

ACADEMIA

Accelerating the world's research.

Engenharia de software conceitos e práticas

Raquel Aguiar

Related papers

[Download a PDF Pack](#) of the best related papers ↗



[SWEBOk - traduzido](#)

Josimar Candido

[Engenharia Software 3Edicao sommerville \(1\)](#)

Gustavo Santos Leandro

[UNIVERSIDADE GAMA FILHO](#)

Eduardo Fernando



RAUL SIDNEI WAZLAWICK

ENGENHARIA DE **Software**

CONCEITOS E PRÁTICAS



Engenharia de Software

Conceitos e Práticas

Raul Sidnei Wazlawick



Preencha a **ficha de cadastro** no final deste livro
E receba gratuitamente informações sobre os
lançamentos e as promoções da Elsevier.

Consulte também nosso catálogo completo, últimos lançamentos
e serviços exclusivos no site
www.elsevier.com.br

Engenharia de Software

Conceitos e Práticas

Raul Sidnei Wazlawick

1^a edição



ELSEVIER

© 2013, Elsevier Editora Ltda.

Todos os direitos reservados e protegidos pela Lei nº 9.610, de 19/02/1998.

Nenhuma parte deste livro, sem autorização prévia por escrito da editora, poderá ser reproduzida ou transmitida sejam quais forem os meios empregados: eletrônicos, mecânicos, fotográficos, gravação ou quaisquer outros.

Coordenação editorial: Adriana Ayami Takimoto

Copidesque: Andrea Vidal

Revisão: Eloiza Mendes Lopes

Editoração Eletrônica: Thomson Digital

Elsevier Editora Ltda.

Conhecimento sem Fronteiras

Rua Sete de Setembro, 111 – 16º andar

20050-006 – Centro – Rio de Janeiro – RJ – Brasil

Rua Quintana, 753 – 8º andar

04569-011 – Brooklin – São Paulo – SP

Serviço de Atendimento ao Cliente

0800-0265340

sac@elsevier.com.br

ISBN: 978-85-352-6120-2

Nota: Muito zelo e técnica foram empregados na edição desta obra. No entanto, podem ocorrer erros de digitação, impressão ou dúvida conceitual. Em qualquer das hipóteses, solicitamos a comunicação ao nosso Serviço de Atendimento ao Cliente, para que possamos esclarecer ou encaminhar a questão.

Nem a editora nem o autor assumem qualquer responsabilidade por eventuais danos ou perdas a pessoas ou bens originados do uso desta publicação.

CIP-BRASIL. CATALOGAÇÃO-NA-FONTE
SINDICATO NACIONAL DOS EDITORES DE LIVROS, RJ

W372e

Wazlawick, Raul Sidnei, 1967-

Engenharia de software: conceitos e práticas / Raul Sidnei Wazlawick. - Rio de Janeiro:
Elsevier, 2013.

28 cm

ISBN 978-85-352-6084-7

1. Engenharia de software 2. Software - Desenvolvimento. I. Título.

13-0677.

CDD: 005.1

CDU: 004.41

30.01.13 01.02.13

042501



Sobre o Autor

Raul Sidnei Wazlawick é Professor Associado IV do Departamento de Informática e Estatística da Universidade Federal de Santa Catarina (UFSC), Bacharel em Ciência da Computação (UFSC, 1988), Mestre em Ciência da Computação pela Universidade Federal do Rio Grande do Sul (UFRGS, 1991) e Doutor em Engenharia (UFSC, 1993), com Pós-Doutorado em Informática pela Universidade Nova de Lisboa (1998). Desde 1986, trabalha na área de Engenharia de Software, sendo docente desde 1992 na UFSC e em cursos e palestras em dezenas de outras universidades e empresas no Brasil e no exterior. Trabalha também em projetos de pesquisa em consultoria em empresas para melhoria de processos de desenvolvimento de software. Foi membro do Conselho da Sociedade Brasileira de Computação (SBC), coordenador dos cursos de Bacharelado e Mestrado em Ciência da Computação da UFSC, coordenador do XX Simpósio Brasileiro de Engenharia de Software (SBES 2006) e *chair* do Working Group 3.2 (*Higher Education*) da International Federation for Information Processing (IFIP). Foi também membro da Comissão de Especialistas em Ensino de Informática e Computação do Ministério da Educação (MEC). Orientou dezenas de dissertações de mestrado, teses de doutorado e trabalhos de graduação; tendo também publicado mais de uma centena de artigos científicos em eventos e periódicos internacionais. Recebeu, juntamente com sua orientanda Marília Guterres Ferreira, o prêmio Best Paper Award na Conferência Ibero-americana de Engenharia de Software em 2011. É bolsista de produtividade em desenvolvimento tecnológico e extensão inovadora pelo CNPq.





Prefácio

O SWEBOK¹ (*Software Engineering Book of Knowledge*), organizado pela IEEE Computer Society, define o corpo de conhecimentos usualmente aceito relacionado à Engenharia de Software (IEEE Computer Society, 2004). Esta publicação relaciona dez áreas de conhecimento, das quais sete são tratadas aqui neste livro (Teste de Software, Manutenção de Software, Gerenciamento de Configuração de Software, Gerenciamento de Engenharia de Software, Processo de Engenharia de Software, Ferramentas e Métodos de Engenharia de Software e Qualidade de Software), enquanto as outras três áreas (Requisitos de Software, *Design* de Software e Construção de Software) são tratadas no livro *Análise e Projeto de Sistemas de Informação Orientados a Objetos* (Wazlawick, 2011).

Este livro não possui um capítulo específico sobre ferramentas de Engenharia de Software, porque elas são mencionadas ao longo de todos os capítulos onde se fazem necessárias. As referências incluem, sempre que possível, produtos gratuitos e proprietários, bem como a referência para o *site* da empresa ou grupo responsável pela ferramenta.

As referências bibliográficas, também, sempre que disponíveis na Internet, têm sua URL referenciada no local em que aparecem no texto, de forma a facilitar uma eventual busca pelas fontes originais deste trabalho.

O livro não esgota o assunto relacionado à Engenharia de Software, mas os tópicos mais fundamentais estão detalhados de modo que o leitor consiga efetivamente usar as técnicas, e não apenas ouvir falar delas.

A profundidade com que os tópicos são abordados corresponde ao que o autor entende que seria adequado a um aluno de graduação em Sistemas de Informação ou Ciência da Computação, para que possa desempenhar a função de engenheiro de software adequadamente no mercado de trabalho.

O livro também pode ser de grande valia para profissionais que tenham interesse em se reciclar ou dar o passo inicial para implantar processos produtivos mais organizados em suas empresas.

O conteúdo deste livro pode ser abordado possivelmente em uma disciplina de 90 horas ou ainda em três disciplinas de 30 horas cada, em que cada disciplina abordaria uma das partes do livro: processo de engenharia de software, gerenciamento de projeto de software e qualidade de software. Uma quarta disciplina de cerca de 40 a 50 horas ainda poderia ser acrescentada com o uso do livro *Análise e Projeto de Sistemas de Informação Orientados a Objetos*, que complementa este livro com a apresentação de técnicas detalhadas para requisitos, análise, *design* e geração de código.

Não foram incluídos exercícios no livro porque estão disponibilizados no *site* da editora, bastando, para acessá-los, usar o código fornecido pelo próprio livro. Esta escolha visa, por um lado, permitir que o livro tenha um preço mais acessível, já que diminui o número total de páginas e, por outro lado, permite que novos exercícios possam ser dinamicamente adicionados pelo autor, à medida que são produzidos.

Os leitores são incentivados a entrarem em contato com o autor pelo email raul@inf.ufsc.br para apresentar sugestões de melhoria ao livro, pois, de acordo com as sugestões apresentadas no próprio livro, o *feedback* do usuário é fundamental para a criação de produtos de qualidade.

Este livro foi produzido em ciclos iterativos entre os anos de 2010 e 2012, a partir de diversas fontes bibliográficas e da experiência do autor na área, que iniciou em 1985 seus estudos e, até hoje, leciona, publica e presta consultoria na área de Engenharia de Software.

O livro pode ser lido do início ao fim, mas também pode ser lido em qualquer outra ordem, visto que os capítulos são autocontidos e eventuais referências a outros capítulos são feitas quando necessário.

Alguns detalhamentos relacionados a normas técnicas podem ser omitidos da leitura, e mantidos à mão para referência, mas, mesmo assim, foram mantidos no corpo do livro, visto que, apesar de extensos, tais trechos reforçam as boas práticas de Engenharia de Software que devem ser seguidas para que sejam desenvolvidos sistemas com qualidade.

