normas de componentes. Livros de Geraghty e seus colegas (*COM-CORBA Interoperability*, Prentice-Hall, 1999), Pritchard (*COM and CORBA Side by Side: Architectures, Strategies, and Implementations*, Addison-Wesley, 1999) e Rosen e seus colegas (*Integrating CORBA and COM Applications*, Wiley, 1999) consideram as questões associadas com o uso de ambas, CORBA e COM, como fundamento para o desenvolvimento baseado em componentes.

JavaBeans

Asbury, S. e Weiner, S. R., *Developing Java Enterprise Applications*, Wiley, 1999.

Anderson, G. e Anderson, P., *Enterprise Javabeans Component Architecture*, Prentice-Hall, 2002.

Monson-Haefel, R., *Enterprise JavaBeans, 3ª ed.*, O´Reilly & Associates, 1999.

Roman, E., *et al.*, *Mastering Enterprise JavaBeans*, 2ª ed, Wiley, 2001.

Uma ampla variedade de fontes de informação sobre engenharia de software baseada em componentes está disponível na Internet. Uma lista atualizada de referências que são relevantes pode ser encontrada no site deste livro: **http://www.mhhe.com/pressman**.

REENGENHARIA

CAPÍTULO 31

CONCEITOS-CHAVE

análise de inventário 686
arquiteturas C/S 694
arquiteturas OO 694
economia 696
engenharia avante 693
engenharia reversa 688
estruturas de dados 692
manutenção 684
modelo de processo de BPR 683
processo de reengenharia 685
reestruturação 691

Num artigo pioneiro, escrito para a *Harvard Business Review*, Michael Hammer [HAM90] lançou as fundações de uma revolução no modo de pensar gerencial a respeito de processos do negócio e computação:

> Já é hora de parar de pavimentar trilhas de gado. Em vez de embutir processos desatualizados em silício e software, deveríamos descartá-los e começar de novo. Deveríamos "reengenheirar" os nossos negócios: usar o poder da moderna tecnologia da informação para reprojetar radicalmente nossos processos de negócio a fim de conseguir aperfeiçoamentos drásticos em seu desempenho.
>
> Toda empresa opera sob muitas regras desarticuladas... A reengenharia procura romper com as antigas regras sobre a condução e a organização do negócio.

Como em todas as revoluções, a chamada para a luta de Hammer resultou em modificações tanto positivas quanto negativas. Durante os anos de 1990, algumas empresas fizeram um esforço legítimo para aplicar a reengenharia e os resultados levaram ao aperfeiçoamento da competitividade. Outras confiaram apenas na redução e na terceirização (em vez de praticar a reengenharia) para melhorar seus resultados. Freqüentemente resultaram organizações com pouco potencial para crescimento futuro [DEM95].

Durante esta primeira década do século XXI, a onda associada com a reengenharia se atenuou, mas o processo propriamente dito continua em empresas grandes e pequenas.

PANORAMA

O que é? Considere qualquer produto de tecnologia que tenha servido bem a você. Você o utiliza regularmente, mas ele está ficando velho. Quebra com freqüência, leva mais tempo do que gostaria para reparar e deixou de representar a tecnologia mais recente. O que fazer? Se o produto é hardware, você provavelmente vai jogá-lo fora e comprará um modelo mais novo. Mas se for software, feito sob encomenda, essa opção pode não ser possível. Você precisará reconstruí-lo. Terá que criar um produto com funcionalidade adicional, melhor desempenho e confiabilidade, e manutenibilidade aperfeiçoada. Isso é o que chamamos de reengenharia.

Quem faz? No nível da organização, a reengenharia é realizada por especialistas de negócio (freqüentemente empresas de consultoria). No nível de software, a reengenharia é realizada por engenheiros de software.

Por que é importante? Vivemos em um mundo que está se modificando rapidamente. As demandas nas funções de negócio e na tecnologia de informação que a suportam estão se modificando em um ritmo que coloca enorme pressão competitiva em toda organização comercial. Tanto o negócio quanto o software que o apóia (ou constitui) precisam ser trabalhados pela reengenharia para manter o ritmo.

Quais são os passos? A reengenharia de processo do negócio (*Bussiness Process Reengineering*, BPR) define as metas do negócio, identifica e avalia os processos de negócio existentes, e cria processos de negócio revisados que satisfazem melhor os objetivos atuais. O processo de reengenharia de software inclui análise de inventário, reestruturação de documentos, engenharia reversa, reestruturação de programas e dados e engenharia avante. O objetivo dessas atividades é criar versões dos programas existentes com mais qualidade e melhor manutenibilidade.

Qual é o produto do trabalho? Diversos produtos de trabalho de reengenharia (por exemplo, modelos de análise, modelos de projeto e procedimentos de teste) são produzidos. O resultado final é o processo do negócio reengenheirado e/ou do software que o apóia reengenheirado.

Como tenho certeza de que fiz corretamente? Use as mesmas práticas de SQA aplicadas em todo o processo de engenharia de software – revisões técnicas formais avaliam os modelos de análise e projeto, revisões especializadas consideram a aplicabilidade e a compatibilidade com o negócio, são aplicados testes para descobrir erros no conteúdo, na funcionalidade e na interoperabilidade.

Document shared on www.docsity.com

PONTO CHAVE

BPR freqüentemente resulta em nova funcionalidade de software, enquanto reengenharia de software trabalha para substituir a funcionalidade do software existente, por um software melhor, mais manutenível.

A ligação entre a reengenharia do negócio e a engenharia de software está em uma visão de sistema.

O software é freqüentemente a realização das regras de negócio que Hammer discute. À medida que essas regras se modificam, o software também deve ser modificado. Hoje em dia, importantes empresas têm dezenas de milhares de programas de computador que apóiam as regras de negócio antigas. À medida que os gerentes trabalham para modificar as regras, a fim de conseguir maior eficiência e competitividade, o software deve acompanhar o ritmo. Em alguns casos, isso significa a criação de novos sistemas importantes baseados em computador[1]. Mas em muitos outros, significa a modificação ou a reconstrução de aplicações existentes.

Neste capítulo, examinamos a reengenharia de um modo descendente, começando com um breve panorama da reengenharia do processo de negócios e prosseguindo para uma discussão mais detalhada das atividades técnicas que ocorrem quando o software é trabalhado por reengenharia.

31.1 Reengenharia de Processo do Negócio

Reengenharia de processo do negócio (*business process reengineering*, BPR) estende-se muito além do âmbito das tecnologias da informação e da engenharia de software. Entre as várias definições (a maioria um tanto abstrata) que têm sido sugeridas para BPR destaca-se uma publicada na revista *Fortune* [STE93]: "a busca para, e a implementação de, modificações radicais no processo do negócio para conseguir resultados inovadores". Contudo, como essa busca é conduzida e como sua implementação é obtida? Mais importante, como podemos garantir que a "modificação radical" sugerida vai de fato levar a "resultados inovadores" em vez do caos organizacional?

> "Encarar o amanhã pensando em usar os métodos de ontem é imaginar a vida estagnada."
>
> James Bell

31.1.1 Processos de Negócio

Um processo de negócio é "um conjunto de tarefas logicamente relacionadas, realizadas para conseguir um resultado definido do negócio" [DAV90]. No processo do negócio, equipamento, pessoal, recursos materiais e procedimentos de negócios são combinados para produzir um resultado especificado. Exemplos de processos de negócio incluem projetar um produto novo, adquirir serviços e suprimentos, contratar um novo empregado e pagar fornecedores. Cada um demanda um conjunto de tarefas e cada um se apóia em recursos diferentes no negócio.

AVISO

Como engenheiro de software, seu trabalho ocorre na base dessa hierarquia. Certifique-se, no entanto, de que alguém tenha dedicado sério raciocínio aos níveis acima. Se isso não foi feito, seu trabalho corre risco.

Todo processo de negócio tem um cliente definido — uma pessoa ou grupo que recebe o resultado (por exemplo, uma idéia, um relatório, um projeto, um produto). Além disso, os processos de negócio cruzam as fronteiras organizacionais. Eles exigem que diferentes grupos organizacionais participem das "tarefas logicamente relacionadas" que definem o processo.

No Capítulo 6, mencionamos que todo sistema é na verdade uma hierarquia de subsistemas. Um negócio não é exceção. Cada sistema de negócio (também chamado de *função do negócio*) é composto de um ou mais processos de negócio, e cada processo de negócio é definido como um conjunto de subprocessos.

BPR pode ser aplicada em qualquer nível da hierarquia, mas, à medida que o escopo da BPR se amplia (i. e., à medida que subimos na hierarquia), os riscos associados crescem dramaticamente. Por esse motivo, a maioria dos esforços de BPR focaliza processos individuais ou subprocessos.

1 A explosão de aplicações e sistemas baseados na Web, discutidos na Parte 3 deste livro, é indicativa dessa tendência.

> "Tranqüilizamo-nos assim que sabemos da existência de algo antigo em uma coisa nova."
>
> F. W. Nietzsche

31.1.2 Um Modelo BPR

Veja na Web

Vasta informação sobre BPR pode ser encontrada em www.brint.com/BPR.htm.

Como a maioria das atividades de engenharia, a reengenharia de processo do negócio é iterativa. As metas do negócio e os processos que as alcançam precisam ser adaptados a um ambiente de negócios mutante. Por essa razão, não há começo nem fim para a BPR — é um processo evolutivo. Um modelo para a reengenharia de processos de negócio é mostrado na Figura 31.1. O modelo define seis atividades:

Definição do negócio. As metas do negócio são identificadas dentro do contexto de quatro motivações importantes: redução de custo, redução de prazo, aperfeiçoamento da qualidade e poder e desenvolvimento pessoal. As metas podem ser definidas no negócio ou em um componente específico do negócio.

Identificação do processo. São identificados processos críticos para alcançar as metas definidas no negócio. Eles podem ser classificados por importância, por necessidade de modificação ou de qualquer outro modo que seja apropriado para a atividade de reengenharia.

Avaliação do processo. O processo existente é rigorosamente analisado e medido. As tarefas do processo são identificadas; os custos e o tempo consumidos pelas tarefas do processo são registrados, e problemas de qualidade/desempenho são isolados.

Especificação e projeto do processo. Com base na informação obtida durante as três primeiras atividades de BPR, são preparados casos de uso (Capítulo 7) para cada processo que deve ser reprojetado. No contexto de BPR, casos de uso identificam um cenário que fornece algum resultado a um cliente. Com o caso de uso como especificação do processo, um novo conjunto de tarefas é projetado para o processo.

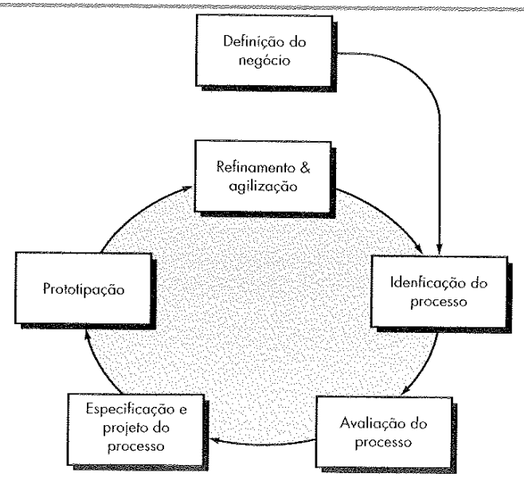
Prototipação. Um processo de negócio reprojetado precisa ser prototipado antes de ser totalmente integrado no negócio. Essa atividade "testa" o processo de modo que possam ser feitos refinamentos.

Refinamento e instanciação. Com base na realimentação obtida do protótipo, o processo de negócio é refinado e então instanciado para um sistema do negócio.

Essas atividades de BPR são algumas vezes usadas em conjunto com ferramentas de análise do fluxo de trabalho. O objetivo dessas ferramentas é construir um modelo do fluxo de trabalho

FIGURA 31.1

Um modelo de BPR



Document shared on www.docsity.com

FERRAMENTAS DE SOFTWARE

Reengenharia de Processo do Negócio (Business Process Reengineering, BPR)

Objetivo: O objetivo das ferramentas BPR é apoiar análise e avaliação de processos do negócio e especificação e projeto de novos.

Mecânica: A mecânica das ferramentas varia. Em geral, ferramentas BPR permitem a um analista de negócio modelar os processos do negócio existentes em um esforço para avaliar as ineficiências do fluxo de trabalho ou problemas funcionais. Uma vez identificados problemas existentes, as ferramentas permitem ao analista fazer o protótipo e/ou simular processos de negócio revisados.

Ferramentas Representativas[2]

Extend, desenvolvida por ImagineThat, Inc. (*www.imaginethatinc.com*), é uma ferramenta de simulação para modelagem de processos existentes e exploração de novos. *Extend* fornece capacidade abrangente "o que – se" para permitir a um analista de negócio explorar diferentes cenários de processo.

e-Work, desenvolvida por Metastorm (*www.metastorm.com*), fornece apoio à gestão de processo de negócio tanto para processos manuais quanto automatizados.

IceTools, desenvolvida por Blue Ice (*www.blueice.com*), é uma coleção de gabaritos BPR para Microsoft Office e Microsoft Project.

SpeeDev, desenvolvida por NimbleStar Group (www.numblestar.com), é uma das muitas ferramentas que permitem a uma organização modelar um fluxo de trabalho de processo (neste caso, fluxo de trabalho IT).

Workflow tools, desenvolvida por MetaSoftware (*www.metasoftware.com*), incorpora um conjunto de ferramentas de modelagem, simulação e cronogramação de fluxo de trabalho.

Uma lista útil de ligações para ferramentas BPR pode ser encontrada em *http://www.donald-firesmith.com/Components/Producers/Tools/BusinessProcessReengineeringTools.html*.

existente em um esforço para melhor analisar os processos existentes. Além disso, as técnicas de modelagem comumente associadas com as atividades de engenharia de processo de negócio (Capítulo 6) podem ser usadas para implementar as quatro primeiras atividades descritas no modelo do processo.

31.2 REENGENHARIA DE SOFTWARE

O cenário é bastante comum: Uma aplicação serviu às necessidades do negócio de uma empresa por 10 ou 15 anos. Durante esse tempo foi corrigida, adaptada e aperfeiçoada muitas vezes. O pessoal abordou esse trabalho com a melhor das intenções, mas boas práticas de engenharia de software foram sempre deixadas de lado (pressionada por outros aspectos). Agora a aplicação está instável. Ainda funciona, mas toda vez que uma modificação é tentada, efeitos colaterais inesperados e sérios ocorrem. No entanto, a aplicação precisa continuar a evoluir. O que fazer?