Este livro é dedicado a todos aqueles que acreditam e trabalham para que o Brasil possa um dia ser referência mundial na área de produção de software com qualidade e criatividade. Vamos melhorar nossos processos de produção para que isso ocorra ainda nesta geração!

Ingleses do Rio Vermelho, distrito de Florianópolis, 16 de fevereiro de 2012.

p0075

¹www.computer.org/portal/web/swebok/html/contents

fn0010



Sumário

| | |
|--------------------------|-----|
| <i>Índice de Tabelas</i> | xix |
| <i>Índice de Figuras</i> | xxi |

| | |
|--|---------------|
| 1 Introdução | 1 |
| 1.1 A Crise dos Desenvolvedores de Software Menos Preparados | 1 |
| 1.2 Os Eternos Mitos | 2 |
| 1.3 (In)Definição de Engenharia de Software | 4 |
| 1.4 O Engenheiro de Software | 5 |
| 1.5 Evolução da Engenharia de Software | 6 |
| 1.6 Tipos de Software do Ponto de Vista da Engenharia | 6 |
| 1.7 Princípios da Engenharia de Software | 7 |
| PARTE 1: PROCESSO E MODELOS DE PROCESSO | 9 |
| 2 Processo | 11 |
| 2.1 Fases | 12 |
| 2.2 Disciplinas | 13 |
| 2.3 Atividades ou Tarefas | 13 |
| 2.3.1 Artefatos | 13 |
| 2.3.2 Responsáveis e Participantes | 14 |
| 2.3.3 Recursos | 14 |
| 2.4 Detalhamento de Atividades | 15 |
| 2.4.1 Passos | 15 |
| 2.4.2 Procedimentos | 15 |
| 2.4.3 Regras | 15 |
| 2.5 <i>Template</i> e Exemplo de Documento de Atividade | 15 |
| 2.6 Equipe de Processo | 17 |
| 2.7 Norma NBR ISO/IEC 12207 | 18 |
| 3 Modelos de Processo Prescritivos | 21 |
| 3.1 Codificar e Consertar | 23 |
| 3.2 Modelo Cascata | 25 |
| 3.3 <i>Sashimi</i> (Cascata Entrelaçado) | 30 |
| 3.4 Modelo V | 31 |

| | | |
|----------|--|-----------|
| 3.5 | Modelo W | 32 |
| 3.6 | Cascata com Subprojetos | 33 |
| 3.7 | Cascata com Redução de Risco | 34 |
| 3.8 | Modelo Espiral | 35 |
| 3.9 | Prototipação Evolucionária | 37 |
| 3.10 | Entregas em Estágios | 38 |
| 3.11 | Modelo Orientado a Cronograma | 39 |
| 3.12 | Entrega Evolucionária | 40 |
| 3.13 | Modelos Orientados a Ferramentas | 40 |
| 3.14 | Linhas de Produto de Software | 40 |
| 3.14.1 | Desenvolvimento do Núcleo de Ativos | 41 |
| 3.14.2 | Desenvolvimento do Produto | 43 |
| 3.14.3 | Gerência | 43 |
| 3.14.4 | Juntando Tudo | 44 |
| 4 | Modelos Ágeis | 45 |
| 4.1 | <i>FDD – Feature-Driven Development</i> | 46 |
| 4.1.1 | DMA – Desenvolver Modelo Abrangente | 47 |
| 4.1.2 | CLF – Construir Lista de Funcionalidades | 48 |
| 4.1.3 | PPF – Planejar por Funcionalidade | 49 |
| 4.1.4 | DPF – Detalhar por Funcionalidade | 50 |
| 4.1.5 | CPF – Construir por Funcionalidade | 51 |
| 4.1.6 | Ferramentas para FDD | 52 |
| 4.2 | <i>DSDM – Dynamic Systems Development Method</i> | 52 |
| 4.2.1 | Análise de Viabilidade | 53 |
| 4.2.2 | Análise de Negócio | 53 |
| 4.2.3 | Iteração do Modelo Funcional | 54 |
| 4.2.4 | Iteração de <i>Design</i> e Construção | 55 |
| 4.2.5 | Implantação | 55 |
| 4.2.6 | Papéis no DSDM | 55 |
| 4.3 | <i>Scrum</i> | 56 |
| 4.3.1 | Perfis | 56 |
| 4.3.2 | <i>Product Backlog</i> | 56 |
| 4.3.3 | <i>Sprint</i> | 57 |
| 4.3.4 | <i>Daily Scrum</i> | 59 |
| 4.3.5 | Funcionamento Geral do <i>Scrum</i> | 60 |
| 4.4 | <i>XP – eXtreme Programming</i> | 60 |
| 4.4.1 | Práticas XP | 62 |
| 4.4.2 | Regras de Planejamento | 63 |
| 4.4.3 | Regras de Gerenciamento | 64 |
| 4.4.4 | Regras de <i>Design</i> | 64 |



| | | |
|---|--|------------|
| 4.4.5 | Regras de Codificação | 65 |
| 4.4.6 | Regras de Teste | 66 |
| 4.5 | <i>Crystal Clear</i> | 66 |
| 4.5.1 | Entregas Frequentes | 68 |
| 4.5.2 | Melhoria Reflexiva | 68 |
| 4.5.3 | Comunicação Osmótica | 69 |
| 4.5.4 | Segurança Pessoal | 69 |
| 4.5.5 | Foco | 70 |
| 4.5.6 | Acesso Fácil a Especialistas | 71 |
| 4.5.7 | Ambiente Tecnologicamente Rico | 71 |
| 4.6 | ASD – <i>Adaptive Software Development</i> | 72 |
| 4.6.1 | Especular | 73 |
| 4.6.2 | Colaborar | 73 |
| 4.6.3 | Aprender | 73 |
| 4.6.4 | Inicialização e Encerramento | 73 |
| 4.6.5 | Juntando Tudo | 73 |
| 5 | UP – Processo Unificado | 75 |
| 5.1 | Caracterização do Processo Unificado | 76 |
| 5.1.1 | Dirigido por Casos de Uso | 76 |
| 5.1.2 | Centrado na Arquitetura | 76 |
| 5.1.3 | Iterativo e Incremental | 77 |
| 5.1.4 | Focado em Riscos | 77 |
| 5.2 | Fases do Processo Unificado | 77 |
| 5.2.1 | Concepção | 78 |
| 5.2.2 | Elaboração | 79 |
| 5.2.3 | Construção | 79 |
| 5.2.4 | Transição | 80 |
| 5.3 | RUP – <i>Rational Unified Process</i> | 80 |
| 5.3.1 | Os Blocos de Construção do RUP | 80 |
| 5.3.2 | Disciplinas | 86 |
| 5.4 | AUP – <i>Agile Unified Process</i> | 105 |
| 5.5 | OpenUp – <i>Open Unified Process</i> | 107 |
| 5.6 | EUP – <i>Enterprise Unified Process</i> | 108 |
| 5.7 | OUM – <i>Oracle Unified Method</i> | 109 |
| 5.8 | RUP-SE – <i>Rational Unified Process-Systems Engineering</i> | 111 |
| PARTE 2: PLANEJAMENTO E GERÊNCIA DE PROJETOS | | 113 |
| 6 | Planejamento | 115 |
| 6.1 | Seleção de Projetos | 115 |
| 6.2 | Termo de Abertura | 116 |