Software não-manutenível não é um problema novo. De fato, a ênfase cada vez maior na reengenharia de software tem sido motivada por problemas de manutenção de software que têm crescido em tamanho durante mais de 40 anos.

31.2.1 Manutenção de Software

Há mais de três decadas, a manutenção de software foi caracterizada [CAN72] como um "iceberg". Esperamos que o imediatamente visível seja tudo o que existe, mas sabemos que uma enorme massa de possíveis problemas e custo fica sob a superfície. No início dos anos de 1970, o iceberg de manutenção era suficientemente grande para afundar um porta-aviões. Hoje, poderia facilmente afundar toda a Marinha!

A manutenção de software existente pode ser responsável por mais de 60% de todo o esforço despendido por uma organização de desenvolvimento, e a porcentagem continua a crescer à medida que mais software é produzido [HAN93]. Leitores inexperientes podem perguntar por que é necessária tanta manutenção e por que é despendido tanto esforço. Osborne e Chikofsky [OSB90] nos dão uma resposta parcial:

> Grande parte do software do qual dependemos atualmente tem em média de 10 a 15 anos. Mesmo quando esses programas foram criados, usando as melhores técnicas de projeto e codificação conhecidas na época (e muitos não o foram), o tamanho do programa e o espaço de armazenamento eram preocupações importantes. Depois, migraram para novas plataformas, foram ajustados para modificações na tecnologia de máquina e de sistemas operacionais e foram melhorados para satisfazer as novas necessidades do usuário — tudo sem preocupação suficiente com a arquitetura global. O resultado são estruturas mal projetadas, mal codificadas, de lógica pobre e mal documentadas em relação aos sistemas de software, para os quais somos chamados a fim de mantê-los rodando...

Outra razão para o problema de manutenção de software é a mobilidade do pessoal de software. É provável que a equipe (ou pessoa) de software que fez o trabalho original não esteja por perto. Pior, gerações subseqüentes de pessoal de software modificaram o sistema e foram embora. Hoje em dia, pode ser que não se encontre ninguém que tenha algum conhecimento direto do sistema legado. Como mencionamos no Capítulo 27, a natureza ubíqua da modificação permeia todo o trabalho de software. Modificação é inevitável quando sistemas baseados em computador são construídos; conseqüentemente, precisamos desenvolver mecanismos para avaliar, controlar e realizar modificações.

> "Manutenibilidade e inteligibilidade de programas são conceitos paralelos: quanto mais difícil é entender um programa, mais difícil é mantê-lo."
>
> **Gerald Berns**

PONTO CHAVE

Manutenção de software engloba quatro atividades: correção de erros, adaptação, aperfeiçoamento e reengenharia.

Ao ler os parágrafos precedentes, um leitor pode protestar: "Mas eu não gasto 60% do meu tempo consertando erros nos programas que desenvolvi". Manutenção de software é, sem dúvida, muito mais do que "consertar erros". Podemos definir manutenção descrevendo quatro atividades [SWA76] desenvolvidas depois de um programa ser liberado para uso. *Manutenção de software* pode ser definida pela identificação de quatro diferentes atividades: manutenção corretiva, manutenção adaptativa, manutenção perfectiva ou de melhoria, e manutenção preventiva ou reengenharia. Apenas cerca de 20% de todo o trabalho de manutenção é gasto "consertando erros". Os restantes 80% são gastos adaptando sistemas existentes a modificações no seu ambiente externo, fazendo melhorias solicitadas por usuários e submetendo uma aplicação a reengenharia, para uso futuro. Quando a manutenção é considerada como abrangendo todas essas atividades, é relativamente fácil ver por que absorve tanto esforço.

Veja na Web

Uma excelente fonte de informação sobre reengenharia de software pode ser encontrada em www.reenginering.net.

31.2.2 Um Modelo de Processo de Reengenharia de Software

A reengenharia leva tempo; tem um custo significativo em dinheiro, e absorve recursos que poderiam, por outro lado, ser usados em necessidades imediatas. Por todas essas razões, a reengenharia não é conseguida em poucos meses ou mesmo em poucos anos. A reengenharia de sistemas de informação é uma atividade que absorve recursos de tecnologia da informação durante muitos anos. Por isso, toda a organização precisa de uma estratégia pragmática para a reengenharia de software.

Uma estratégia exeqüível é incluída em um modelo de processo de reengenharia. Vamos discutir o modelo posteriormente nesta seção, mas, de início, vejamos alguns princípios básicos.

A reengenharia é uma atividade de reconstrução e a reengenharia de sistemas de informação pode ser mais bem entendida se considerarmos uma atividade análoga: a reconstrução de uma casa. Considere a seguinte situação.

Você comprou uma casa em outro Estado. Nunca viu realmente a propriedade, mas a adquiriu por um preço espantosamente baixo, com a advertência de que talvez ela tivesse que ser completamente reconstruída. Como você procederia?

2 As ferramentas mencionadas aqui não representam uma recomendação, mas, em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

Document shared on www.docsity.com

- Antes de começar a reconstrução, seria razoável inspecionar a casa. Para determinar se ela está precisando ser reconstruída, você (ou um inspetor profissional) criaria uma lista de critérios, de modo que sua inspeção fosse sistemática.
- Antes de demolir e reconstruir a casa toda, certifique-se de que a estrutura está fraca. Se a casa estiver estruturalmente firme, poderá ser possível "reformar", sem reconstruir (a um custo muito mais baixo e em muito menos tempo).
- Antes de começar a reconstrução, entenda como o original foi construído. Dê uma olhada atrás das paredes. Verifique a fiação, a canalização e a estrutura interna. Mesmo se for para descartar, o conhecimento adquirido servirá muito quando iniciar a construção.
- Se começar a reconstruir, use apenas os materiais mais modernos e duradouros. Isso pode custar um pouco mais agora, mas vai ajudá-lo a evitar manutenção dispendiosa e demorada depois.
- Se decidir reconstruir, seja disciplinado a respeito. Use práticas que irão resultar em alta qualidade — hoje e no futuro.

Embora esses princípios focalizem a reconstrução de uma casa, eles se aplicam igualmente bem à reengenharia de sistemas e a aplicações baseadas em computador.

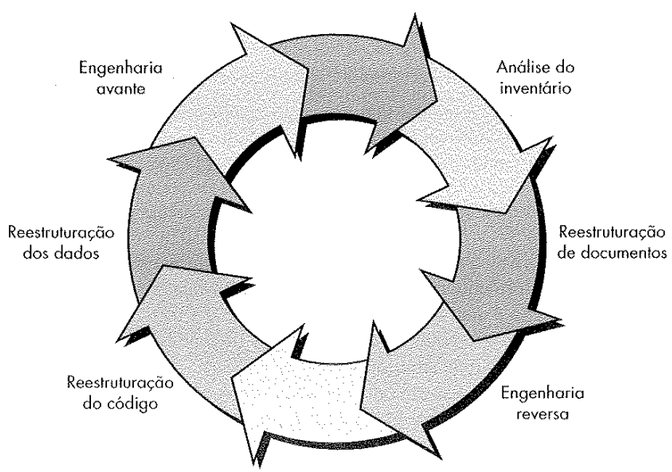
Para implementar esses princípios, aplicamos um modelo de processo de reengenharia de software que define as seis atividades mostradas na Figura 31.2. Em alguns casos, essas atividades ocorrem em uma seqüência linear, mas esse não é sempre o caso. Por exemplo, pode ser que a engenharia reversa (entendimento do funcionamento interno de um programa) tenha que ocorrer antes que a reestruturação de documentos possa começar.

O paradigma de reengenharia mostrado na figura é um modelo cíclico. Isso significa que cada uma das atividades apresentadas como parte do paradigma pode ser revisitada. Para qualquer ciclo particular, o processo pode terminar depois de qualquer uma dessas atividades.

Análise de inventário (*inventory analysis*). Toda organização de software deve ter um inventário de todas as aplicações. O inventário pode ser nada mais do que um modelo de planilha contendo informação que fornece uma descrição detalhada (por exemplo, tamanho, idade, criticalidade para o negócio) de cada aplicação ativa. Ordenando essa informação de acordo com a criticalidade para o negócio, longevidade, manutenibilidade corrente e outros critérios localmente importantes, aparecem candidatos a reengenharia. Recursos podem então ser alocados para aplicações candidatas a trabalho de reengenharia.

FIGURA 31.2

Um modelo de processo de reengenharia de software



AVISO

Se o tempo e os recursos estão com baixo estoque, você pode considerar a aplicação do princípio de Pareto para o software que está sendo engenheirado. Aplicar o processo de reengenharia em 20% do software que é o responsável por 80% dos problemas.

AVISO

Crie apenas a documentação necessária para melhorar o entendimento do software, nem uma página a mais.

É importante notar que o inventário deve ser revisitado em um ciclo regular. O estado de aplicações (por exemplo, criticalidade para o negócio) pode mudar em função do tempo e, como resultado, as prioridades para reengenharia se deslocarão.

Reestruturação de documentos (*document restructuring*). Pouca documentação é a marca registrada de muitos sistemas herdados. Mas o que fazemos a respeito? Quais são as nossas opções?

1. *A criação de documentação consome tempo demais.* Se o sistema funciona, vamos conviver com o que temos. Em alguns casos, essa é a abordagem correta. Não é possível recriar documentação para centenas de programas de computador. Se um programa está relativamente estático, sua vida útil está chegando ao fim e é improvável que passe por modificação significativa, deixe assim!
2. *A documentação precisa ser atualizada, mas temos recursos limitados.* Vamos usar a abordagem "documentar quando tocar". Pode não ser necessário redocumentar totalmente uma aplicação. Em vez disso, as partes do sistema que estão atualmente sofrendo modificações são totalmente documentadas. Ao longo do tempo, uma coleção de documentação útil e relevante vai evoluir.
3. *O sistema é crítico para o negócio e precisa ser totalmente redocumentado.* Uma abordagem inteligente, nesse caso, é limitar a documentação ao mínimo essencial.

Cada uma dessas opções é viável. Uma organização de software deve escolher aquela que é mais adequada a cada caso.

Engenheria reversa. O termo *engenharia reversa* (*reverse engineering*) tem sua origem no mundo do hardware. Uma empresa desmonta um produto competidor de hardware, em um esforço para entender os "segredos" do projeto e a fabricação do concorrente. Esses segredos poderiam ser facilmente entendidos se fossem obtidas as especificações do projeto e da fabricação do concorrente. Mas esses documentos são privados e não estão disponíveis à empresa que está fazendo a engenharia reversa. Na verdade, a engenharia reversa bem-sucedida deriva uma ou mais especificações de projeto e de fabricação de um produto pelo exame de espécimes reais do produto.

A engenharia reversa de software é bastante semelhante. Na maioria dos casos, no entanto, o programa a passar por engenharia reversa não é o de um concorrente. Em vez disso, é trabalho da própria empresa (freqüentemente feito há muitos anos antes). Os "segredos" a ser entendidos são obscuros, porque nenhuma especificação foi jamais desenvolvida. Assim sendo, a engenharia reversa de software é o processo de análise de um programa, em um esforço para representá-lo em uma abstração mais alta do que o código-fonte. A engenharia reversa é um processo de recuperação de projeto. As ferramentas de engenharia reversa extraem informação do projeto de dados, arquitetural e procedimental, para um programa existente.

Veja na Web

Uma grande variedade de recursos para a comunidade de reengenharia pode ser obtida em www.comp.lancs.ac.uk/projects/RenaissanceWeb/.

Reestruturação de código (*code reestructuring*). O tipo mais comum de reengenharia (na verdade, o uso do termo *reengenharia* é questionável neste caso) é a *reestruturação de código*[3]. Alguns sistemas legados têm uma arquitetura de programa relativamente sólida, mas módulos individuais foram codificados de um modo que se torna difícil entendê-los, testá-los e mantê-los. Em tais casos, o código dos módulos suspeitos pode ser reestruturado.

Para realizar essa atividade, o código-fonte é analisado usando uma ferramenta de reestruturação. Violações das construções de programação estruturada são registradas e o código é então reestruturado (isso pode ser feito automaticamente). O código resultante reestruturado é revisado e testado para garantir que nenhuma anomalia foi introduzida. A documentação interna do código é atualizada.

Reestruturação de dados (*data restructuring*). Um programa com arquitetura de dados fraca será difícil de adaptar e aperfeiçoar. De fato, para muitas aplicações, a arquitetura de dados está mais relacionada à viabilidade do programa no longo prazo do que ao código-fonte propriamente dito.

3 Reestruturação de código tem alguns dos elementos de "refatoração", um conceito de reprojeto introduzido no Capítulo 4 e discutido em outras partes deste livro.

Document shared on www.docsity.com

Diferentemente da reestruturação de código, que ocorre em um nível relativamente baixo de abstração, a reestruturação de dados é uma atividade de reengenharia de escala plena. Na maioria dos casos, a reestruturação de dados começa com uma atividade de engenharia reversa. A arquitetura de dados atual é dissecada e os modelos de dados necessários são definidos (Capítulo 9). Objetos de dados e atributos são identificados e as estruturas de dados existentes são revisadas quanto à qualidade.

Quando a estrutura de dados é fraca (por exemplo, arquivos planos estão atualmente implementados, quando uma abordagem relacional simplificaria bastante o processamento), os dados passam por reengenharia.

Como a arquitetura de dados tem uma forte influência na arquitetura do programa e nos algoritmos que o constituem, as modificações nos dados resultarão invariavelmente tanto em modificações arquiteturais quanto de código.

Engenharia avante (*forward engineering*). Em um mundo ideal, as aplicações seriam reconstruídas usando um "motor de reengenharia" automatizado. O programa antigo alimentaria o motor, seria analisado, reestruturado e depois reproduzido de forma que exibisse os melhores aspectos de qualidade de software. No curto prazo, é improvável que tal "motor" apareça, mas fornecedores de CASE introduziram ferramentas que fornecem um subconjunto limitado dessas habilidades, que cuidam de domínios de aplicação específicos (por exemplo, aplicações que são implementadas usando um sistema de bases de dados específico). Mais importante, essas ferramentas de reengenharia estão se tornando cada vez mais sofisticadas.

A engenharia avante, também chamada de *renovação* ou *recomposição* [CHI90], não apenas recupera informação de projeto de software existente, mas usa essa informação para alterar ou reconstituir o sistema existente em um esforço para aperfeiçoar sua qualidade global. Na maioria dos casos, o software trabalhado por reengenharia reimplementa a função do sistema existente e também adiciona novas funções e/ou melhora o desempenho geral.

31.3 Engenharia Reversa

A *engenharia reversa* produz uma imagem de "orifício mágico". Alimentamos uma listagem-fonte, desestruturada, não-documentada no orifício, e no outro extremo sai uma documentação completa para o programa de computador. Infelizmente, o orifício mágico não existe. A engenharia reversa pode extrair informação de projeto do código-fonte, mas o nível de abstração, a completeza da documentação, o grau em que ferramentas e analista trabalham juntos, e a direcionalidade do processo são altamente variáveis.

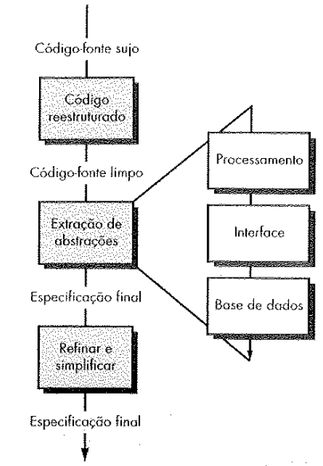
O *nível de abstração* de um processo de engenharia reversa e as ferramentas usadas para executá-lo referem-se à sofisticação da informação de projeto, que pode ser extraída do código-fonte. Idealmente, o nível de abstração deveria ser tão alto quanto possível. Isto é, o processo de engenharia reversa deveria ser capaz de originar representações de projeto procedimental (uma abstração de baixo nível), informação de programa e estrutura de dados (em um nível de abstração um tanto mais alto), modelos de fluxo de dados e de controle (em um nível de abstração relativamente alto) e modelos entidade-relacionamento (em um alto nível de abstração). À medida que o nível de abstração aumenta, é fornecida informação ao engenheiro de software que permitirá entendimento mais fácil do programa.

A *completeza* de um processo de engenharia reversa refere-se ao nível de detalhe que é fornecido em um nível de abstração. Na maioria dos casos, a completeza diminui à medida que o nível de abstração aumenta. Por exemplo, dada uma listagem de código-fonte, é relativamente fácil desenvolver uma representação completa do projeto procedimental. Representações simples de projeto também podem ser derivadas, mas é muito mais difícil desenvolver um conjunto completo de diagramas ou modelos UML.

A completeza melhora em proporção direta com a quantidade de análise realizada por quem está fazendo a engenharia reversa. *Interatividade* se refere ao grau em que uma pessoa é "integrada" com ferramentas automáticas para criar um processo efetivo de engenharia reversa. Na

FIGURA 31.3

O processo de engenharia reversa



PONTO CHAVE

Três tópicos de engenharia reversa precisam ser tratados: nível de abstração, completeza e direcionalidade.

maioria dos casos, à medida que o nível de abstração aumenta, a interatividade precisa aumentar ou a completeza sofrerá.

Se a *direcionalidade* do processo de reengenharia reversa for em um único sentido, toda a informação extraída do código-fonte será fornecida ao engenheiro de software, que pode então usá-la durante qualquer atividade de manutenção. Se a direcionalidade for nos dois sentidos, a informação será alimentada em uma ferramenta de reengenharia que tentará reestruturar ou regenerar o programa antigo.

O processo de engenharia reversa é representado na Figura 31.3. Antes que as atividades de engenharia reversa possam começar, o código-fonte desestruturado ("sujo") é reestruturado (Seção 31.4.1), de modo que contenha apenas as construções de programação estruturada[4]. Isso torna o código-fonte mais fácil de ler e fornece a base para todas as atividades subseqüentes de engenharia reversa.

O âmago da engenharia reversa é uma atividade chamada *extração de abstrações*. O engenheiro deve avaliar o programa antigo e extrair do código-fonte (freqüentemente não-documentado) uma especificação significativa do processamento que é realizado, a interface com o usuário que é aplicada, e as estruturas de dados ou banco de dados do programa que são usadas.

31.3.1 Engenharia Reversa para Entender Dados

Veja na Web

Recursos úteis para "recuperar projeto e entender o programa" podem ser encontrados em www.sel.iit.nrc.ca/projects/dr/dr.html.

A engenharia reversa de dados ocorre em diferentes níveis de abstração. No nível de programa, estruturas de dados internas do programa devem freqüentemente ser submetidas a engenharia reversa como parte de um esforço de reengenharia global. No nível do sistema, estruturas de dados globais (por exemplo, arquivos, bancos de dados) freqüentemente são submetidos à reengenharia para acomodar novos paradigmas de gestão de bancos de dados (por exemplo, a mudança de arquivos planos para sistemas de bases de dados relacionais ou orientados a objetos). A engenharia reversa das estruturas de dados globais atuais arma o palco para a introdução de um novo banco de dados para todo o sistema.

Estruturas de dados internas. Técnicas de engenharia reversa para os dados internos de um programa focalizam uma definição de classes de objetos. Isso é conseguido pelo exame do código do programa com o objetivo de agrupar variáveis do programa relacionadas. Em muitos casos,

4 O código pode ser reestruturado usando um *motor de reestruturação* — uma ferramenta que reestrutura o código-fonte.

Document shared on www.docsity.com

AVISO

Combinações relativamente insignificantes nas estruturas de dados podem causar problemas potencialmente catastróficos em anos futuros. Considere o bug do ano 2000 como um exemplo.

a organização dos dados no código identifica tipos abstratos de dados. Por exemplo, estruturas de registro, arquivos, listas e outras estruturas de dados freqüentemente fornecem um indicador inicial de classes obtido de um banco de dados central.

Estrutura do banco de dados. Independentemente de sua organização lógica e estrutura física, um banco de dados permite a definição de objetos de dados e apóia alguns métodos para estabelecer relações entre objetos. Assim sendo, reengenharia de um esquema de banco de dados para outro exige um entendimento dos objetos e seus relacionamentos existentes.

Os seguintes passos [PRE94] podem ser usados para definir o modelo de dados existente como precursor de um novo modelo de banco de dados por meio de reengenharia: (1) construir um modelo de objetos inicial, (2) determinar candidatos importantes, (3) refinar as classes provisórias, (4) definir generalizações, e (5) descobrir associações (use técnicas análogas à abordagem CRC). Uma vez conhecida a informação definida nos passos precedentes, uma série de transformações [PRE94] pode ser aplicada para mapear a antiga estrutura do banco de dados em uma nova estrutura.

31.3.2 Engenharia Reversa para Entender o Processamento

Engenharia reversa para entender o processamento começa com uma tentativa de entender, e depois extrair, abstrações procedimentais representadas pelo código-fonte. Para entender as abstrações procedimentais, o código é analisado em diferentes níveis de abstração: sistema, programa, componente, padrão e declaração.

A funcionalidade global de todo o sistema da aplicação deve ser entendida antes que ocorra um trabalho mais detalhado de engenharia reversa. Isso estabelece um contexto para posterior análise e possibilita o entendimento de aspectos de interoperabilidade entre as aplicações do sistema. Cada um dos programas que constitui o sistema de aplicação representa uma abstração funcional em um alto nível de detalhes. Um diagrama de blocos, representando a interação entre essas abstrações funcionais, é criado. Cada componente realiza alguma subfunção e representa uma abstração procedimental definida. É criada uma narrativa de processamento para cada componente. Em algumas situações, já existem especificações de sistema, programa e componente. Quando esse é o caso, as especificações são revisadas quanto à concordância com o código existente[5].

> "Existe uma paixão pela compreensão, como existe uma paixão pela música. Essa paixão é bastante comum em crianças, mas posteriormente se perde na maioria das pessoas."
>
> **Albert Einstein**

As coisas tornam-se mais complexas quando o código de um componente é considerado. O engenheiro procura seções do código que representem padrões procedimentais genéricos. Em quase todo o componente, uma seção do código prepara os dados para processamento (dentro do módulo), uma seção do código diferente faz o processamento e outra seção prepara os resultados do processamento para mandar para fora do componente. Dentro de cada uma dessas seções, podemos encontrar pequenos padrões, por exemplo, validação de dados e verificação de limites, que freqüentemente ocorrem dentro da seção de código que prepara dados para processamento.

Para grandes sistemas, a engenharia reversa é geralmente obtida usando uma abordagem semi-automática. Ferramentas automatizadas são usadas para ajudar o engenheiro de software a entender a semântica do código existente. A saída desse processo é então passada para ferramentas de reestruturação e para engenharia avante, para completar o processo de reengenharia.

31.3.3 Engenharia Reversa das Interfaces com o Usuário

IGUs sofisticadas tornaram-se uma exigência de produtos e sistemas de todo o tipo baseados em computador. Assim sendo, o desenvolvimento das interfaces com o usuário tornou-se um dos tipos mais comuns de atividade de reengenharia. Mas, antes que uma interface com o usuário possa ser reconstruída, deve ocorrer a engenharia reversa.

5 Freqüentemente, especificações escritas no início da vida de um programa nunca são atualizadas. À medida que são feitas modificações, o código não continua mais de acordo com a especificação.

? *Como posso entender o funcionamento de uma interface com o usuário existente?*

Para entender totalmente uma interface com o usuário existente, a estrutura e o comportamento da interface devem ser especificados. Merlo e seus colegas [MER93] sugerem três perguntas básicas que devem ser respondidas tão logo a engenharia reversa da UI comece:

- Quais são as ações básicas (por exemplo, pressões de teclas e cliques de mouse) que a interface deve processar?
- Qual a descrição compacta da resposta comportamental do sistema a essas ações?
- O que quer dizer "substituição" ou, mais precisamente, que conceito de equivalência de interfaces é relevante aqui?

A notação de modelagem comportamental (Capítulo 8) pode fornecer um meio para desenvolver respostas às primeiras duas questões. A maior parte da informação necessária para criar um modelo comportamental pode ser obtida pela observação da manifestação externa da interface existente. Mas a informação adicional necessária para criar o modelo comportamental precisa ser extraída do código.

É importante notar que uma IGU substituta não pode espelhar exatamente a interface antiga (de fato, pode ser radicalmente diferente). Freqüentemente, vale a pena desenvolver novas metáforas de interação. Por exemplo, uma IGU antiga exige que um usuário forneça um fator de escala (que varia de 1 a 10) para reduzir ou ampliar uma imagem gráfica. Uma IGU submetida à reengenharia poderia usar uma barra de rolagem e um mouse para conseguir a mesma função.

FERRAMENTAS DE SOFTWARE

Engenharia Reversa

Objetivo: Ajudar engenheiros de software a entender a estrutura de projeto interna de programas complexos.

Mecânica: Na maioria dos casos, ferramentas de engenharia reversa aceitam código-fonte como entrada e produzem uma variedade de representações de projeto estrutural, procedimental, de dados e comportamental.

Ferramentas Representativas[6]

Imagix 4D, desenvolvida por Imagix (*www.imagix.com*), "ajuda desenvolvedores de software a entender um software complexo ou legado C e C++" pela engenharia reversa e documentação do código-fonte.

Understand, desenvolvida por Scientific Toolworks, Inc. (*www.scitools.com*), tem analisadores sintáticos Ada, Fortran, C, C++ e Java "para fazer engenharia reversa, documentar automaticamente, calcular métricas de código e ajudá-lo a entender, navegar e manter código-fonte".

Uma lista abrangente de ferramentas de engenharia reversa pode ser encontrada em *http://scgwiki.iam.unibe.ch:8080/SCG/370*.

31.4 REESTRUTURAÇÃO

A reestruturação do software modifica o código-fonte e/ou os dados em um esforço de torná-los mais amenos a modificações futuras. Em geral, a reestruturação não modifica a arquitetura global do programa. Ela tende a se concentrar nos detalhes de projeto de módulos individuais e nas estruturas de dados locais definidas nos módulos. Se o esforço de reestruturação se estende além dos limites dos módulos e abrange a arquitetura do software, a reestruturação transforma-se em engenharia avante (Seção 31.5).

A reestruturação ocorre quando a arquitetura básica de uma aplicação é sólida, apesar de o interior técnico precisar de trabalho. É iniciada quando as partes principais do software são reparáveis e apenas um subconjunto de todos os módulos e dados precisa de uma grande modificação[7].

6 As ferramentas mencionadas aqui não representam uma recomendação, mas, em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

7 Algumas vezes, é difícil fazer uma distinção entre uma grande reestruturação e um redesenvolvimento. Ambos são reengenharia.

Document shared on www.docsity.com

31.4.1 Reestruturação de Código

AVISO

Apesar de a reestruturação de código poder aliviar problemas imediatos, associados com depuração ou pequenas modificações, não constitui reengenharia. Um benefício real só é alcançado quando os dados e a arquitetura são reestruturados.

Reestruturação de código é realizada para executar um projeto que produz a mesma função que o programa original, porém com mais qualidade. Em geral, as técnicas de reestruturação de código (por exemplo, as técnicas de simplificação lógica de Warnier [WAR74]) modelam a lógica do programa usando álgebra booleana e depois aplicam uma série de regras de transformação que produzem a lógica reestruturada. O objetivo é pegar um código "emaranhado" e originar um projeto procedimental que respeite a filosofia da programação estruturada (Capítulo 11).

Outras técnicas de reestruturação foram propostas também para uso com ferramentas de reengenharia. Um diagrama de intercâmbio de recursos mapeia cada módulo do programa e os recursos (tipos de dados, procedimentos e variáveis) que são intercambiados entre ele e os outros módulos. Pela criação de representações do fluxo de recursos, a arquitetura do programa pode ser reestruturada para um acoplamento mínimo entre os módulos.

31.4.2 Reestruturação dos Dados

Antes que a reestruturação dos dados possa começar, uma atividade de engenharia reversa chamada de *análise do código-fonte* deve ser conduzida. Todas as declarações em linguagem de programação que contêm definições de dados, descrições de arquivos, entradas e saídas, e descrições de interfaces são avaliadas. O objetivo é extrair itens de dados e objetos, para obter informação sobre o fluxo de dados e entender as estruturas de dados existentes que foram implementadas. Essa atividade é algumas vezes chamada de *análise de dados* [RIC89].

Uma vez completada a análise de dados, começa o *reprojeto dos dados*. Em sua forma mais simples, a etapa de *padronização de registro de dados* esclarece definições de dados para obter consistência entre os nomes dos itens de dados, ou entre os formatos de registro físico dentro da estrutura de dados existente ou do formato de arquivo. Outra forma de reprojeto, chamada de *racionalização dos nomes dos dados*, garante que todas as convenções de denominação de dados atendam aos padrões locais e que os sinônimos sejam eliminados à medida que os dados fluam através do sistema.