| | | |
|----------|--|------------|
| 6.3 | Declaração de Escopo | 116 |
| 6.4 | Planejamento de Projeto com Ciclos Iterativos | 117 |
| 6.4.1 | Estimação da Duração e Esforço nas Diferentes Fases do Projeto | 118 |
| 6.4.2 | Estimação da Duração e Número dos Ciclos Iterativos | 119 |
| 6.4.3 | Número de Iterações | 120 |
| 6.4.4 | Definição dos Marcos ou Entregas | 120 |
| 6.5 | Planejamento de Iteração | 121 |
| 6.5.1 | WBS – Estrutura Analítica da Iteração | 122 |
| 6.5.2 | Os Dez Mandamentos da WBS | 123 |
| 6.5.3 | Identificação dos Responsáveis por Atividade | 124 |
| 6.5.4 | Identificação dos Recursos Necessários e Custo | 124 |
| 6.5.5 | Identificação das Dependências entre Atividades | 124 |
| 6.5.6 | Cronograma | 126 |
| 7 | Estimativas de Esforço | 129 |
| 7.1 | SLOC e KSLOC | 130 |
| 7.1.1 | Estimação de KSLOC | 130 |
| 7.1.2 | Transformando Pontos de Função em KSLOC | 131 |
| 7.1.3 | Como Contar Linhas de Código | 132 |
| 7.2 | COCOMO | 134 |
| 7.2.1 | Modelo Básico | 135 |
| 7.2.2 | Modelo Intermediário | 136 |
| 7.2.3 | Modelo Avançado | 138 |
| 7.3 | COCOMO II | 138 |
| 7.3.1 | Fatores de Escala | 140 |
| 7.3.2 | Multiplicadores de Esforço | 143 |
| 7.3.3 | Aplicando COCOMO II para as fases do UP | 152 |
| 7.3.4 | Calibragem do Modelo | 153 |
| 7.3.5 | Questionário EPML | 154 |
| 7.4 | Pontos de Função | 156 |
| 7.4.1 | Interpretação e Classificação dos Requisitos como Funções | 158 |
| 7.4.2 | UFP – Pontos de Função Não Ajustados | 160 |
| 7.4.3 | AFP – Pontos de Função Ajustados | 162 |
| 7.4.4 | Duração e Custo de um Projeto | 164 |
| 7.4.5 | Detalhamento dos Fatores Técnicos | 165 |
| 7.5 | Pontos de Caso de Uso | 170 |
| 7.5.1 | UAW – Complexidade de Atores | 171 |
| 7.5.2 | UUCW – Complexidade dos Casos de Uso | 171 |
| 7.5.3 | UUCP – Pontos de Caso de Uso Não Ajustados | 172 |
| 7.5.4 | TCF – Fatores Técnicos | 172 |
| 7.5.5 | EF – Fatores Ambientais | 173 |



| | | |
|---|--|------------|
| 7.5.6 | UCP – Pontos de Caso de Uso Ajustados | 173 |
| 7.5.7 | Esforço | 173 |
| 7.5.8 | Métodos de Contagem para Casos de Uso Detalhados | 174 |
| 7.5.9 | FUCS – <i>Full Use Case Size</i> | 174 |
| 7.6 | Pontos de Histórias | 177 |
| 7.6.1 | Atribuição de Pontos de Histórias | 177 |
| 7.6.2 | Medição de Velocidade | 178 |
| 8 Riscos | | 179 |
| 8.1 | Plano de Gerência de Riscos | 180 |
| 8.2 | Identificação de Riscos | 181 |
| 8.2.1 | Riscos Tecnológicos | 182 |
| 8.2.2 | Riscos Relacionados a Pessoas | 182 |
| 8.2.3 | Riscos de Projeto | 182 |
| 8.3 | <i>Checklist</i> de Riscos | 182 |
| 8.4 | Análise de Riscos | 193 |
| 8.5 | Planos de Mitigação de Riscos | 194 |
| 8.5.1 | Plano de Redução de Probabilidade de Risco | 196 |
| 8.5.2 | Plano de Redução de Impacto de Risco | 197 |
| 8.6 | Plano de Contingência | 197 |
| 8.7 | Monitoramento de Riscos | 198 |
| 8.8 | Controle de Risco | 200 |
| 8.9 | Comunicação de Riscos | 201 |
| 9 Gerenciamento de Projeto de Software | | 203 |
| 9.1 | O Gerente de Projeto | 204 |
| 9.2 | Gerenciamento de Projetos segundo o PMBOK | 205 |
| 9.3 | Prince2 – <i>Projects IN Controlled Environments 2</i> | 207 |
| 9.4 | Condução de Projeto de Software | 209 |
| 9.4.1 | Folha de Tempo | 210 |
| 9.4.2 | Acompanhamento de Problemas | 211 |
| 9.4.3 | Registro de Artefatos | 212 |
| 9.5 | Medição em Engenharia de Software | 212 |
| 9.5.1 | Classificações de Métricas | 213 |
| 9.5.2 | Planejamento de um Programa de Métricas | 214 |
| 9.6 | Revisão e Avaliação | 215 |
| 9.7 | Fechamento | 216 |
| 10 Gerenciamento de Configuração e Mudança | | 219 |
| 10.1 | Conceitos Básicos | 220 |
| 10.1.1 | Item de Configuração de Software | 220 |

| | |
|---|------------|
| 10.1.2 Relacionamentos de Itens de Configuração de Software | 220 |
| 10.1.3 Rastreabilidade | 221 |
| 10.1.4 Versões de Itens de Configuração de Software | 221 |
| 10.1.5 Configuração de Software | 221 |
| 10.1.6 <i>Baseline e Release</i> | 222 |
| 10.2 Controle de Versão | 222 |
| 10.2.1 Repositório | 223 |
| 10.2.2 Políticas de Compartilhamento de Itens | 223 |
| 10.2.3 Envio de Versões | 224 |
| 10.3 Controle de Mudança | 224 |
| 10.4 Auditoria de Configuração | 225 |
| 10.5 Ferramentas para Controle de Versão | 225 |
| PARTE 3: QUALIDADE | 227 |
| 11 Qualidade de Produto | 229 |
| 11.1 Modelo de Qualidade SquaRE – ISO/IEC 25010:2011 | 229 |
| 11.1.1 Adequação Funcional | 232 |
| 11.1.2 Confiabilidade | 232 |
| 11.1.3 Usabilidade | 234 |
| 11.1.4 Eficiência de Desempenho | 234 |
| 11.1.5 Segurança | 235 |
| 11.1.6 Compatibilidade | 235 |
| 11.1.7 Capacidade de manutenção | 235 |
| 11.1.8 Portabilidade | 236 |
| 11.1.9 Qualidades do Software em Uso | 236 |
| 11.2 Modelo de Qualidade de Dromey | 236 |
| 11.3 Instalação de um Programa de Melhoria de Qualidade | 241 |
| 11.4 Gestão da Qualidade | 241 |
| 11.4.1 <i>Walkthrough</i> | 242 |
| 11.4.2 Inspeções Fagan | 243 |
| 11.4.3 Método <i>Cleanroom</i> | 244 |
| 11.5 Medição da Qualidade | 245 |
| 11.6 Requisitos de Qualidade | 246 |
| 11.7 GQM (<i>Goal/Question/Metric</i>) e Avaliação da Qualidade | 247 |
| 12 Qualidade de Processo | 251 |
| 12.1 ISO/IEC 90003 | 251 |
| 12.1.1 Requisitos e Orientações Sistêmicos | 252 |
| 12.1.2 Requisitos e Orientações de Gerenciamento | 253 |
| 12.1.3 Requisitos e Orientações Relacionados a Recursos | 254 |



| | | |
|-----------|--|------------|
| 12.1.4 | Requisitos e Orientações para Realização de Projetos | 255 |
| 12.1.5 | Requisitos e Orientações para Ações Corretivas | 257 |
| 12.2 | ISO/IEC 15504 – SPICE | 259 |
| 12.2.1 | Processo de Avaliação | 261 |
| 12.2.2 | Detalhamento da Dimensão de Processos | 262 |
| 12.3 | CMMI – <i>Capability Maturity Model Integration</i> | 266 |
| 12.3.1 | Práticas e Objetivos Específicos e Genéricos | 267 |
| 12.3.2 | Níveis de Capacidade | 268 |
| 12.3.3 | Níveis de Maturidade | 268 |
| 12.3.4 | Áreas de Processo do CMMI-DEV v1.3 | 268 |
| 12.3.5 | Níveis de Maturidade CMMI | 270 |
| 12.4 | MPS.BR | 271 |
| 12.5 | Melhoria de Processo de Software (SEI-IDEAL) | 276 |
| 12.5.1 | Iniciação | 278 |
| 12.5.2 | Diagnóstico | 279 |
| 12.5.3 | Estabelecimento | 280 |
| 12.5.4 | Ação | 281 |
| 12.5.5 | Alavancagem | 283 |
| 12.5.6 | Gerenciamento do Programa de SPI | 284 |
| 12.6 | Fatores Humanos | 284 |
| 12.7 | Linha de Processo de Software | 287 |
| 13 | Teste | 289 |
| 13.1 | Fundamentos | 290 |
| 13.1.1 | Erro, Defeito e Falha | 290 |
| 13.1.2 | Verificação, Validação e Teste | 291 |
| 13.1.3 | Teste e Depuração | 291 |
| 13.1.4 | <i>Stubs e Drivers</i> | 291 |
| 13.2 | Níveis de Teste de Funcionalidade | 292 |
| 13.2.1 | Teste de Unidade | 292 |
| 13.2.2 | Teste de Integração | 294 |
| 13.2.3 | Teste de Sistema | 295 |
| 13.2.4 | Teste de Aceitação | 296 |
| 13.2.5 | Teste de Ciclo de Negócio | 297 |
| 13.2.6 | Teste de Recessão | 297 |
| 13.3 | Testes Suplementares | 297 |
| 13.3.1 | Teste de Interface com Usuário | 298 |
| 13.3.2 | Teste de Performance (Carga, Estresse e Resistência) | 298 |
| 13.3.3 | Teste de Segurança | 299 |
| 13.3.4 | Teste de Recuperação de Falha | 299 |
| 13.3.5 | Teste de Instalação | 299 |