Quando a reestruturação se estende além da padronização e da racionalização, são feitas modificações físicas em estruturas de dados existentes para tornar o processo de dados mais efetivo. Isso pode significar uma translação de um formato de arquivo para outro ou, em alguns casos, uma translação de um tipo de banco de dados para outro.

FERRAMENTAS DE SOFTWARE

Reestruturação de Software

Objetivo: O objetivo das ferramentas de reestruturação é transformar um software de computador mais antigo não estruturado em linguagens de programação e estruturas de projeto modernas.

Mecânica: Em geral, o código-fonte é introduzido e transformado em um programa mais bem estruturado. Em alguns casos a transformação ocorre com a mesma linguagem de programação. Em outros casos, uma linguagem mais antiga é transformada em uma linguagem de programação mais moderna.

Ferramentas Representativas[8]

DMS Software Reengineering Toolkit, desenvolvida por Semantic Design (*www.semdesigns.com*), fornece uma variedade de capacidades de reestruturação para COBOL, C/C++, Java, FORTRAN 90 e VHDL.

FORESYS, desenvolvida por Simulog (www.simulog.fr), analisa e transforma programas escritos em FORTRAN.

Function Encapsulation Tool, desenvolvida por Wayne State University (*www.cs.wayne.edu/_vip/RefactoringTools/*), refabrica programas antigos C em C++.

plusFORT, desenvolvida por Polyhedron (*www.polyhedron.com*), é um conjunto de ferramentas FORTRAN que contém capacidades para reestruturar programas FORTRAN pobremente projetados em normas modernas de FORTRAN ou C.

8 As ferramentas mencionadas aqui não representam uma recomendação, mas, em vez disso, uma exemplificação de ferramentas dessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

31.5 Engenharia Avante

? *Quais são as opções quando nos defrontamos com um programa pobremente projetado e implementado?*

Para acomodar alterações exigidas pelo usuário, um programa com um fluxo de controle, que graficamente equivale a um prato de macarrão contendo "módulos" com 2 mil declarações, com poucas linhas de comentário em 290 mil declarações-fonte, e mais nenhuma outra documentação, precisa ser modificado. Temos as seguintes opções:

1. Podemos batalhar modificação por modificação, combatendo o projeto implícito e o código-fonte para implementar as mudanças necessárias.
2. Podemos tentar entender o funcionamento interno global do programa em um esforço para tornar as modificações mais eficazes.
3. Podemos reprojetar, recodificar e testar aquelas partes do software que exigem modificação, aplicando uma abordagem de engenharia de software para todos os segmentos revistos.
4. Podemos reprojetar, recodificar e testar completamente o programa usando ferramentas (de reengenharia) CASE para nos ajudar a entender o projeto atual.

Não há uma opção "correta" única. As circunstâncias podem levar à primeira opção, ainda que as outras sejam mais desejáveis.

Em vez de esperar por uma solicitação de manutenção, a organização de desenvolvimento ou de suporte usa os resultados de uma análise de inventário para selecionar um programa que (1) permanecerá em uso durante um número de anos preestabelecido, (2) está atualmente sendo usado com sucesso e (3) provavelmente passará por modificação ou aperfeiçoamento radical em um futuro próximo. Então, a opção 2, 3 ou 4 é aplicada.

Essa abordagem de *manutenção preventiva* foi iniciada por Miller [MIL81] sob o título de *reajuste estruturado*. Esse conceito é definido como "a aplicação de metodologias atuais a sistemas de ontem, para suportar os requisitos de amanhã".

AVISO

Reengenharia é muito parecida com a limpeza dos seus dentes. Você pode pensar em mil razões para adiá-la e você vai conseguir lidar com o adiamento durante algum tempo. Mas talvez suas táticas de adiamento voltem para lhe causar dor.

À primeira vista, a sugestão para redesenvolvermos um programa grande, quando já existe uma versão em funcionamento, pode parecer bastante extravagante. Antes de fazer um julgamento, considere os seguintes pontos:

1. O custo para manter uma linha de código-fonte pode ser de 20 a 40 vezes o custo do desenvolvimento inicial daquela linha.
2. O reprojeto da arquitetura do software (programa e/ou estrutura de dados), usando conceitos modernos de projeto, pode facilitar bastante a manutenção futura.
3. Como já existe um protótipo do software, a produtividade do desenvolvimento deve ser muito maior do que a média.
4. O usuário agora tem experiência com o software. Conseqüentemente, podem ser combinados os novos requisitos e o rumo da modificação com maior facilidade.
5. Ferramentas automatizadas para reengenharia vão automatizar algumas partes do serviço.
6. Um conjunto completo da configuração do software (documentos, programas e dados) passará a existir no fim da manutenção preventiva.

Quando uma organização de desenvolvimento de software vende o software como um produto, a manutenção preventiva é vista como "novas versões" de um programa. Um desenvolvedor de software interno (por exemplo, um grupo de desenvolvimento de software de sistemas comerciais para uma grande empresa de produtos de consumo) pode ter de 500 a 2 mil programas em produção dentro de sua área de responsabilidade. Esses programas podem ser ordenados por importância e depois revistos como candidatos à manutenção preventiva.

O processo de engenharia avante aplica princípios, conceitos e métodos de engenharia de software para recriar uma aplicação existente. Na maioria dos casos, a engenharia avante não cria simplesmente um equivalente moderno de um programa antigo. Em vez disso, requisitos novos do usuário e tecnologia são integrados no esforço de reengenharia. O programa redesenvolvido amplia a capacidade da aplicação antiga.

Document shared on www.docsity.com

31.5.1 Engenharia Avante para Arquiteturas Cliente/Servidor

Na última década, muitas aplicações de computadores de grande porte passaram por reengenharia para acomodar arquiteturas cliente/servidor (incluindo WebApps). Na verdade, recursos computacionais centralizados (inclusive software) foram distribuídos entre várias plataformas-cliente. Apesar de diversos ambientes distribuídos diferentes poderem ser projetados, a aplicação típica de computadores de grande porte, trabalhada por reengenharia para uma arquitetura cliente/servidor, tem as seguintes características:

- A funcionalidade da aplicação migra para cada computador cliente.
- Novas interfaces IGU são implementadas nas instalações do cliente.
- As funções do banco de dados são atribuídas ao servidor.
- Funcionalidade especializada (por exemplo, análise que demanda muitos cálculos) pode permanecer no servidor.
- Novos requisitos de comunicações, segurança, arquivamento e controle podem ser estabelecidos tanto para o cliente quanto para o servidor.

AVISO

Em alguns casos, migração para uma arquitetura cliente/servidor deve ser abordada não como reengenharia, mas como um esforço de desenvolvimento novo. A reengenharia entra em cena apenas quando a funcionalidade específica do sistema antigo precisa ser integrada na nova arquitetura.

É importante notar que a migração de um computador de grande porte para cliente/servidor exige reengenharia, tanto do negócio quanto de software. Além disso, deve ser estabelecida uma "infraestrutura de rede empresarial" [JAY94].

A reengenharia de aplicações-cliente/servidor começa com uma análise rigorosa do ambiente de negócio que abrange o computador de grande porte existente. Três camadas de abstração podem ser identificadas. O *banco de dados* fica na fundação de uma arquitetura cliente/servidor e gerencia transações e consultas das aplicações do servidor. No entanto, essas transações e consultas precisam ser controladas dentro do contexto de um conjunto de regras de negócio (definidas por um processo de negócio existente ou trabalhado por reengenharia). As aplicações-cliente fornecem uma funcionalidade dirigida à comunidade usuária.

As funções do sistema de gestão de banco de dados e a arquitetura de dados do banco de dados existentes devem passar por engenharia reversa como precursoras do reprojeto da camada fundamental do banco de dados. Em alguns casos, é criado um novo modelo de dados (Capítulo 8). De qualquer forma, o banco de dados cliente/servidor é trabalhado por reengenharia para garantir que as transações sejam executadas de maneira consistente, que todas as atualizações sejam realizadas apenas por usuários autorizados; que as regras vitais do negócio sejam seguidas (por exemplo, antes que um registro de fornecedor seja apagado, o servidor assegura-se de que não existem contas a pagar, contratos ou comunicações relacionadas àquele fornecedor), que as consultas possam ser acomodadas eficientemente e que a capacidade de arquivamento foi estabelecida plenamente.

A *camada de regras do negócio* representa o software residente tanto no cliente quanto no servidor. Esse software realiza tarefas de controle e coordenação, para garantir que as transações e consultas entre as aplicações de cliente e o banco de dados respeitem o processo de negócio estabelecido.

A *camada de aplicações-cliente* implementa funções de negócio que são exigidas por grupos específicos de usuários finais. Em muitos exemplos, uma aplicação de computador de grande porte é sedimentada em várias aplicações de micros e trabalhadas por reengenharia. As comunicações entre as aplicações de micro (quando necessárias) são controladas pela camada de regras do negócio.

Uma discussão abrangente do projeto e reengenharia de software cliente/servidor é mais bem abordada em livros dedicados ao assunto. O leitor interessado deve ver [VAN02], [COU00] e [ORF99].

31.5.2 Engenharia Avante para Arquiteturas Orientadas a Objetos

A engenharia de software orientada a objetos tornou-se o paradigma de desenvolvimento preferido por muitas organizações de software. Mas e quanto às aplicações existentes, que foram desenvolvidas usando métodos convencionais? Em alguns casos, a resposta é deixar tais aplicações

"como estão". Em outros, aplicações antigas devem passar por engenharia, de modo que possam ser facilmente integradas em sistemas orientados a objetos de grande porte.

A reengenharia de software convencional, para uma implementação orientada a objetos, usa muitas das técnicas discutidas na Parte 2 deste livro. Primeiro, o software existente passa por engenharia reversa, de modo que os modelos adequados de dados, funcional e comportamental, possam ser criados. Se o sistema trabalhado por reengenharia amplia a funcionalidade ou o comportamento da aplicação original, são criados casos de uso (Capítulos 7 e 8). Os modelos de dados criados durante a engenharia reversa são depois usados em conjunto com a modelagem CRC (Capítulo 8) para estabelecer a base para a definição de classes. Hierarquias de classe, modelos de objeto-relacionamento, modelos de objeto-comportamento e subsistemas são definidos, e o projeto orientado a objetos começa.

À medida que a engenharia avante orientada a objetos progride da análise para o projeto, um modelo de processo CBSE (Capítulo 30) pode ser invocado. Se a aplicação existente situa-se em um domínio que já seja constituído por muitas aplicações orientadas a objeto, é provável que exista uma biblioteca de componentes poderosa que possa ser usada durante a engenharia avante.

Para aquelas classes que precisam passar por engenharia a partir do zero, pode ser possível reusar algoritmos e estruturas de dados da aplicação convencional existente. No entanto, devem ser reprojetados para obedecer à arquitetura orientada a objetos.

31.5.3 Engenharia Avante de Interfaces com o Usuário

À medida que as aplicações migram de um computador de grande porte para micros, os usuários não se dispõem mais a tolerar interfaces antigas, baseadas em caracteres. De fato, uma parte significativa de todo o esforço despendido na transição de computação de grande porte para cliente/servidor pode ser dedicada à reengenharia das interfaces com o usuário das aplicações-cliente.

Merlo e seus colegas [MER95] sugerem o seguinte modelo para trabalhar as interfaces com o usuário por reengenharia:

AVISO

Que passos devemos seguir para trabalhar uma interface com o usuário por reengenharia?

1. *Entender a interface original e os dados que fluem entre ela e o restante da aplicação.* O objetivo é entender como outros elementos de um programa interagem com o código existente, que implementa a interface. Se uma nova IGU precisa ser desenvolvida, o fluxo de dados entre a IGU e o restante do programa precisa ser consistente com os dados que fluem atualmente entre a interface baseada em caracteres e o programa.
2. *Remodelar o comportamento implícito na interface existente em uma série de abstrações que tenham sentido no contexto de uma IGU.* Apesar de o modo de interação poder ser radicalmente diferente, o comportamento de negócios exibido pelos usuários das interfaces nova e antiga (quando considerados em termos de um cenário de uso) deve permanecer o mesmo. Uma interface reprojetada deve continuar a permitir ao usuário exibir o comportamento de negócios adequado. Por exemplo, quando uma consulta ao banco de dados precisar ser feita, a interface antiga pode exigir uma longa série de comandos, baseados em texto, para especificar a consulta. A IGU trabalhada por reengenharia pode efetuar a consulta por meio de uma pequena seqüência de movimentos do mouse, mas o objetivo e o conteúdo da consulta permanecem inalterados.
3. *Introduzir aperfeiçoamentos que tornem o modo de interação mais eficiente.* As falhas ergonômicas da interface existente são estudadas e corrigidas no projeto da nova IGU.
4. *Construir e integrar a nova IGU.* A existência de bibliotecas de classe e de ferramentas automatizadas pode reduzir o esforço necessário para construir a IGU significativamente. No entanto, a integração com o software da aplicação existente pode levar muito tempo. Deve ser tomado cuidado para garantir que a IGU não propague efeitos colaterais adversos para o restante da aplicação.

Veja na Web

Um manual com mais de 300 páginas sobre padrões de reengenharia (desenvolvido como parte do projeto FAMOOS ESPIRIT) pode ser baixado de www.iam.unibe.ch/~scg/Archive/famoos/patterns/index3.html.

"Você pode nos pagar pouco agora ou nos pagar muito mais depois."

Cartaz em uma concessionária de automóveis sugerindo uma revisão

Document shared on www.docsity.com

31.6 A Economia da Reengenharia

Em um mundo perfeito, cada programa não-manutenível seria aposentado imediatamente, substituído por aplicações submetidas a reengenharia de alta qualidade, desenvolvidas usando práticas modernas de engenharia de software. Mas vivemos em um mundo de recursos limitados. A reengenharia drena recursos que podem ser usados para outros fins do negócio. Conseqüentemente, uma organização deve realizar uma análise custo/benefício antes de tentar submeter a reengenharia a uma aplicação existente.

Um modelo de análise custo/benefício para reengenharia foi proposto por Sneed [SNE95]. Nove parâmetros são definidos:

P_1 = custo de manutenção anual corrente para uma aplicação.
P_2 = custo de operação anual corrente para uma aplicação.
P_3 = valor de negócio anual corrente de uma aplicação.
P_4 = custo de manutenção anual previsto após a reengenharia.
P_5 = custo de operação anual previsto após a reengenharia.
P_6 = valor de negócio anual previsto após a reengenharia.
P_7 = custo estimado de reengenharia.
P_8 = período estimado para a reengenharia.
P_9 = fator de risco de reengenharia (P_9 = 1,0 é nominal).
L = esperança de vida do sistema.

O custo associado com a manutenção continuada de uma aplicação candidata (i. e., não é realizada reengenharia) pode ser definido como:

$$C_{manut} = [P_3 - (P_1 + P_2)] \times L \qquad (31\text{-}1)$$

Os custos associados com a reengenharia são definidos usando a seguinte relação:

$$C_{reeng} = [P_6 - (P_4 + P_5) \times (L - P_8) - (P_7 \times P_9)] \qquad (31\text{-}2)$$

Usando os custos apresentados nas equações (31-1) e (31-2), o benefício total da reengenharia pode ser calculado como:

$$\text{custo/benefício} = C_{reeng} - C_{manut} \qquad (31\text{-}3)$$

A análise custo/benefício apresentada nas equações pode ser realizada para todas as aplicações de alta prioridade identificadas durante a análise do inventário (Seção 31.2.2). As aplicações que exibem o maior custo/benefício podem ser destinadas à reengenharia, enquanto o trabalho nas outras pode ser adiado até que haja disponibilidade de recursos.

31.7 Resumo

A reengenharia ocorre em dois níveis de abstração diferentes. No nível do negócio, a reengenharia focaliza o processo do negócio com o objetivo de fazer modificações para aperfeiçoar a competitividade em algumas áreas. No nível do software, a reengenharia examina sistemas de informação e aplicações com o objetivo de reestruturá-los ou reconstruí-los, de modo que apresentem mais qualidade.

A reengenharia de processo do negócio define metas do negócio, identifica e avalia os processos de negócio existentes (no contexto das metas definidas), especifica e projeta os processos revisados, faz seus protótipos, refina e agiliza os processos dentro de um negócio. A BPR tem um foco que se estende além do software. O resultado da BPR é freqüentemente a definição dos modos pelos quais as tecnologias da informação podem suportar melhor o negócio.

A engenharia de software abrange uma série de atividades que inclui análise do inventário, reestruturação de documentos, engenharia reversa, reestruturação de programas e dados e engenharia avante. O objetivo dessas atividades é criar versões dos programas existentes que apresentem mais qualidade e melhor manutenibilidade — programas que serão bastante viáveis para o século XXI.

A análise de inventário permite que uma organização avalie cada aplicação sistematicamente, com o objetivo de determinar quais são candidatas à reengenharia. A reestruturação de documentos cria um arcabouço de documentação necessário para o suporte no longo prazo de uma aplicação. A engenharia reversa é o processo de análise de um programa, em um esforço de extrair informação de projeto de dados, arquitetural e procedimental. Finalmente, a engenharia avante reconstrói um programa usando práticas modernas de engenharia de software e informação adquirida durante a engenharia reversa.

O custo/benefício da reengenharia pode ser determinado quantitativamente. O custo do estado atual, isto é, o custo associado com o suporte e a manutenção atuais de uma aplicação existente é comparado com os custos projetados da reengenharia e da redução resultante em custos de manutenção. Em quase todos os casos nos quais um programa tem uma vida longa, e no momento apresenta baixa mantenabilidade, a reengenharia representa uma estratégia de negócios eficiente em termos de custo.

Referências Bibliográficas

[CAN72] Canning, R., "The Maintenance 'Iceberg'", *EDP Analyzer*, v. 10, n. 10, out. 1972.

[CAS88] "Case Tools for Reverse Engineering", *CASE Outlook*, CASE Consulting Group, v. 2, n. 2, 1988, p. 1-15.

[CHI90] Chikofsky, E. J. e Cross, J. H., II, "Reverse Engineering and Design Recovery: A Taxonomy", *IEEE Software*, jan. 1990, p. 13-17.

[COU00] Coulouris, G., Dollimore, J. e Kindberg, T., *Distributed Systems: Concepts and Design*, 3ª ed., Addison-Wesley, 2000.

[DAV90] Davenport, T. H. e Young, J. E., "The New Industrial Engineering: Information Technology and Business Process Redesign", *Sloan Management Review*, verão 1990, p. 11-27.

[DEM95] DeMarco, T., "Lean and Mean", *IEEE Software*, nov. 1995, p. 101-102.

[HAM90] Hammer, M., "Reengineer Work: Don't Automate, Obliterate", *Harvard Business Review*, jul./ago. 1990, p. 104-112.

[HAN93] Manna, M., "Maintenance Burden Begging for a Remedy", *Datamation*, abr. 1993, p. 53-63.

[JAY94] Jaychandra, Y., *Re-engineering the Networked Enterprise*, McGraw-Hill, 1994.

[MER93] Merlo, E. *et al.*, "Reverse Engineering of User Interfaces", *Proc. Working Conference on Reverse Engineering*, IEEE, Baltimore, maio 1993, p. 171-178.

[MER95] Merlo, E. *et al.*, "Reengineering User Interfaces", *IEEE Software*, jan. 1995, p. 64-73.

[MIL81] Miller, J., em *Techniques of Program and System Maintenance* (G. Parikh, ed.), Winthrop Publishers, 1981.

[ORF99] Orfali, R., Harkey, D. e Edwards, J., *Client/Server Survival Guide*, 3ª ed., Wiley, 1999.

[OSB90] Osborne, W. M. e Chikofsky, E. J., "Fitting Pieces to the Maintenance Puzzle", *IEEE Software*, jan. 1990, p. 10-11.

[PRE94] Premerlani, W. J. e Blaha, M. R., "An Approach for Reverse Engineering of Relational Databases", *CACM*, v. 37, n. 5, maio 1994, p. 42-49.

[RIC89] Ricketts, J. A., DelMonaco, J. C. e Weeks, M. W., "Data Reengineering for Application Systems", *Proc. Conf. Software Maintenance – 1989*, IEEE, 1989, p.174-179.

[SNE95] Sneed, H., "Planning the Reengineering of Legacy Systems", *IEEE Software*, jan. 1995, p. 24-25.

[STE93] Stewart, T. A., "Reengineering: The Hot New Managing Tool", *Fortune*, ago. 23, 1993, p. 41-48.

[SWA76] Swanson, E. B., "The Dimensions of Maintenance", *Proc. Second Intl. Conf. Software Engineering*, IEEE, out. 1976, p. 492-497.

[VAN02] Van Steen, M. e Tanenbaum, A., *Distributed Systems: Principles and Paradigms*, Prentice-Hall, 2002.

[WAR74] Warnier, J. D., *Logical Construction of Programs*, Van Nostrand-Reinhold, 1974.

Document shared on www.docsity.com

Problemas e Pontos a Considerar

31.1. Considere qualquer emprego que tenha tido durante os últimos cinco anos. Descreva o processo do negócio no qual você tomou parte. Use o modelo de BPR descrito na Seção 31.1.3 para recomendar modificações no processo, em um esforço de torná-lo mais eficiente.

31.2. Pesquise sobre a eficácia da reengenharia de processo de negócios. Apresente argumentos pró e contra essa abordagem.

31.3. Seu instrutor deve selecionar um dos programas desenvolvido pela turma durante este curso. Troque seu programa aleatoriamente com outro colega. Não explique ou explore o programa. Agora, implemente um aperfeiçoamento (especificado pelo seu instrutor) no programa que você recebeu.

a. Realize todas as tarefas de engenharia de software, inclusive um rápido *walkthrough* (mas não com o autor do programa).
b. Mantenha um registro cuidadoso dos erros encontrados durante o teste.
c. Discuta sua experiência em sala.

31.4. Explore uma lista de verificação de análise de inventário apresentada no site da SEPA e tente desenvolver um sistema quantitativo de avaliação de software que poderia ser aplicado aos programas existentes, em um esforço de escolher programas candidatos à reengenharia. Seu sistema deve ir além da análise econômica apresentada na Seção 31.6.

31.5. Sugira alternativas à documentação manuscrita ou eletrônica convencional que poderia servir como base para a reestruturação de documentos. (Dica: Pense em novas tecnologias descritivas que poderiam ser usadas para comunicar o objetivo ao software.)

31.6. Algumas pessoas acreditam que a tecnologia de inteligência artificial vai aumentar o nível de abstração do processo de engenharia reversa. Faça uma pesquisa sobre esse assunto (i. e., o uso de IA para engenharia reversa) e redija um trabalho resumido, assumindo uma posição sobre isso.

31.7. Por que é difícil conseguir completeza quando o nível de abstração aumenta?

31.8. Por que a interatividade precisa aumentar se a completeza tiver que aumentar?

31.9. Usando informação obtida via Internet, apresente características de três ferramentas de engenharia reversa para sua classe.

31.10. Há uma diferença sutil entre reestruturação e engenharia avante. Qual é ela?

31.11. Pesquise a literatura para encontrar um ou mais trabalhos que discutam estudos de caso sobre reengenharia de computador de grande porte para cliente/servidor. Apresente um resumo.

31.12. Como você poderia determinar P_4 até P_7 em um modelo de custo/benefício apresentado na Seção 31.6?

Leituras e Fontes de Informação Adicionais

Como muitos assuntos em moda na comunidade de negócios, a onda em torno da reengenharia de processo do negócio deu lugar a uma visão mais pragmática do assunto. Hammer e Champy (*Reengineering the Corporation*, HarperBusiness, revised edition, 2001) precipitaram o interesse inicial com seu livro de sucesso de vendas. Depois Hammer (*Beyond Reengineering: How the Processed-Centered Organization Is Changing Our Work and Our Lives*, Harper-Collins, 1997) refinou sua visão focalizando tópicos "centrados no processo".

Livros de Smith e Fingar (*Business Process Management (BPM): The Third Wave*, Meghan-Kiffer Press, 2003), Jacka e Keller (*Business Process Mapping: Improving Customer Satisfaction*, Wiley, 2001), Sharp e McDermott (*Workflow Modeling*, Artech House, 2001), Andersen (*Business Process Improvement Toolbox*, American Society for Quality, 1999) e Harrington *et al.*(*Business Process Improvement Workbook*, McGraw-Hill, 1997) apresentam estudos de caso e diretrizes detalhadas para BPR.

Feldmann (*The Practical Guide to Business Process Reengineering Using IDEF0*, Dorset House, 1998) discute uma notação de modelagem que apóia a BPR. Berztiss (*Software Methods for Business Reengineering*, Springer, 1996) e Spurr *et al.* (*Software Assistance for Business Reengineering*, Wiley, 1994) discutem ferramentas e técnicas que facilitam a BPR.

Secord e seus colegas (*Modernizing Legacy Systems*, Addison-Wesley, 2003), *Ulrich (Legacy Systems: Transformation Strategies*, Prentice-Hall, 2002), Valenti (*Successful Software Reengineering*, IRM Press, 2002) e Rada (*Reengineering Software: How to Reuse Programming to Build New, State-of-the-Art Software*, Fitzroy

Dearborn Publishers, 1999) enfocam estratégias e práticas para reengenharia em um nível técnico. Miller (*Reengineering Legacy Software Systems*, Digital Press, 1998) "fornece um arcabouço para guardar sistemas de aplicação sincronizados com estratégias de negócio e mudanças tecnológicas". Umar (*Application (Re)Engineering: Building Web-Based Applications and Dealing with Legacies*, Prentice-Hall, 1997) fornece guia que vale a pena para empresas que querem transformar sistemas legados em ambientes baseados na Web. Cook (*Building Enterprise Information Architectures: Reengineering Information Systems*, Prentice-Hall, 1996) discute a ligação entre BPR e tecnologia de informação. Aiken (*Data Reverse Engineering*, McGraw-Hill, 1996) discute como recuperar, reorganizar e reusar dados organizacionais. Arnold (*Software Reengineering*, IEEE Computer Society Press, 1993) reuniu uma excelente antologia de artigos pioneiros que enfocam tecnologias de reengenharia de software.

Uma ampla variedade de fontes de informação sobre reengenharia de software está disponível na Internet. Uma lista atualizada de referências da World Wide Web que são relevantes pode ser encontrada no site deste livro: **http://www.mhhe.com/pressman**.

Document shared on www.docsity.com

CAPÍTULO

A Estrada Adiante

Conceitos-Chave

alcance das modificações . . 701
conhecimento 705
dados 704
ética 707
informação 704
pessoas 702
processo 703
software revisitado 701
tendências de tecnologia . . 706

Nos 31 capítulos que precederam a este, exploramos um processo para a engenharia de software. Apresentamos tanto procedimentos gerenciais quanto métodos técnicos, princípios básicos e técnicas especializadas, atividades orientadas a pessoas e tarefas que são próprias para a automação, anotações com lápis e papel e ferramentas de software. Argumentamos que medições, disciplina e um enfoque prevalente na qualidade vão resultar em um software que satisfaz as necessidades do cliente, um software que é confiável, um software que é manutenível, um software que é *melhor*. No entanto, nunca prometemos que a engenharia de software é uma panacéia.

À medida que começamos nossa jornada por um novo século, as tecnologias de software e sistemas permanecem um desafio para todo o profissional de software e toda a empresa que constrói sistemas baseados em computador. Apesar de ter escrito essas palavras com uma visão do século XX, Max Hopper [HOP90] descreve corretamente o estado atual da situação:

> Uma vez que as modificações na tecnologia de informação estão se tornando tão rápidas e inclementes, e as conseqüências de ficar para trás são irreversíveis, ou as empresas dominam a tecnologia ou morrem... Pense nisso como um moinho de tecnologia. As empresas terão de correr mais e mais para ficar no mesmo lugar.

As modificações na tecnologia da engenharia de software são de fato "rápidas e inclementes", e, ao mesmo tempo, o progresso é freqüentemente bastante lento. Até que seja tomada a decisão de adotar um novo método (ou uma nova ferramenta), seja realizado o treinamento necessário para entender sua aplicação e seja introduzida a tecnologia na cultura de desenvolvimento de software, algo mais novo (e ainda melhor) já surgiu e o processo começa novamente.

Neste capítulo, examinamos a estrada adiante. Nosso objetivo não é explorar toda a área de pesquisa, que é promissora. Nem é olhar para uma "bola de cristal" e prognosticar o futuro.