| | | |
|-----------|---|------------|
| 13.4 | Teste Estrutural | 299 |
| 13.4.1 | Complexidade Ciclomática | 300 |
| 13.4.2 | Grafo de Fluxo | 301 |
| 13.4.3 | Caminhos Independentes | 302 |
| 13.4.4 | Casos de Teste | 304 |
| 13.4.5 | Múltiplas Condições | 304 |
| 13.4.6 | Caminhos Impossíveis | 306 |
| 13.4.7 | Limitações | 307 |
| 13.5 | Teste Funcional | 308 |
| 13.5.1 | Particionamento de Equivalência | 309 |
| 13.6 | TDD – Desenvolvimento Orientado a Testes | 311 |
| 13.7 | Medição em Teste | 312 |
| 13.8 | Depuração | 313 |
| 13.9 | Prova de Correção de Programas | 314 |
| 14 | Manutenção e Evolução de Software | 317 |
| 14.1 | Necessidade de Manutenção e Evolução de Software (Leis de Lehman) | 318 |
| 14.1.1 | Lei da Mudança Contínua | 318 |
| 14.1.2 | Lei da Complexidade Crescente | 319 |
| 14.1.3 | Lei Fundamental da Evolução de Programas: Autorregulação | 319 |
| 14.1.4 | Lei da Conservação da Estabilidade Organizacional: Taxa de Trabalho Invariante | 319 |
| 14.1.5 | Lei da Conservação da Familiaridade: Complexidade Percebida | 319 |
| 14.1.6 | Lei do Crescimento Contínuo | 319 |
| 14.1.7 | Lei da Qualidade Decrescente | 320 |
| 14.1.8 | Lei do Sistema Realimentado | 320 |
| 14.2 | Classificação das Atividades de Manutenção | 320 |
| 14.2.1 | Manutenção Corretiva | 320 |
| 14.2.2 | Manutenção Adaptativa | 321 |
| 14.2.3 | Manutenção Perfectiva | 321 |
| 14.2.4 | Manutenção Preventiva | 322 |
| 14.3 | Processo de Manutenção | 322 |
| 14.4 | Ferramenta para Manutenção de Software | 323 |
| 14.5 | Tipos de Atividades de Manutenção e suas Métricas | 323 |
| 14.5.1 | Reparação de Defeitos | 323 |
| 14.5.2 | Remoção de Módulos Sujeitos a Erros | 324 |
| 14.5.3 | Supporte a Usuários | 324 |
| 14.5.4 | Migração entre Plataformas | 325 |
| 14.5.5 | Conversão de Arquitetura | 325 |
| 14.5.6 | Adaptações Obrigatórias | 325 |



| | |
|--|-----|
| 14.5.7 Otimização de Performance | 325 |
| 14.5.8 Melhorias | 325 |
| 14.6 Modelos de Estimação de Esforço de Manutenção | 326 |
| 14.6.1 Modelo ACT | 326 |
| 14.6.2 Modelo de Manutenção de CII | 327 |
| 14.6.3 Modelos FP e SMPEEM | 329 |
| 14.7 Engenharia Reversa e Reengenharia | 329 |
| 14.7.1 Engenharia Reversa de Código | 330 |
| 14.7.2 Engenharia Reversa de Dados | 332 |
| <i>Posfácio</i> | 333 |
| <i>Referências</i> | 335 |
| <i>Índice remissivo</i> | 339 |



Índice de Tabelas

| | | |
|-------------|--|-----|
| Tabela 5.1 | Detalhamento do <i>workflow</i> para a disciplina de gerenciamento de projeto | 90 |
| Tabela 5.2 | Detalhamento do <i>workflow</i> para a disciplina de análise e <i>design</i> | 97 |
| Tabela 6.1 | Esforço e duração de um projeto típico por fase do UP | 120 |
| Tabela 6.2 | Um exemplo de plano de projeto simplificado com definição de entregas | 121 |
| Tabela 7.1 | <i>Backfire table</i> para conversão de UFP em SLOC | 131 |
| Tabela 7.2 | Combinação de risco de pessoal e risco tecnológico para a escolha do tipo de projeto no contexto de COCOMO | 135 |
| Tabela 7.3 | Valores de <i>ab</i> , <i>bb</i> , <i>cd</i> e <i>db</i> em função do tipo de projeto | 135 |
| Tabela 7.4 | Fatores influenciadores de custo do modelo COCOMO intermediário | 136 |
| Tabela 7.5 | Valores de <i>ai</i> e <i>bi</i> em função do tipo de projeto | 137 |
| Tabela 7.6 | Aplicação de notas aos fatores influenciadores de custo em cenário otimista e pessimista | 137 |
| Tabela 7.7 | Forma de obtenção do equivalente numérico para PREC | 141 |
| Tabela 7.8 | Forma de obtenção do equivalente numérico para FLEX | 142 |
| Tabela 7.9 | Forma de obtenção do equivalente numérico para RESL | 143 |
| Tabela 7.10 | Forma de obtenção do equivalente numérico para TEAM | 144 |
| Tabela 7.11 | Forma de obtenção do equivalente numérico para PMAT | 144 |
| Tabela 7.12 | Forma de obtenção do equivalente numérico para RELY | 145 |
| Tabela 7.13 | Forma de obtenção do equivalente numérico para DATA | 145 |
| Tabela 7.14 | Forma de obtenção do equivalente numérico para CPLX | 146 |
| Tabela 7.15 | Forma de obtenção do equivalente numérico para RUSE | 147 |
| Tabela 7.16 | Forma de obtenção do equivalente numérico para DOCU | 147 |
| Tabela 7.17 | Forma de obtenção do equivalente numérico para TIME | 147 |
| Tabela 7.18 | Forma de obtenção do equivalente numérico para STOR | 147 |
| Tabela 7.19 | Forma de obtenção do equivalente numérico para PVOL | 148 |
| Tabela 7.20 | Forma de obtenção do equivalente numérico para ACAP | 148 |
| Tabela 7.21 | Forma de obtenção do equivalente numérico para PCAP | 148 |
| Tabela 7.22 | Forma de obtenção do equivalente numérico para PCON | 148 |
| Tabela 7.23 | Forma de obtenção do equivalente numérico para APEX | 149 |
| Tabela 7.24 | Forma de obtenção do equivalente numérico para PLEX | 149 |
| Tabela 7.25 | Forma de obtenção do equivalente numérico para LTEX | 149 |
| Tabela 7.26 | Forma de obtenção do equivalente numérico para TOOL | 149 |
| Tabela 7.27 | Forma de obtenção do equivalente numérico para SITE | 150 |
| Tabela 7.28 | Forma de obtenção do equivalente numérico para SCED | 150 |
| Tabela 7.29 | Forma de obtenção do equivalente numérico para PERS | 151 |
| Tabela 7.30 | Forma de obtenção do equivalente numérico para RCPX | 151 |
| Tabela 7.31 | Forma de obtenção do equivalente numérico para PDIF | 151 |
| Tabela 7.32 | Forma de obtenção do equivalente numérico para PREX | 152 |
| Tabela 7.33 | Forma de obtenção do equivalente numérico para FCIL | 152 |
| Tabela 7.34 | Aplicação de esforço e tempo linear às fases do UP | 152 |
| Tabela 7.35 | Exemplo de cálculo de tempo e esforço para as fases do UP de um projeto com $E=56$ e $T=11,5$ | 153 |
| Tabela 7.36 | Resumo do esforço relativo às disciplinas UP nas diferentes fases | 153 |
| Tabela 7.37 | Exemplo de calibragem para a constante <i>A</i> | 154 |
| Tabela 7.38 | Complexidade funcional de entradas externas | 161 |
| Tabela 7.39 | Complexidade funcional de saídas e consultas | 161 |

| | |
|--|-----|
| Tabela 7.40 Complexidade funcional de arquivos internos e externos | 161 |
| Tabela 7.41 Pontos de função não ajustados por tipo e complexidade de função | 161 |
| Tabela 7.42 Exemplo de identificação de funções a partir de requisitos | 163 |
| Tabela 7.43 Fatores técnicos de ajuste de pontos de caso de uso | 172 |
| Tabela 7.44 Fatores ambientais de ajuste de pontos de caso de uso | 173 |
| Tabela 7.45 Atribuição de pontos aos casos de uso de acordo com seu tipo | 176 |
| Tabela 8.1 Forma de cálculo para a importância de um risco | 194 |
| Tabela 8.2 Identificação e análise de riscos de um projeto fictício | 195 |
| Tabela 8.3 Planos de redução de probabilidade | 196 |
| Tabela 8.4 Planos de redução de impacto | 198 |
| Tabela 9.1 Equivalência entre grupos de processo de gerência no PMBOK e no SWEBOK | 205 |
| Tabela 11.1 Fases genéricas do ciclo de vida da Norma ISO/IEC 15288 e suas equivalentes SQuaRE | 230 |
| Tabela 11.2 Modelo de qualidade da ISO 25010:2011 | 233 |
| Tabela 11.3 Propriedades de variáveis que impactam na qualidade de software | 239 |
| Tabela 11.4 Propriedades de expressões que impactam na qualidade de software | 239 |
| Tabela 11.5 Propriedades de comandos que impactam na qualidade de software | 239 |
| Tabela 11.6 Propriedades de interfaces com hardware que impactam na qualidade de software | 240 |
| Tabela 11.7 Propriedades de interfaces com a base de dados que impactam na qualidade de software | 240 |
| Tabela 11.8 Propriedades de comunicação de dados que impactam na qualidade de software | 240 |
| Tabela 11.9 Associação de riscos às características e subcaracterísticas de qualidade | 248 |
| Tabela 11.10 Exemplo de aplicação do modelo GQM22 | 248 |
| Tabela 12.1 Exemplo de avaliação SPICE | 262 |
| Tabela 12.2 Níveis de capacidade e maturidade do CMMI | 267 |
| Tabela 12.3 As 22 áreas de processo de CMMI-DEV | 269 |
| Tabela 12.4 Processos e atributos de processos que definem os níveis de maturidade do MPS.BR | 275 |
| Tabela 12.5 Comparação entre os modelos IDEAL e CM | 286 |
| Tabela 12.6 Ferramentas motivacionais para aplicar nas diferentes fases do modelo IDEAL | 287 |
| Tabela 13.1 Exemplo de plano de teste de sistema para um caso de uso | 296 |
| Tabela 13.2 Casos de teste | 304 |
| Tabela 13.3 Casos de teste para o programa da Figura 13.6 | 305 |
| Tabela 13.4 Casos de teste funcional para uma operação de sistema | 310 |
| Tabela 14.1 Tempo de resposta ao erro em função de sua gravidade | 324 |
| Tabela 14.2 Forma de obtenção do equivalente numérico para RELY na fase de manutenção | 327 |
| Tabela 14.3 Forma de cálculo do equivalente numérico para SU | 328 |
| Tabela 14.4 Forma de cálculo do equivalente numérico para UNFM | 328 |

Índice de Figuras

| | | |
|-------------|---|-----|
| Figura 2.1 | Um <i>template</i> de atividade de projeto | 16 |
| Figura 2.2 | Um exemplo de atividade descrita de acordo com o <i>template</i> apresentado | 17 |
| Figura 3.1 | Modelo Codificar e Consertar | 24 |
| Figura 3.2 | Fases e marcos do Modelo Cascata, de acordo com Boehm | 27 |
| Figura 3.3 | Modelo Cascata original (1970) | 28 |
| Figura 3.4 | Como as interações entre fases poderiam ser (Modelo Cascata Dupla) | 29 |
| Figura 3.5 | Como as interações entre fases do Modelo Cascata acabam acontecendo na prática | 30 |
| Figura 3.6 | Ciclo de vida <i>Sashimi</i> | 31 |
| Figura 3.7 | Modelo V | 32 |
| Figura 3.8 | Modelo W | 33 |
| Figura 3.9 | Modelo Cascata com Subprojetos | 34 |
| Figura 3.10 | Modelo Cascata com Redução de Risco | 35 |
| Figura 3.11 | Ciclo de vida Espiral | 36 |
| Figura 3.12 | Modelo Prototipação Evolucionária | 38 |
| Figura 3.13 | Modelo Entregas em Estágios | 38 |
| Figura 3.14 | Modelo Orientado a Cronograma | 39 |
| Figura 3.15 | Custo/benefício de SPL | 41 |
| Figura 3.16 | As três atividades essenciais para SPL | 42 |
| Figura 3.17 | A atividade de desenvolvimento do núcleo de ativos | 42 |
| Figura 3.18 | Atividade de desenvolvimento do produto em uma SPL | 44 |
| Figura 4.1 | Modelo FDD | 47 |
| Figura 4.2 | Estrutura conceitual da lista de funcionalidades | 48 |
| Figura 4.3 | Exemplo de <i>product backlog</i> | 57 |
| Figura 4.4 | Um típico quadro de andamento de tarefas de um <i>sprint</i> | 58 |
| Figura 4.5 | Um <i>sprint burndown</i> ideal | 59 |
| Figura 4.6 | Um <i>sprint burndown</i> em que as tarefas adicionais são incluídas após o início do <i>sprint</i> | 60 |
| Figura 4.7 | Modelo Scrum | 61 |
| Figura 4.8 | Estrutura do ciclo de vida do <i>Crystal Clear</i> | 67 |
| Figura 4.9 | Modelo ASD | 74 |
| Figura 5.1 | Diferentes ênfases de cada disciplina nas diferentes fases do RUP | 87 |
| Figura 5.2 | <i>Workflow</i> da disciplina gerenciamento de projeto | 89 |
| Figura 5.3 | <i>Workflow</i> da disciplina de modelagem de negócio | 93 |
| Figura 5.4 | <i>Workflow</i> da disciplina de requisitos | 94 |
| Figura 5.5 | <i>Workflow</i> da disciplina de análise e <i>design</i> | 96 |
| Figura 5.6 | <i>Workflow</i> da disciplina de implementação | 98 |
| Figura 5.7 | <i>Workflow</i> da disciplina de testes | 100 |
| Figura 5.8 | <i>Workflow</i> da disciplina de implantação | 102 |
| Figura 5.9 | <i>Workflow</i> da disciplina de gerenciamento de mudança e configuração | 104 |
| Figura 5.10 | <i>Workflow</i> da disciplina de ambiente | 105 |
| Figura 5.11 | Ciclo de vida OpenUP | 108 |
| Figura 5.12 | Modelo de ciclo de vida de uma iteração segundo OpenUP | 108 |
| Figura 5.13 | Ciclo de vida OUM | 110 |
| Figura 6.1 | Perfil de duração e esforço típicos para um projeto usando UP | 118 |
| Figura 6.2 | WBS em uma ferramenta de gerência de projetos, com duração prevista e dependências | 125 |
| Figura 6.3 | Uma rede PERT para as atividades da WBS | 126 |

| | | |
|--------------|---|-----|
| Figura 6.4 | Diagrama Gantt para as atividades da WBS | 127 |
| Figura 7.1 | Região de estimativa de esforço de COCOMO II | 138 |
| Figura 7.2 | Momento de aplicação dos modelos <i>early design</i> e <i>post-architecture</i> | 139 |
| Figura 7.3 | Relação entre o tamanho da equipe e o tempo linear de desenvolvimento de um projeto | 140 |
| Figura 7.4 | Gráfico de velocidade de projeto em pontos de histórias | 178 |
| Figura 8.1 | Um possível ciclo de vida para o <i>status</i> de um risco | 200 |
| Figura 9.1 | Uma folha de tempo (<i>timesheet</i>) | 211 |
| Figura 10.1 | Uma classe que depende de uma versão desatualizada de outra classe | 223 |
| Figura 11.1 | Abordagem conceitual para qualidade de acordo com a ISO/IEC 25010:2011 | 231 |
| Figura 11.2 | Modelo de qualidade de Dromey | 238 |
| Figura 11.3 | Relação entre investimento em qualidade e economia relacionada a falhas | 246 |
| Figura 11.4 | Estrutura hierárquica do modelo GQM | 247 |
| Figura 12.1 | As duas dimensões de avaliação do SPICE | 259 |
| Figura 12.2 | Ciclo de vida do modelo IDEAL | 277 |
| Figura 12.3 | Fases, estágios e desistências no processo de mudança | 284 |
| Figura 12.4 | Modelo conceitual de uma linha de processo de software | 288 |
| Figura 13.1 | Exemplo de programa com complexidade ciclomática 4 | 301 |
| Figura 13.2 | Regras para criação de grafos de fluxo | 302 |
| Figura 13.3 | Regra para criação de grafo de fluxo para estruturas com condição OR | 303 |
| Figura 13.4 | Regra para criação de grafo de fluxo para estruturas com condição AND | 303 |
| Figura 13.5 | Grafo de fluxo do programa da Figura 13.1 | 303 |
| Figura 13.6 | Um programa com uma decisão baseada em múltiplas condições | 304 |
| Figura 13.7 | Grafo de fluxo do programa da Figura 13.6 | 305 |
| Figura 13.8 | Um programa com caminhos impossíveis de testar | 306 |
| Figura 13.9 | Grafo de fluxo do programa da Figura 13.8 | 306 |
| Figura 13.10 | Modelo conceitual de referência para exemplo | 309 |
| Figura 13.11 | Processo de desenvolvimento orientado a testes | 312 |
| Figura 13.12 | Evolução esperada das medidas de defeitos de produto ao longo de um projeto | 313 |
| Figura 14.1 | Relação esquemática entre os diferentes termos relacionados à engenharia reversa | 330 |

Engenharia de Software

Conceitos e Práticas

Raul Sidnei Wazlawick



Preencha a **ficha de cadastro** no final deste livro
E receba gratuitamente informações sobre os
lançamentos e as promoções da Elsevier.