Panorama

O que é? O futuro nunca é fácil de prever – apesar dos gurus, das cabeças pensantes e dos especialistas da indústria. A estrada adiante está suja com as carcaças de novas tecnologias excitantes, que nunca vingaram de fato (apesar da onda) e são freqüentemente moldadas por tecnologias mais modestas, que de algum modo modificam a direção e a largura da via expressa. Assim sendo, não vamos tentar prever o futuro. Em vez disso, vamos discutir alguns dos assuntos que você precisará considerar, para entender como o software e a engenharia de software vão se modificar nos próximos anos.

Quem faz? Todos!

Por que é importante? Por que os antigos reis contratavam pitonisas? Por que as principais corporações multinacionais contratam firmas de consultoria e institutos de pesquisa para preparar previsões? Por que uma porcentagem substancial do público lê horóscopos? Desejamos saber o que vem por aí, de modo que possamos nos preparar.

Quais são os passos? Não há fórmula para antever a estrada adiante. Tentamos fazer isso coletando dados, organizando-os para obter informação útil, examinando associações sutis para extrair conhecimento e, a partir desse conhecimento, sugerir prováveis ocorrências que prevêem como serão as coisas em algum momento no futuro.

Qual o produto do trabalho? Um panorama do futuro próximo que pode ou não ser correto.

Como tenho certeza de que fiz corretamente? Antever a estrada adiante é uma arte, não uma ciência. De fato, é raro quando uma previsão séria do futuro é absolutamente correta ou inequivocamente errada (com exceção, felizmente, das previsões do fim do mundo). Procuramos tendências e tentamos extrapolá-las para o futuro. Podemos avaliar a correção da extrapolação apenas com o passar do tempo.

Em vez disso, exploramos o alcance das modificações e o modo pelo qual as modificações propriamente ditas vão afetar o processo de engenharia de software nos anos vindouros.

32.1 A Importância do Software — Revisitada

A importância do software de computador pode ser enunciada de vários modos. No Capítulo 1, o software foi caracterizado como um diferenciador. A função incorporada pelo software diferencia os produtos, sistemas e serviços, e fornece vantagem competitiva no mercado. Mas o software é mais do que um diferenciador. Os programas, documentos e dados que o constituem o software ajudam a gerar o bem mais importante que qualquer indivíduo, negócio ou governo pode adquirir — a *informação*. Pressman e Herron [PRE91] descrevem software do seguinte modo:

> Software de computador é uma entre poucas tecnologias importantes que terão um impacto significativo em praticamente todo o aspecto da sociedade moderna... É um mecanismo para automatizar negócios, indústria e governo, um meio para transferir nova tecnologia, um método de captar conhecimento (*expertise*) valioso para ser usado por outros, um modo de diferenciar os produtos de uma empresa de seus competidores e uma janela para o conhecimento coletivo de uma corporação. O software é central para quase todo aspecto do negócio. E, de muitos modos, o software também é uma tecnologia oculta. Encontramos software (freqüentemente sem perceber) quando viajamos em serviço, quando fazemos qualquer compra no varejo, quando passamos pelo banco, quando damos um telefonema, quando consultamos o médico ou realizamos qualquer uma das centenas de atividades cotidianas que refletem a vida moderna.

A difusão do software nos leva a uma simples conclusão: sempre que uma tecnologia tem um amplo impacto — o impacto que pode salvar vidas ou colocá-las em perigo, construir negócios ou destruí-los, informar líderes governamentais ou confundi-los —, ela deve ser "tratada com cuidado".

> "Previsões são difíceis de fazer, especialmente quando elas tratam do futuro."
>
> **Mark Twain**

32.2 O Alcance da Modificação

As modificações na computação ao longo dos últimos 50 anos foram norteadas por avanços nas ciências exatas — física, química, ciência dos materiais, engenharia. Essa tendência continuará durante o primeiro quarto do século XXI. O impacto de novas tecnologias é penetrante – em comunicações, energia, cuidados com a saúde, transporte, entretenimento, economia, fabricação e guerras, para citar apenas algumas.

Info

Tecnologias a Observar

Os editores do *PC Magazine* [PCM03] preparam um número anual "Future Tech" "que [seleciona] entre todo palavreado (há muito disso) para identificar as 20 mais promissoras tecnologias do amanhã". As tecnologias mencionadas variam em ampla gama desde cuidados com a saúde até guerras. No entanto, é interessante notar que software e engenharia de software têm um papel significativo a desempenhar em todas elas, ou como um habilitador da tecnologia ou uma parte integral dela.

As seguintes representam uma amostra das tecnologias observadas:

Nanotubos de carbono — com uma estrutura fina semelhante a grafite, nanotubos de carbono podem servir como condutores para transmitir sinais de um ponto para outro e como transistores, usando mudança de sinal para guardar informação. Esses dispositivos são promissores para uso no desenvolvimento de dispositivos eletrônicos menores, mais rápidos, de menor energia e

Document shared on www.docsity.com

menos dispendiosos (por exemplo, microprocessadores, memórias, mostradores).

Biossensores — sensores microeletrônicos externos ou implantáveis já estão em uso para detectar tudo que se relaciona a agentes químicos no ar que respiramos em níveis sangüíneos, em um paciente cardíaco. À medida que esses sensores tornam-se mais sofisticados, podem ser implantados em pacientes médicos para monitorar uma variedade de condições relacionadas à saúde ou presas ao uniforme de um soldado para monitorar a presença de armas biológicas e químicas.

Mostradores OLED — Um OLED "usa uma molécula projetada com base em carbono que emite luz quando uma corrente elétrica passa por ela. Junte muitas moléculas e você obterá um mostrador superfino de qualidade espantosa — não é necessária nenhuma luz de fundo absorvedora de energia" [PCM03]. O resultado — mostradores ultrafinos que podem ser enrolados ou dobrados, pulverizados em uma superfície curva ou adaptado de outro modo a um ambiente específico.

Computação em malha — essa tecnologia (disponível atualmente) cria uma rede que aproveita os bilhões de ciclos de CPU não usados de cada máquina da rede e permite que tarefas de computação excessivamente complexas sejam completadas sem um supercomputador dedicado. Para um exemplo da vida real que reúne mais de 4,5 milhões de computadores visite o site http://setiathome.berkeley.edu/.

Máquinas cognitivas — o "Santo Graal" no campo da robótica é o desenvolvimento de máquinas que estão conscientes do seu ambiente, que podem "captar indícios, responder a situações permanentemente mutáveis e interagir com pessoas naturalmente" [PCM03]. Máquinas cognitivas estão ainda nos estágios iniciais de desenvolvimento, mas o potencial (se algum dia alcançado) é enorme.

Veja na Web

Para previsão sobre o futuro da tecnologia e outros assuntos, ver www.futurefacing.com.

No longo prazo, avanços revolucionários na computação podem ser norteados pelas ciências humanas — psicologia humana, sociologia, filosofia, antropologia e outras. O período de gestação das tecnologias de computação, que podem ser derivadas dessas disciplinas, é muito difícil de prever, mas as primeiras influências já começaram (por exemplo, as comunidades — uma construção antropológica de usuários que são um primeiro esforço em redes par a par).

A influência das ciências humanas pode ajudar a moldar a direção da pesquisa em computação nas ciências exatas. Por exemplo, o projeto de futuros computadores pode ser guiado mais pelo entendimento da fisiologia do cérebro do que pelo conhecimento de microeletrônica convencional.

No curto prazo, as modificações que vão afetar a engenharia de software ao longo da próxima década serão influenciadas por quatro fontes simultâneas: (1) as pessoas que fazem o trabalho, (2) o processo que elas aplicam, (3) a natureza da informação e (4) a tecnologia de computação subjacente. Nas seções seguintes, cada um desses componentes — pessoal, processo, informação e tecnologia — é examinado com mais detalhes.

32.3 Pessoas e a Maneira Como Elas Constroem Sistemas

O software necessário para sistemas de alta tecnologia torna-se mais e mais complexo com o passar dos anos, e o tamanho dos programas resultantes aumenta proporcionalmente. O rápido crescimento do tamanho do programa "médio" criaria poucos problemas para nós, se não fosse por um único fato: à medida que o tamanho do programa aumenta, o número de pessoas que têm de trabalhar nele também aumenta.

A experiência mostra que, à medida que o número de pessoas em uma equipe de projeto de software aumenta, a produtividade global do grupo pode sofrer. Um modo de contornar esse problema é criar várias equipes de engenharia de software, compartimentalizando conseqüentemente as pessoas em grupos de trabalho individuais. No entanto, enquanto o número de equipes de engenharia de software cresce, a comunicação entre elas se torna tão difícil e demorada quanto a comunicação entre indivíduos. Pior, a comunicação (entre indivíduos ou equipes) tende a ser ineficiente — isto é, gasta-se muito tempo transferindo muito pouco conteúdo de informação e, muito freqüentemente, informação importante "se perde nas frestas".

> "O choque futuro [é] a destruição e a desorientação perturbadoras que induzimos em indivíduos, sujeitando-os a muitas modificações em muito pouco tempo."
>
> **Alvin Toffler**

Se a comunidade de engenharia de software tiver que lidar efetivamente com o dilema de comunicação, a estrada adiante, para os engenheiros de software, é preciso incluir modificações radicais no modo pelo qual indivíduos e equipes comunicam-se uns com os outros. E-mail, sites Web e videoconferência centralizada são agora comuns como mecanismos para conectar um grande número de pessoas a uma rede de informação. A importância dessas ferramentas no contexto do trabalho de engenharia de software não pode ser por demais enfatizada. Com um sistema efetivo de correio eletrônico ou sistema de mensagem instântanea, o problema encontrado por um engenheiro de software em Nova York pode ser resolvido com a ajuda de um colega em Tóquio. Em um sentido muito real, sessões de conversa (*chat sessions*) bem focadas e grupos de interesse especializados tornam-se repositórios de conhecimento que permitem que a sabedoria coletiva de um grande número de tecnólogos seja concentrada em um problema técnico ou em um assunto gerencial.

O vídeo personaliza a comunicação. No seu ponto alto, permite que colegas em diferentes locais (ou em diferentes continentes) se "encontrem" regularmente. Mas o vídeo também fornece outro benefício. Pode ser usado como repositório de conhecimento sobre o software e para treinar recém-admitidos a um projeto.

AVISO

Mais e mais "não programadores" estão construindo suas próprias "pequenas" aplicações. É provável que essa tendência crescente se acelere no futuro. Esses "civis" devem aplicar a tecnologia discutida neste livro? Provavelmente, não. Mas eles deveriam adotar uma filosofia ágil de engenharia de software, mesmo se não adotarem a prática.

> "A resposta artística adequada à tecnologia digital é acolhê-la como uma nova janela para tudo o que é eternamente humano e usá-la com paixão, sabedoria, coragem e alegria."
>
> **Ralph Lonbreglia**

A evolução de agentes inteligentes vai modificar também os padrões de trabalho de um engenheiro de software, ampliando enormemente a capacidade de ferramentas de software. Agentes inteligentes melhorarão a capacidade do engenheiro de verificar produtos de trabalhos de engenharia usando conhecimento específico do domínio, realizando tarefas burocráticas, fazendo pesquisa orientada e coordenando a comunicação homem a homem.

Finalmente, a aquisição de conhecimento está mudando de modo profundo. Na Internet, um engenheiro de software pode subscrever um grupo de interesse especial que enfoca áreas de tecnologia de interesse imediato. Uma questão colocada em um grupo de interesse provoca respostas de outras partes interessadas de todo o mundo. A World Wide Web fornece a um engenheiro de software a maior biblioteca de trabalhos e relatórios de pesquisa, tutoriais, comentários e referências sobre engenharia de software do mundo.

Se a história passada for uma indicação, é razoável dizer que as pessoas em si não mudarão. No entanto, os modos pelos quais elas se comunicam, o ambiente no qual elas trabalham, a disciplina que aplicam e, conseqüentemente, a cultura geral do desenvolvimento de software se modificarão de modo significativo e até mesmo profundo.

32.4 O "Novo" Processo de Engenharia de Software

É razoável caracterizar as duas primeiras décadas de prática de engenharia de software como a era do "pensamento linear". Influenciada pelo modelo clássico de ciclo de vida, a engenharia de software era abordada como uma atividade linear, na qual uma série de passos seqüenciais podia ser aplicada em um esforço para resolver problemas complexos. No entanto, abordagens lineares para o desenvolvimento do software vão contra o modo pelo qual a maioria dos sistemas são realmente construídos. Na realidade, sistemas complexos evoluem iterativamente, até mesmo incrementalmente. É por essa razão que um grande segmento da comunidade de engenharia de software está mudando para modelos ágeis e incrementais de desenvolvimento de software.

Modelos de processo ágil e incremental reconhecem que a incerteza domina a maioria dos projetos, que os prazos são com freqüência impossivelmente curtos e que a iteração possibilita a

Document shared on www.docsity.com

entrega de uma solução parcial, mesmo quando não é possível um produto completo dentro do prazo estabelecido. Modelos evolutivos enfatizam a necessidade de produtos de trabalho incremental, análise de risco, planejamento e depois revisão de planos e realimentação por parte do cliente. Em muitas instâncias, a equipe de software aplica um "manifesto ágil" (Capítulo 4) que enfatiza "indivíduos e interações acima de processos e ferramentas; trabalho no software acima de documentação abrangente; colaboração com o cliente acima de negociação de contrato e resposta à modificação acima de seguimento de um plano" [BEC01].

> "O melhor preparativo para um bom trabalho amanhã é fazer um bom trabalho hoje."
>
> **Elbert Hubbard**

Tecnologias de objeto, acopladas à engenharia de software baseada em componentes (Capítulo 30), são um crescimento natural da tendência em direção a modelos de processo incrementais e evolutivos. Ambas terão um profundo impacto na produtividade do desenvolvimento de software e na qualidade do produto. O reuso de componentes fornece benefícios imediatos e atraentes. Quando o reuso é acoplado a ferramentas CASE para prototipação da aplicação, incrementos de programa podem ser construídos muito mais rapidamente do que pelo uso de abordagens convencionais. A prototipação traz o cliente para o processo. Conseqüentemente, é provável que clientes e usuários tornem-se muito mais envolvidos no desenvolvimento de software. Isso, por sua vez, pode levar a maior satisfação do usuário final e a melhor qualidade global do software.