Consulte também nosso catálogo completo, últimos lançamentos
e serviços exclusivos no site
www.elsevier.com.br

Engenharia de Software

Conceitos e Práticas

Raul Sidnei Wazlawick

1^a edição



© 2013, Elsevier Editora Ltda.

Todos os direitos reservados e protegidos pela Lei nº 9.610, de 19/02/1998.

Nenhuma parte deste livro, sem autorização prévia por escrito da editora, poderá ser reproduzida ou transmitida sejam quais forem os meios empregados: eletrônicos, mecânicos, fotográficos, gravação ou quaisquer outros.

Coordenação editorial: Adriana Ayami Takimoto

Copidesque: Andrea Vidal

Revisão: Eloiza Mendes Lopes

Editoração Eletrônica: Thomson Digital

Elsevier Editora Ltda.

Conhecimento sem Fronteiras

Rua Sete de Setembro, 111 – 16º andar
20050-006 – Centro – Rio de Janeiro – RJ – Brasil

Rua Quintana, 753 – 8º andar
04569-011 – Brooklin – São Paulo – SP

Serviço de Atendimento ao Cliente

0800-0265340

sac@elsevier.com.br

ISBN: 978-85-352-6120-2

Nota: Muito zelo e técnica foram empregados na edição desta obra. No entanto, podem ocorrer erros de digitação, impressão ou dúvida conceitual. Em qualquer das hipóteses, solicitamos a comunicação ao nosso Serviço de Atendimento ao Cliente, para que possamos esclarecer ou encaminhar a questão.

Nem a editora nem o autor assumem qualquer responsabilidade por eventuais danos ou perdas a pessoas ou bens originados do uso desta publicação.

CIP-BRASIL. CATALOGAÇÃO-NA-FONTE
SINDICATO NACIONAL DOS EDITORES DE LIVROS, RJ

W372e

Wazlawick, Raul Sidnei, 1967-

Engenharia de software: conceitos e práticas / Raul Sidnei Wazlawick. - Rio de Janeiro:
Elsevier, 2013.

28 cm

ISBN 978-85-352-6084-7

1. Engenharia de software 2. Software - Desenvolvimento. I. Título.

13-0677.

CDD: 005.1

CDU: 004.41

30.01.13 01.02.13

042501

Sobre o Autor

Raul Sidnei Wazlawick é Professor Associado IV do Departamento de Informática e Estatística da Universidade Federal de Santa Catarina (UFSC), Bacharel em Ciência da Computação (UFSC, 1988), Mestre em Ciência da Computação pela Universidade Federal do Rio Grande do Sul (UFRGS, 1991) e Doutor em Engenharia (UFSC, 1993), com Pós-Doutorado em Informática pela Universidade Nova de Lisboa (1998). Desde 1986, trabalha na área de Engenharia de Software, sendo docente desde 1992 na UFSC e em cursos e palestras em dezenas de outras universidades e empresas no Brasil e no exterior. Trabalha também em projetos de pesquisa em consultoria em empresas para melhoria de processos de desenvolvimento de software. Foi membro do Conselho da Sociedade Brasileira de Computação (SBC), coordenador dos cursos de Bacharelado e Mestrado em Ciência da Computação da UFSC, coordenador do XX Simpósio Brasileiro de Engenharia de Software (SBES 2006) e *chair* do Working Group 3.2 (*Higher Education*) da International Federation for Information Processing (IFIP). Foi também membro da Comissão de Especialistas em Ensino de Informática e Computação do Ministério da Educação (MEC). Orientou dezenas de dissertações de mestrado, teses de doutorado e trabalhos de graduação; tendo também publicado mais de uma centena de artigos científicos em eventos e periódicos internacionais. Recebeu, juntamente com sua orientanda Marília Guterres Ferreira, o prêmio Best Paper Award na Conferência Ibero-americana de Engenharia de Software em 2011. É bolsista de produtividade em desenvolvimento tecnológico e extensão inovadora pelo CNPq.

Prefácio

O SWEBOK¹ (*Software Engineering Book of Knowledge*), organizado pela IEEE Computer Society, define o corpo de conhecimentos usualmente aceito relacionado à Engenharia de Software (IEEE Computer Society, 2004). Esta publicação relaciona dez áreas de conhecimento, das quais sete são tratadas aqui neste livro (Teste de Software, Manutenção de Software, Gerenciamento de Configuração de Software, Gerenciamento de Engenharia de Software, Processo de Engenharia de Software, Ferramentas e Métodos de Engenharia de Software e Qualidade de Software), enquanto as outras três áreas (Requisitos de Software, *Design* de Software e Construção de Software) são tratadas no livro *Análise e Projeto de Sistemas de Informação Orientados a Objetos* (Wazlawick, 2011).

Este livro não possui um capítulo específico sobre ferramentas de Engenharia de Software, porque elas são mencionadas ao longo de todos os capítulos onde se fazem necessárias. As referências incluem, sempre que possível, produtos gratuitos e proprietários, bem como a referência para o *site* da empresa ou grupo responsável pela ferramenta.

As referências bibliográficas, também, sempre que disponíveis na Internet, têm sua URL referenciada no local em que aparecem no texto, de forma a facilitar uma eventual busca pelas fontes originais deste trabalho.

O livro não esgota o assunto relacionado à Engenharia de Software, mas os tópicos mais fundamentais estão detalhados de modo que o leitor consiga efetivamente usar as técnicas, e não apenas ouvir falar delas.

A profundidade com que os tópicos são abordados corresponde ao que o autor entende que seria adequado a um aluno de graduação em Sistemas de Informação ou Ciência da Computação, para que possa desempenhar a função de engenheiro de software adequadamente no mercado de trabalho.

O livro também pode ser de grande valia para profissionais que tenham interesse em se reciclar ou dar o passo inicial para implantar processos produtivos mais organizados em suas empresas.

O conteúdo deste livro pode ser abordado possivelmente em uma disciplina de 90 horas ou ainda em três disciplinas de 30 horas cada, em que cada disciplina abordaria uma das partes do livro: processo de engenharia de software, gerenciamento de projeto de software e qualidade de software. Uma quarta disciplina de cerca de 40 a 50 horas ainda poderia ser acrescentada com o uso do livro *Análise e Projeto de Sistemas de Informação Orientados a Objetos*, que complementa este livro com a apresentação de técnicas detalhadas para requisitos, análise, *design* e geração de código.

Não foram incluídos exercícios no livro porque estão disponibilizados no *site* da editora, bastando, para acessá-los, usar o código fornecido pelo próprio livro. Esta escolha visa, por um lado, permitir que o livro tenha um preço mais acessível, já que diminui o número total de páginas e, por outro lado, permite que novos exercícios possam ser dinamicamente adicionados pelo autor, à medida que são produzidos.

Os leitores são incentivados a entrarem em contato com o autor pelo email raul@inf.ufsc.br para apresentar sugestões de melhoria ao livro, pois, de acordo com as sugestões apresentadas no próprio livro, o *feedback* do usuário é fundamental para a criação de produtos de qualidade.

Este livro foi produzido em ciclos iterativos entre os anos de 2010 e 2012, a partir de diversas fontes bibliográficas e da experiência do autor na área, que iniciou em 1985 seus estudos e, até hoje, leciona, publica e presta consultoria na área de Engenharia de Software.

O livro pode ser lido do início ao fim, mas também pode ser lido em qualquer outra ordem, visto que os capítulos são autocontidos e eventuais referências a outros capítulos são feitas quando necessário.

Alguns detalhamentos relacionados a normas técnicas podem ser omitidos da leitura, e mantidos à mão para referência, mas, mesmo assim, foram mantidos no corpo do livro, visto que, apesar de extensos, tais trechos reforçam as boas práticas de Engenharia de Software que devem ser seguidas para que sejam desenvolvidos sistemas com qualidade.

Este livro é dedicado a todos aqueles que acreditam e trabalham para que o Brasil possa um dia ser referência mundial na área de produção de software com qualidade e criatividade. Vamos melhorar nossos processos de produção para que isso ocorra ainda nesta geração!

Ingleses do Rio Vermelho, distrito de Florianópolis, 16 de fevereiro de 2012.

¹www.computer.org/portal/web/swebok/html/contents

Introdução



Este capítulo apresenta uma introdução ao assunto de engenharia de software, iniciando pelos clássicos *a crise do software* (Seção 1.1) e *os mitos do software* (Seção 1.2). Procura-se dar um entendimento às expressões “engenharia de software” (Seção 1.3) e “engenheiro de software” (Seção 1.4), já que a literatura não é homogênea em relação a elas. A *evolução* da área é brevemente apresentada (Seção 1.5), bem como uma classificação dos *tipos de sistemas* referentes à engenharia de software (Seção 1.6), uma vez que este livro se especializa em engenharia de software para sistemas de informação (outros tipos de sistema poderão necessitar de técnicas específicas que não são tratadas aqui). Finalmente, os *princípios* da engenharia de software, que permeiam todos os capítulos do livro, são brevemente apresentados (Seção 1.7).

1.1 A Crise dos Desenvolvedores de Software Menos Preparados

Por algum motivo, os livros de engenharia de software quase sempre iniciam com o tema “crise do software”. Essa expressão vem dos anos 1970. Mas o que é isso, afinal? O software está em crise? Parece que não, visto que hoje o software está presente em quase todas as atividades humanas. Mas as *pessoas* que desenvolvem software estão em crise há décadas e, em alguns casos, parecem impotentes para sair dela.

Em grande parte, parece haver desorientação em relação a como planejar e conduzir o processo de desenvolvimento de software. Muitos desenvolvedores concordam que não utilizam um processo adequado e que deveriam investir em algum, mas ao mesmo tempo dizem que não têm tempo ou recursos financeiros para fazê-lo. Essa história se repete há décadas.

A expressão “crise do software” foi usada pela primeira vez com impacto por Dijkstra (1971)¹. Ele avaliava que, considerando o rápido progresso do hardware e das demandas por sistemas cada vez mais complexos, os desenvolvedores simplesmente estavam se perdendo, porque a engenharia de software, na época, era uma disciplina incipiente.

Os problemas relatados por Dijkstra eram os seguintes:

- a)** Projetos que estouraram o cronograma.
- b)** Projetos que estouraram o orçamento.
- c)** Produto final de baixa qualidade ou que não atenda aos requisitos.
- d)** Produtos não gerenciáveis e difíceis de manter e evoluir.

¹Disponível em: <www.cs.utexas.edu/users/EWD/ewd03xx/EWD340.PDF>. Acesso em: 21 jan. 2013.

Alguma semelhança com sistemas do início do século XXI? Muitas! Sucedeu que, embora a engenharia de software tenha evoluído como ciência, sua aplicação na prática ainda é muito limitada, especialmente em empresas de pequeno porte e em empresas novas.

Mesmo depois de 40 anos, ainda são comuns as queixas da alta administração das empresas em relação ao setor de informática quanto a prazos que não são cumpridos, custos ainda muito elevados, sistemas em uso que demandam muita manutenção e também ao fato de que é difícil recrutar profissionais qualificados.

Os usuários também estão infelizes: encontram erros e falhas inadmissíveis em sistemas entregues, sentem-se inseguros em usar tais sistemas, reclamam da constante necessidade de manutenção e do seu alto custo.

Os desenvolvedores, por sua vez, não estão mais satisfeitos: sentem que sua produtividade é baixa em relação ao seu potencial, lamentam a falta de qualidade no produto gerado por seu trabalho, sofrem pressão para cumprir prazos e orçamentos apertados, e ficam inseguros com as mudanças de tecnologia que afetam sua qualificação em relação ao mercado.

Booch (1994) afirma: “*We often call this condition ‘the software crisis’, but frankly, a malady that has carried on this long must be called ‘normal’*”². Pode-se concluir que a crise do software continuará enquanto os desenvolvedores continuarem a utilizar processos artesanais e a não capitalizar erros e acertos.

Teixeira (2010)³ compara o desenvolvimento de software ao artesanato da Idade Média, quando, por exemplo, um artesão fazia um par de sapatos como um produto único para cada cliente. Buscava-se a matéria-prima, cortava-se e costurava-se para produzir um sapato que servisse ao cliente. O software, em muitas empresas, ainda é desenvolvido dessa forma artesanal.

Porém, apenas a adoção de processos efetivamente industriais para a produção de software poderá fazer essa área desenvolver-se mais rapidamente, com mais qualidade e, finalmente, sair dessa dificuldade crônica que já nem pode ser chamada de crise.

1.2 Os Eternos Mitos

São bastante conhecidos também os *mitos* do software, identificados por Pressman (2005). Esses mitos são crenças tácitas e explícitas que permeiam a cultura de desenvolvimento de software. Os mais experientes acabam percebendo que elas não têm fundamento, constituindo-se realmente em mitos, mas a cada ano novos desenvolvedores de software entram no mercado e reavivam as velhas crenças, já que seu apelo é grande.

Pressman classifica os mitos em três grupos: *administrativos*, do *cliente* e do *profissional*.

Seguem alguns comentários sobre os mitos *administrativos*:

- a) *A existência de um manual de procedimentos e padrões é suficiente para a equipe produzir com qualidade.* Na verdade, deve-se questionar se o manual é realmente usado, se ele é completo e atualizado. Deve-se trabalhar com processos que possam ser gerenciáveis e otimizados, ou seja, sempre que a equipe identificar falhas no processo, deve haver outro processo para modificá-lo.
- b) *A empresa deve produzir com qualidade, pois tem ferramentas e computadores de última geração.* Na verdade, ferramentas e computadores de boa qualidade são condições necessárias, mas não suficientes. Parafraseando Larman (2001), comprar uma ferramenta não transforma você instantaneamente em arquiteto.
- c) *Se o projeto estiver atrasado, sempre é possível adicionar mais programadores para cumprir o cronograma.* O desenvolvimento de software é uma tarefa altamente complexa. Adicionar mais pessoas sem que haja um planejamento prévio pode causar mais atrasos. Se não fosse assim, um programa de 20 mil linhas poderia ser escrito rapidamente por 20 mil programadores.
- d) *Um bom gerente pode gerenciar qualquer projeto.* Gerenciar não é fazer, e o desenvolvimento de software é um processo complexo por vários motivos. Assim, mesmo que o gerente seja competente, se não houver boa comunicação com a equipe, competência técnica e um processo de trabalho previsível, ele pouco poderá fazer para obter um produto com a qualidade desejada.

²Frequentemente chamamos essa condição de “crise do software”, mas, francamente, uma doença que já dura tanto tempo devia ser chamada de “normalidade”.

³Disponível em: <pt.scribd.com/doc/37503268/A-Crise-de-SW-Hugo-Vidal-Teixeira>. Acesso em: 21 jan. 2013.

Entre os mitos relacionados ao *cliente*, pode-se citar:

- a) *Uma declaração geral de objetivos é suficiente para iniciar a fase de programação. Os detalhes podem ser adicionados depois.* É verdade que não se pode esperar que a especificação inicial do sistema esteja correta e completa antes de se iniciar a programação, mas ter isso como meta é péssimo. Deve-se procurar obter o máximo de detalhes possível antes de iniciar a construção do sistema. Técnicas mais sofisticadas de análise de requisitos e uma equipe bem treinada poderão ajudar a construir as melhores especificações possíveis sem perda de tempo.
- b) *Os requisitos mudam com frequência, mas sempre é possível acomodá-los, pois o software é flexível.* Na verdade, o código é fácil de mudar – basta usar um editor. Mas mudar o código sem introduzir erros é uma tarefa bastante improvável, especialmente em empresas com baixa maturidade de processo. O software só será efetivamente flexível se for construído com esse fim. É necessário, entre outras coisas, identificar os requisitos permanentes e os transitórios, e, no caso dos transitórios, preparar o sistema para sua mudança utilizando padrões de *design* adequados. Mesmo que o software não seja um elemento físico, como um edifício ou uma ponte (mais difíceis de serem modificados), a mudança do software também implica esforço e custo (em tempo), e muitas vezes esse esforço e esse custo não são triviais.
- c) *Eu sei do que preciso.* Os desenvolvedores costumam dizer o inverso: o cliente não sabe do que precisa. É necessário que os analistas entendam que os clientes (a não ser que sejam técnicos especializados) raramente sabem do que realmente precisam e têm grande dificuldade para descrever e até mesmo para lembrar-se de suas necessidades. De outro lado, analistas muitas vezes confundem as *necessidades do cliente* (alvo da *análise*) com as *soluções possíveis* (alvo do *design*). Por exemplo, um analista pode achar que o cliente precisa de um sistema *web* com tabelas relacionais, mas essa não é uma boa descrição de uma necessidade. É a descrição de uma solução para uma necessidade que possivelmente não foi estabelecida de maneira clara. Outras soluções poderiam ser aplicadas.