O rápido crescimento das tecnologias de redes e multimídia (por exemplo, o aumento exponencial de WebApps ao longo da última década) está modificando tanto o processo de engenharia de software quanto os seus participantes. Novamente, encontramos um paradigma ágil e incremental que enfatiza imediatismo, segurança e estética, bem como maior preocupação com a engenharia de software convencional. Modernas equipes de software (por exemplo, equipe de engenharia da Web) freqüentemente combinam tecnólogos com especialistas em conteúdo (por exemplo, artistas, músicos, videográficos) para construir uma fonte de informação para uma comunidade de usuários, que não só é grande como imprevisível. O software que surgiu do trabalho desses tenologistas já resultou em radicais modificações econômicas e culturais. Contudo, apesar de os conceitos e princípios básicos discutidos neste livro serem aplicáveis, o processo de engenharia de software precisa ser adaptado.

32.5 Novos Modos de Representar Informação

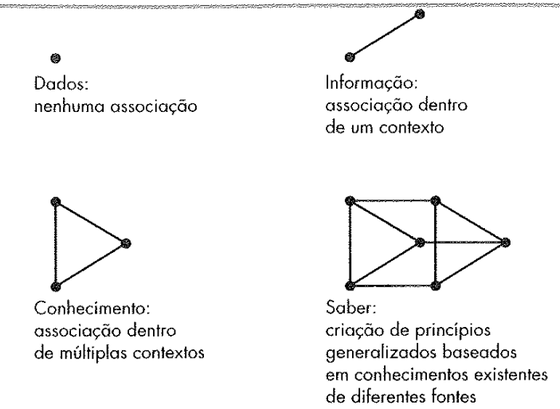
Ao longo da história da computação, uma transição sutil ocorreu na terminologia usada para descrever o trabalho de desenvolvimento do software realizado para a comunidade de negócios. Há 40 anos, o termo *processamento de dados* era a frase operativa para descrever o uso de computadores em um contexto de negócios. Hoje, processamento de dados deu lugar a outra frase — *tecnologia da informação* — que implica a mesma coisa, mas apresenta um deslocamento de foco sutil. A ênfase não está meramente em processar grandes quantidades de dados, mas sim em extrair informação significativa desses dados. Obviamente, o objetivo foi sempre esse, mas a mudança de terminologia reflete uma mudança muito mais importante na filosofia gerencial.

Quando aplicações de software são discutidas atualmente, as palavras *dados* e *informação* ocorrem repetidamente. Encontramos a palavra *conhecimento* em algumas aplicações de inteligência artificial, mas seu uso é relativamente raro. Claramente, ninguém discute *saber* no contexto de aplicações de software.

Dados são informação bruta — coleções de fatos que precisam ser processados para serem significativos. Informação é derivada da associação de fatos em um dado contexto. Conhecimento associa informação obtida em um contexto com outra informação obtida em um contexto diferente. Finalmente, saber ocorre quando princípios generalizados são derivados de conhecimentos díspares. Cada uma dessas quatro visões de "informação" é representada esquematicamente na Figura 32.1.

FIGURA 32.1

Um espectro de "informação"



Até hoje, a grande maioria de todo o software foi construída para processar dados ou informação. Engenheiros de software estão agora igualmente preocupados com sistemas que processam conhecimento[1]. Conhecimento é bidimensional. Informação, coletada de uma variedade de tópicos relacionados e não-relacionados, é conectada para formar um corpo de fatos que chamamos de *conhecimento*. A chave é a nossa capacidade de associar informação de uma variedade de fontes diferentes, que podem não ter nenhuma conexão óbvia, e combiná-las de um modo que nos forneça algum benefício distinto.

> "Sabedoria é o poder que nos permite usar conhecimento para benefício próprio e de outro."
>
> **Thomas J. Watson**

Para ilustrar a progressão de dados para conhecimento, considere os dados do senso que indicam que a taxa de natalidade em 1996, nos Estados Unidos, era de 4,9 milhões. Esse número representa um valor de dados. Relacionando essa amostra de dados com as taxas de natalidade dos 40 anos precedentes, podemos derivar uma amostra de informação útil — os *baby boomers* dos anos 1950 e início dos anos 1960 que estavam envelhecendo fizeram um último esforço para ter filhos antes do fim dos seus anos de fertilidade. Além disso, os da "geração X" começaram seus anos de fertilidade. Os dados do censo foram então conectados a outras amostras de informação aparentemente não-relacionadas. Por exemplo, o número atual de professores de escolas primárias que vão se aposentar na próxima década, o número de estudantes universitários que estão sendo graduados em educação primária e secundária, a pressão dos políticos para manter os impostos baixos e, conseqüentemente, limitar aumentos de salários para os professores.

Todas essas amostras de informação podem ser combinadas para formular uma representação de conhecimento — haverá pressão significativa no sistema educacional dos Estados Unidos na primeira década do século XXI e essa pressão vai continuar por mais de uma década. Usando esse conhecimento, uma oportunidade de negócio pode surgir. Pode haver uma oportunidade significativa de desenvolver novos modos de aprendizado, que sejam mais efetivos e menos dispendiosos que as abordagens atuais.

A estrada adiante do software leva a sistemas que processam conhecimento. Temos processado dados usando computadores por mais de 50 anos e extraído informação por mais de três décadas. Um dos desafios mais significativos com que se depara a comunidade de engenharia de software é construir sistemas que dêem o passo seguinte ao longo do espectro — sistemas que extraiam conhecimento, de dados e de informação, de um modo que seja prático e benéfico.

1 O crescimento rápido das tecnologias de mineração de dados (*data mining*) e armazéns de dados (*data warehousing*) reflete essa crescente tendência.

Document shared on www.docsity.com

32.6 Tecnologia como Condutora

O pessoal que constrói e usa software, o processo de engenharia de software que é aplicado e a informação que é produzida, são todos afetados pelos avanços na tecnologia de software e hardware. Historicamente, o hardware tem servido como condutor da tecnologia em computação. Uma tecnologia nova de hardware oferece potencial. Construtores de software reagem então às demandas dos clientes, em uma tentativa de esgotar o potencial.

Na estrada adiante para a tecnologia de hardware é provável que o progresso seja feito ao longo de dois caminhos paralelos. Ao longo de um caminho as tecnologias de hardware vão continuar a evoluir a passos rápidos. Com maior capacidade fornecida pelas arquiteturas tradicionais de hardware, as demandas sobre os engenheiros de software vão continuar a crescer.

Mas as reais modificações na tecnologia de hardware podem ocorrer ao longo de outro caminho. O desenvolvimento de arquiteturas de hardware não-tradicionais (por exemplo, nanotubos de carbono, microprocessadores EUL, máquinas cognitivas, computação em malha) pode causar modificações radicais na espécie de software que construímos e modificações fundamentais na nossa abordagem para a engenharia de software. Como essas abordagens não-tradicionais estão amadurecendo somente agora, é difícil determinar qual terá impacto mais amplo e mesmo mais difícil de prever como o mundo do software vai mudar para acomodá-las.

A estrada adiante para a engenharia de software é direcionada por tecnologias de software. Reuso e a engenharia de software baseada em componentes oferecem a melhor oportunidade para aumentos de ordem de grandeza na qualidade do sistema e no prazo de colocação no mercado. De fato, com o passar do tempo, o negócio de software pode começar a parecer muito com o negócio de hardware atual. Pode haver fornecedores que construam dispositivos discretos (componentes reusáveis de software), outros fornecedores que construam componentes de sistema (por exemplo, um conjunto de ferramentas para interação humano/computador) e integradores de sistema que forneçam soluções (produtos e sistemas construídos sob encomenda) para o usuário final.

A engenharia de software vai mudar — disso podemos estar certos. Mas, independentemente de quão radicais sejam essas mudanças, podemos garantir que a qualidade nunca vai perder sua importância, e que a análise e o projeto efetivos e teste competente terão sempre um lugar no desenvolvimento de sistemas baseados em computador.

INFO

Tendências de Tecnologia

J. P. Cripwell Associates (www.jpcripwell.com), uma firma de consultoria especializada em gestão de conhecimento e engenharia de informação, discute cinco impulsionadores de tecnologias que vão influenciar as direções da tecnologia nos próximos anos.

Combinação de tecnologias. Quando duas importantes tecnologias são combinadas, o impacto do resultado da combinação é freqüentemente maior do que a soma do impacto de cada uma separadamente. Por exemplo, a tecnologia de satélite GPS, acoplada à capacidade de computação a bordo, acoplada a tecnologias de mostradores LCD, resultou em sistemas sofisticados de mapeamento automobilístico. Tecnologias freqüentemente evoluem ao longo de caminhos separados, mas impacto significativo em negócio ou na sociedade ocorre somente quando alguém as combina para resolver um problema.

Fusão de dados. Quanto mais dados adquirimos, de mais dados necessitamos. Mais importante, quanto mais dados adquirimos, mais difícil é extrair informação útil.

De fato, freqüentemente precisamos adquirir ainda mais dados para entender (1) quais dados são importantes, que dados são relevantes para uma necessidade ou fonte específica e quais dados devem ser usados para tomada de decisão. Esse é o problema de fusão de dados. J. P. Cripwell usa um sistema avançado de monitoração de tráfego de automóvel como exemplo. Sensores digitais de velocidade (na estrada) e câmeras digitais sensoriam um acidente. A severidade do acidente precisa ser determinada (via câmera?). Baseado na severidade o sistema de monitoração deve contatar polícia, bombeiro ou ambulância; o tráfego deve ser desviado; mídia (rádio) precisa divulgar alertas e carros individuais (se equipados com sensores digitais ou comunicação sem fio) precisam ser informados. Para conseguir isso, uma variedade de decisões baseadas nos dados adquiridos do sistema de monitoração (fusão de dados) deve ser tomada.

Empuxo tecnológico. Anos atrás, um problema surgiu e foi desenvolvida tecnologia para resolvê-lo. Como o problema era evidente para muitas pessoas, o mercado para nova tecnologia estava bem definido. Hoje em dia, algumas tecnologias evoluem como soluções à procura de problemas. Um mercado deve ser forçado a reconhecer que precisa da nova tecnologia (por exemplo, telefones móveis, PDAs). À medida que as pessoas reconhecem a necessidade, a tecnologia acelera, melhora e freqüentemente se transforma à medida que combinações de tecnologias evoluem.

Redes e descobertas acidentais: Neste contexto redes implicam conexões entre pessoas ou entre pessoas e informação. À medida que a rede cresce, a probabilidade de sinergia entre dois nós da rede (por exemplo, pessoas e fontes de informação) também cresce. Uma conexão acidental (descoberta por acaso) pode levar à inspiração e a uma nova tecnologia ou aplicação.

Sobrecarga de informação. Um vasto mar de informação está acessível a quem quer que tenha uma conexão com a Internet. O problema, naturalmente, é encontrar a informação correta, determinar sua validade e depois traduzi-la em aplicação prática no nível de negócio ou pessoal.

32.7 As Responsabilidades do Engenheiro de Software

Veja na Web

Uma discussão completa do código de ética da ACM/IEEE pode ser encontrada em seeri.etsu.edu/Codes/default.shtm.

Engenharia de software evoluiu para uma profissão respeitada mundialmente. Como profissionais, engenheiros de software devem ater-se a um código de ética que guia o trabalho que fazem e os produtos que produzem. Uma força-tarefa conjunta da ACM/IEEE-CS (ACM/IEEE-CS Joint Task Force) produziu um código de ética de engenharia de software e práticas profissionais *Software Engineering Code of Ethics and Professional Practices* (Versão 5.1). O código declara:

> Engenheiros de software devem se dedicar a fazer da análise, especificação, projeto, desenvolvimento, teste e manutenção de software uma profissão benéfica e respeitada. De acordo com essa dedicação à saúde, à segurança e ao bem-estar do público, engenheiros de software devem adotar os oito princípios a seguir:
>
> 1. PÚBLICO — Engenheiros de software devem agir consistentemente com o interesse público.
>
> 2. CLIENTE E EMPREGADOR — Engenheiros de software devem agir de um modo que seja do melhor interesse de seus clientes e empregadores e consistente com o interesse público.
>
> 3. PRODUTO — Engenheiros de software devem garantir que seus produtos e modificações relacionadas atendam às melhores normas profissionais possíveis.
>
> 4. JULGAMENTO — Engenheiros de software devem manter integridade e independência em seu julgamento profissional.
>
> 5. GESTÃO — Gerentes e líderes de engenharia de software devem subscrever e promover uma abordagem ética à gestão do desenvolvimento e manutenção de software.
>
> 6. PROFISSÃO — Engenheiros de software devem avançar na integridade e reputação da profissão, de forma consistente com o interesse público.
>
> 7. COLEGAS — Engenheiros de software devem ser justos e colaborar com seus colegas.
>
> 8. INDIVIDUAL — Engenheiros de software devem participar de aprendizado vitalício com respeito à prática da sua profissão e promover uma abordagem ética da prática profissional.

Apesar de cada um desses oito princípios serem igualmente importantes, surge um tema preponderante: um engenheiro de software deve trabalhar para o interesse público. No nível pessoal, o engenheiro de software deve respeitar as seguintes regras:

- Nunca roubar dados para ganho pessoal.
- Nunca distribuir ou vender informação que tenha um proprietário e que fora obtida como parte de seu trabalho em um projeto de software.
- Nunca destruir maliciosamente ou modificar programas, arquivos ou dados de outra pessoa.
- Nunca violar a privacidade de um indivíduo, grupo ou organização.

Document shared on www.docsity.com

- Nunca invadir um sistema por esporte ou lucro.
- Nunca criar ou disseminar um vírus ou verme de computador.
- Nunca usar tecnologia de computação para facilitar a discriminação ou o assédio.

Durante a década passada, alguns membros da indústria de software empenharam-se na luta por legislação protecionista que [SEE03]:

1. Permita a empresas entregar software sem revelar defeitos conhecidos;
2. Isente os desenvolvedores de responsabilidade por quaisquer danos resultantes desses defeitos conhecidos;
3. Cerceie outros de revelar defeitos sem permissão do desenvolvedor original;
4. Permita a incorporação de software de "efeito retardado" em um produto que pode danificar (via comando remoto) a operação do produto;
5. Isente desenvolvedores de software com "efeito retardado" de danos se o software for danificado por terceiros.

Como toda legislação, o debate freqüentemente se concentra em tópicos que são políticos, não tecnológicos. No entanto, muitas pessoas (inclusive este autor) acham que legislação protecionista, se inadequadamente redigida, conflita com o código de ética de engenharia de software pela isenção indireta de engenheiro de software da sua responsabilidade de produzir software de alta qualidade.

32.8 Conclusão

Vinte e cinco anos se passaram desde que a primeira edição deste livro foi escrita. Ainda lembro de estar sentado à minha mesa como jovem professor, escrevendo o manuscrito (à mão) de um livro sobre um assunto com o qual poucas pessoas se preocupavam e ainda menos realmente entendiam. Lembro das cartas de rejeição de editores, que alegavam (delicada, mais firmemente) que nunca haveria mercado para um livro sobre "engenharia de software". Felizmente, a McGraw-Hill decidiu dar-lhe uma oportunidade[2], e o resto, como se diz, é história.

Ao longo dos últimos 25 anos este livro mudou dramaticamente — em escopo, tamanho, estilo e conteúdo. Como a engenharia de software, ele cresceu e (espero) amadureceu ao longo dos anos.

Uma abordagem de engenharia para o desenvolvimento de software de computador é hoje bagagem convencional. Apesar de o debate sobre o "paradigma correto" continuar, a importância da agilidade, o grau de automação e métodos mais efetivos, os princípios subjacentes da engenharia de software são agora aceitos por toda a indústria. Por que então só recentemente estamos vendo sua ampla adoção?

A resposta, penso, está na dificuldade de transição da tecnologia e da modificação cultural que a acompanha. Apesar de a maioria de nós apreciar a necessidade de uma disciplina de engenharia para software, lutamos contra a inércia da prática anterior e nos defrontamos com novos domínios de aplicação (e os desenvolvedores que neles trabalham) que parecem prontos a repetir os erros do passado.

Para facilitar a transição, precisamos de muitas coisas — um processo de software ágil, adaptável e sensível, métodos mais efetivos, ferramentas mais poderosas, melhor aceitação por parte dos profissionais e apoio por parte dos gerentes, e uma dose não pequena de educação e "propaganda". A engenharia de software não tem tido o benefício de uma propaganda massiva, mas com o passar do tempo o conceito se vende por si só. De certo modo, este livro é uma "propaganda" para a tecnologia.

Você pode não concordar com todas as abordagens descritas neste livro. Algumas das técnicas e opiniões são controversas; outras precisam ser ajustadas para funcionar bem em diferentes ambientes de desenvolvimento de software. No entanto, é meu desejo sincero que *Engenharia de Software* tenha delineado o problema com que deparamos, demonstrado a força dos conceitos de engenharia de software e fornecido um arcabouço para métodos e ferramentas.

2 Na verdade, o crédito deve ser dado a Peter Freeman e Eric Munson, que convenceram a McGraw-Hill de que valia a pena a tentativa.

Ao avançarmos no século XXI, o software tornou-se o mais importante produto e a mais importante indústria no cenário mundial. Seu impacto e sua importância percorreram um caminho bastante longo. E, no entanto, uma nova geração de desenvolvedores de software deve superar muitos dos mesmos desafios com os quais as gerações anteriores se defrontaram. Vamos esperar que as pessoas que enfrentam o desafio — engenheiros de software — tenham a sabedoria de desenvolver sistemas que melhorem a condição humana.

Referências Bibliográficas

[ACM98] ACM/IEEE-CS Joint Task Force, *Software Engineering Code of Ethics and Professional Practice*, 1998, disponível em http://www.acm.org/serving/se/code.htm.

[BEC01] Beck, K. *et al.*, "Manifesto for Agile Software Development", disponível em http://www.agilemanifesto.org/.

[BOL91] Bollinger, T. e McGowen, C., "A Critical Look at Software Capability Evaluations", *IEEE Software*, jul. 1991, p. 25-41.

[GIL96] Gilb, T., "What Is Level Six?", *IEEE Software*, jan. 1996, p. 97-98, 103.

[HOP90] Hopper, M. D., "Rattling SABRE, New Ways to Compete on Information", *Harvard Business Review*, maio/jun. 1990.

[PAU93] Paulk, M. *et al.*, *Capability Maturity Model for Software*, Software Engineering Institute, Carnegie Mellon University, 1993.

[PCM03] "Technologies to Watch", *PC Magazine*, jul. 2003, disponível em http://www.pcmag.com/article2/0,4149,1130591,00.asp.

[PRE91] Pressman, R. S. e Herron, S. R., *Software Shock*, Dorset House, 1991.

[SEE03] The Software Engineering Ethics Research Institute, "UCITA Updates", 2003, disponível em http://seeri.etsu.edu/default.htm.

Problemas e Pontos a Considerar

32.1. Adquira um exemplar dos principais periódicos sobre negócios e notícias (por exemplo, *Newsweek*, *Time*, *Business Week*). Liste todos os artigos ou notícias que podem ser usados para ilustrar a importância do software.

32.2. Um dos mais relevantes domínios de aplicação de software é o de sistemas e aplicações baseadas na Web. Discuta como pessoas, comunicação e processo precisam evoluir para acomodar o desenvolvimento de WebApps de "próxima geração".

32.3. Escreva uma descrição resumida de um ambiente de desenvolvimento ideal para um engenheiro de software por volta de 2010. Descreva os elementos do ambiente (hardware, software e tecnologias de comunicação) e seu impacto na qualidade e no prazo de colocação no mercado.

32.4. Reveja a discussão dos modelos de processos ágil e incremental no Capítulo 4. Faça uma pesquisa e reúna trabalhos recentes sobre o assunto. Resuma os pontos fortes e fracos dos paradigmas evolutivos, baseado nas experiências relatadas nos trabalhos.

32.5. Tente desenvolver um exemplo que comece com a coleta de dados de forma bruta e leve à aquisição de informação, de conhecimento e, finalmente, de saber.

32.6. Selecione uma tecnologia atualmente "quente" (não precisa ser uma tecnologia de software) que esteja sendo discutida na mídia popular e descreva como o software capacita sua evolução e seu impacto.

Leituras e Fontes de Informação Adicionais

Livros que discutem a estrada adiante para o software e para a computação abrangem uma vasta matriz de assuntos técnicos, científicos, econômicos, políticos e sociais. Sterling (*Tomorrow Now*, Random House, 2002) lembra-nos de que o progresso real é raramente ordenado e eficiente. Teich (*Technology and the Future*, Wadworth, 2002) apresenta cuidadoso ensaio sobre o impacto social de tecnologia e como mudanças culturais influenciam a tecnologia. Naisbitt, Philips e Naisbitt (*High Tech/High Touch*, Nicholas Brealey, 2001) observam que muitos de nós estaremos "embebedados" com a alta tecnologia e que "a grande ironia da era de alta tecnologia é que ficamos escravizados pelos dispositivos que nos deveriam dar liberdade". Zey (*The Future Factor*, McGraw-Hill, 2000) discute cinco forças que vão moldar o destino humano neste século. Canton (*Technofutures*, Hay House, 1999) discute como a tecnologia vai transformar negócios no século XXI. Robertson (*The New Renaissance: Computers and the Next Level*

Document shared on www.docsity.com

of Civilization, Oxford University Press, 1998) alega que a revolução do computador pode ser o avanço singular mais significativo da história da civilização.

Broderick (*Spike,* Forge, 2001) discute o impacto de tecnologias emergentes. Dertrouzos e Gates (*What Will Be: How the New World of Information Will Change Our Lives,* Harper-Business, 1998) fornecem uma discussão cuidadosa de algumas das direções que as tecnologias de informação podem tomar nas primeiras décadas deste século. Barnatt (*Valueware: Technology, Humanity and Organization,* Praeger Publishing, 1999) apresenta uma discussão interessante de uma "economia de idéias", e como valores econômicos serão criados à medida que negócios cibernéticos evoluem. Negroponte (*Being Digital,* Alfred A. Knopf, 1995) era um campeão de venda na metade da década de 1990 e continua a fornecer uma interessante visão de computação e seus impactos globais.

Kroker e Kroker (*Digital Delirium,* New World Perspectives, 1997) editaram uma coleção controversa de ensaios, poemas e humor que examina o impacto das tecnologias digitais sobre pessoas e sociedade. Brin (*The Transparent Society: Will Technology Force Us to Choose Between Privacy and Freedom?,* Perseus Books, 1999) revisita o contínuo debate associado à inevitável perda de privacidade pessoal que acompanha o crescimento das tecnologias de informação. Shenk (*Data Smog: Surviving the Information Glut,* HarperCollins, 1998) discute os problemas associados com a "sociedade infestada de informação", que está sendo sufocada pelo volume de informação que o software produz.

Brockman (*The Next Fifty Years,* Vintage Books, 2002) e Miller e seus colegas (*21st Century Technologies: Promises and Perils of a Dynamic Future,* Brookings Institution Press, 1999) editaram uma coleção de artigos e ensaios sobre o impacto da tecnologia sobre as estruturas sociais, econômicas e de negócios. Para os interessados em tópicos técnicos, Luryi, Xu e Zaslavsky (*Future Trends in Microelectronics,* Wiley, 1999) editaram uma coleção de artigos sobre as prováveis direções do hardware de computador com ênfase em nanotecnologias. Hayzelden e Bigham (*Software Agents for Future Communication Systems,* Springer-Verlag, 1999) editaram uma coleção que discute as tendências no desenvolvimento de agentes inteligentes de software.

À medida que o software se torna virtualmente parte de cada faceta de nossas vidas, "a cibernética" evolui como um importante tópico de discussão. Livros de Spinello (*Cyberethics: Morality and Law in Cyberspace,* Jones & Bartlett Publishers, 2002), Halbert e Ingulli (*Cyberethics,* South-Western College Publishers, 2001), e Baird e seus colegas (*Cyberethics: Social and Moral Issues in the Computer Age,* Prometheus Books, 2000) consideram o tópico em detalhe. O governo americano publicou volumoso relatório em CD-ROM (*21st Century Guide to Cybercrime,* Progressive Management, 2003) que considera todos os aspectos do crime por computador, tópicos de propriedade intelectual e o National Infrastructure Protection Center (NIPC).

Kurzweil (*The Age of Spiritual Machines, When Computers Exceed Human Intelligence,* Viking/Penguin Books, 1999) argumenta que dentro de 20 anos, a tecnologia de hardware terá a capacidade de modelar completamente o cérebro humano. Borgmann (*Holding on to Reality: The Nature of Information at the Turn of the Millennium,* University of Chicago Press, 1999) escreveu uma interessante história da informação, salientando seu papel na transformação da cultura. Devlin (*InfoSense: Turning Information into Knowledge,* W. H. Freeman & Co., 1999) tenta interpretar o constante fluxo de informação que nos bombardeia diariamente. Gleick (*Faster: The Acceleration of Just About Everything,* Pantheon Books, 2000) discute a taxa sempre em aceleração da modificação tecnológica e seu impacto em todos os aspectos da vida moderna. Jonscher (*The Evolution of Wired Life: From the Alphabet to the Soul-Catcher Chip – How Information Technologies Change Our World,* Wiley, 2000) argumenta que o pensamento e a interação humana transcendem a importância da tencologia.

Uma ampla variedade de fontes de informação sobre rumos futuros das tecnologias relacionadas a software e engenharia de software está disponível na Internet. Uma lista atualizada de referências da World Wide Web pode ser encontrada no site deste livro: **http://www.mhhe.com/pressman**.

ÍNDICE ANALÍTICO

A

Abstração, 190-191
Acessibilidade, 283
Acomodação das estimativas, 532
Acompanhamento de bugs, 608
Acompanhamento de dependência, 606
Acompanhamento de requisitos, 606
Acoplamento, 196, 363
  categorias de, 248
  métricas, 366, 368
Agilidade, 59. *Ver também* processo ágil
  definição de, 61
  fatores humanos, 61
  princípios, 60
Airlie Council, 495
Ambiente de trabalho, 277
Análise, 137. *Ver também* Análise de requisitos
  orientada a objetos, 152
  padrões, 137
  regras práticas, 146
Análise de custo, 676
  processo, 665
  questões econômicas, 675
Análise de domínio, 147
Análise de fluxo de trabalho, 275
Análise de imprevistos, 573
Análise de inventário, 686
Análise de Relacionamento-Navegação (RNA), 421
Análise de requisitos. *Ver também* Análise, 145
  engenharia da Web, 410
  objetivos de, 145
Análise de tarefas, 272
Análise de valor limite, 329, 469
Análise do valor agregado, 557
Análise gramatical, 161
Análise por árvore de decisão, 539
Aperfeiçoamento do processo de software (SPI), 500
  estatístico, 502
Aplicações da Web. *Ver* WebApp
Armazém de dados, 209
Arquétipos, 218
Arquitetura da aplicação, 105
  software de aplicação, 6
Arquitetura de conteúdo, 440
  estruturas, 440
Arquitetura de dados, 105
Arquitetura de Negociador de Solicitação entre Objetos, 671
Arquitetura, 191
  análise de sensibilidade, 222
  de dados, 105
  descrição de, 208
  importância de, 209
  MVC, 443
  WebApps, 442
Árvore de dados, 415
Árvore de requisitos de qualidade, WebApps, 427
Associações, 176
Atenuação de riscos, 571
Atividades de processo, 18-19
  genérico, 19
  PSP, 29
  TSP, 30
Atividades guarda-chuva, 18, 20
Ator(es), 130, 155-156
Atributos, 168
Atributos de dados, 150
Auditoria de configuração, 612
  WebApps, 619
Autenticação, 474
Auto-organização, 62
Autoridade de controle de modificação, 609
Avaliação do risco, 565, 569

B

Bloqueadores contra ataques, 474
Bugs, 583

C

Cálculo de, 568
Caminho crítico, 353
Caminhos de ação, 231
Caminhos de navegação (WoN), 444
Caminhos independentes, 320
Caos, 37
Característica, definição em Desenvolvimento Guiado por Características (FDD), 71
Cardinalidade, 150, 176
Cartões CRC, 171
CasaSegura (quadros), 13, 44, 48, 66, 83, 107, 127, 128-129, 133, 136-137, 139-140, 154, 156-157, 164, 169-170, 175, 194-195, 196-197, 214, 222-223, 230, 244, 248, 249-250, 267, 273, 392-393, 305, 309-310, 317, 321-322, 337, 356, 365-366, 392, 401, 411, 434, 468, 490, 502, 511, 528, 557, 570, 588, 611
CASE. *Ver* Ferramentas
Caso de teste, 340
  derivação, 322
Casos de uso, 53, 272
  análise de tarefas, 272-273
  desenvolvimento de, 129
  escrita, 153
  identificação de eventos, 178
  métricas, 507
  WebApp, 394, 411, 417
Categorias de usuário, definição de, 392
Categorias, software de aplicação, 6
CBA IPI, 27
CBSE, 663
Cenários de usuário, 129

Document shared on www.docsity.com