Outros mitos de Pressman dizem respeito ao *profissional*, e são comentados a seguir:

- a) *Assim que o programa for colocado em operação, nosso trabalho terminou.* Na verdade, ainda haverá muito esforço a ser despendido depois da instalação do sistema, por causa de erros dos mais diversos tipos. Estudos indicam que mais da metade do esforço despendido com um sistema ocorre depois de sua implantação (Von Mayrhofer & Vans, 1995).
- b) *Enquanto o programa não estiver funcionando, não será possível avaliar sua qualidade.* Na verdade, o programa é apenas um dos artefatos produzidos no processo de construção do software (possivelmente o mais importante, mas não o único). Existem formas de avaliar a qualidade de artefatos intermediários como casos de uso e modelos conceituais para verificar se estão adequados mesmo antes da implementação do sistema.
- c) *Se eu esquecer alguma coisa, posso arrumar depois.* Quanto mais o processo de desenvolvimento avança, mais caras ficam as modificações em termos de tempo e dinheiro.
- d) *A única entrega importante em um projeto de software é o software funcionando.* Talvez essa seja a entrega mais importante, porém, se os usuários não conseguirem utilizar o sistema ou se os dados não forem corretamente importados, pouco valor ele terá. O final de um projeto de informática costuma envolver mais do que simplesmente entregar o software na portaria da empresa do cliente. É necessário realizar testes de operação, importar dados, treinar usuários, definir procedimentos operacionais. Assim, outros artefatos, além do software funcionando, poderão ser necessários.

Leveson (1995) também apresenta um conjunto de mitos correntes, a maioria dos quais está relacionada à confiabilidade do software:

- a) *O teste do software ou sua verificação formal pode remover todos os erros.* Na verdade, o software é construído com base em uma especificação de requisitos que usualmente é feita em linguagem natural e, portanto, aberta a interpretações. Nessas interpretações pode haver erros ocultos. Além disso, a complexidade do software contemporâneo é tão grande que se torna inviável testar todos os caminhos possíveis. Assim, a única possibilidade é testar os caminhos mais prováveis de ocorrer ou de provocar erros. As técnicas de teste é que vão indicar quais caminhos é interessante verificar.
- b) *Aumentar a confiabilidade do software aumenta a segurança.* O problema é que o software pode ser confiável apenas em relação à sua especificação, ou seja, ele pode estar *fazendo certo a coisa errada*. Isso normalmente

se deve a requisitos mal compreendidos. É consenso que corrigir erros durante o levantamento de requisitos é muito mais barato do que corrigi-los em um sistema em operação.

- c) *O reúso de software aumenta a segurança.* Certamente, pois os componentes reusados já foram testados. O problema é saber se os requisitos foram corretamente estabelecidos, o que leva de volta ao mito anterior. Além disso, se os componentes reusados forem sensíveis ao contexto (o que acontece muitas vezes), coisas inesperadas poderão acontecer.

Porém, de nada adianta estar consciente dos mitos. Para produzir software com mais qualidade e confiabilidade é necessário utilizar uma série de conceitos e práticas. Ao longo deste livro, vários conceitos e práticas úteis em Engenharia de Software serão apresentados ao leitor de forma que ele possa compreender o alcance dessa área e seu potencial para a melhoria dos processos de produção de sistemas

1.3 (In)Definição de Engenharia de Software

Segundo a Desciclopédia⁴, a Engenharia de Software pode ser definida assim:

A Engenharia de Software forma um aglomerado de conceitos que dizem absolutamente nada e que geram no estudante dessa área um sentimento de “Nossa, li 15kg de livros desta mesma. e não aprendi nada”. É tudo bom senso.

Apesar de considerar que o material publicado na Desciclopédia seja de caráter humorístico, a sensação que muitas vezes se tem da Engenharia de Software é mais ou menos essa mesma. Pelo menos foi essa a sensação do autor deste livro e de muitos de seus colegas ao longo de vários anos.

Efetivamente, não é algo simples conceituar e praticar a Engenharia de Software. Mas é *necessário*. Primeiramente, deve-se ter em mente que os processos de Engenharia de Software são diferentes, dependendo do tipo de software que se vai desenvolver. O Capítulo 2, por exemplo, vai mostrar que, dependendo do nível de conhecimento ou estabilidade dos requisitos, deve-se optar por um ou outro ciclo de vida, o qual será mais adequado ao tipo de desenvolvimento que se vai encontrar. O Capítulo 8, por outro lado, vai mostrar que uma área aparentemente tão subjetiva como “riscos” pode ser sistematizada e tratada efetivamente como um processo de engenharia, e não como adivinhação. O Capítulo 7 apresentará formas objetivas e padronizadas para mensurar o esforço do desenvolvimento de software, de forma a gerar números que sejam efetivamente realistas, o que já vem sendo comprovado em diversas empresas.

Assim, espera-se que este livro consiga deixar no leitor a sensação de que a Engenharia de Software é possível e viável, desde que se compreenda em que ela realmente consiste e como pode ser utilizada na prática.

Várias definições de Engenharia de Software podem ser encontradas na literatura, como, por exemplo:

- a) Engenharia de software é uma profissão dedicada a projetar, implementar e modificar software, de forma que ele seja de alta qualidade, a um custo razoável, manutenível e rápido de construir (Laplante, 2007).
- b) Engenharia de software é a aplicação de abordagens sistemáticas, disciplinadas e quantificáveis ao desenvolvimento, operação e manutenção de software, além do estudo dessas abordagens (IEEE Computer Society, 2004)⁵.

Neste livro será considerada com maior ênfase a definição da Engenharia de Software como o processo de estudar, criar e otimizar os processos de trabalho para os desenvolvedores de software. Assim, embora isso não seja consenso geral, considera-se, que as atividades de levantamento de requisitos, modelagem, *design*⁶ e codificação, por exemplo, não são típicas de um engenheiro de software, embora, muitas vezes, ele seja habilitado a realizá-las. Sua tarefa consiste mais em observar, avaliar, orientar e alterar os processos produtivos quando necessário. A seção seguinte procura deixar mais clara essa distinção entre as atividades dos desenvolvedores e do engenheiro de software.

⁴Disponível em: <desciclo.pedia.ws/wiki/Engenharia_de_Software>. Acesso em: 21 jan. 2013.

⁵Disponível em: <www.computer.org/portal/web/swebok/htmlformat>. Acesso em: 21 jan. 2013.

⁶As palavras inglesas *design* e *project* costumam ser traduzidas para o português como “projeto”. Para evitar confusão entre os dois significados, neste livro o termo *design* não será traduzido.

1.4 O Engenheiro de Software

Uma das primeiras confusões que se faz nesta área é entre o *desenvolvedor* e o *engenheiro* de software. Isso equivale a confundir o engenheiro civil com o pedreiro ou com o mestre de obras.

O desenvolvedor, seja ele analista, *designer*, programador ou gerente de projeto, é um *executor* do processo de construção de software. Os desenvolvedores, de acordo com seus papéis, têm a responsabilidade de descobrir os requisitos e transformá-los em um produto executável. Mas o engenheiro de software tem um metapapel em relação a isso. Pode-se dizer que o engenheiro de software não coloca a mão na massa, assim como o engenheiro civil não vai à obra assentar tijolos ou concretar uma laje.

Então, o engenheiro de software não é um desenvolvedor que trabalha nas atividades de análise e produção de código. Porém, a comparação com a engenharia civil termina por aqui, já que o engenheiro civil será o responsável pela especificação do *design*. Na área de Computação, a especificação do *design* fica a cargo do analista e do *designer*, o primeiro com a responsabilidade de identificar os requisitos e o segundo de desenhar uma solução que utilize a tecnologia para transformar esses requisitos em um sistema executável. No âmbito do software, há muito tempo esses papéis têm sido confundidos com o do engenheiro de software, mas por que dar outro nome ao analista e *designer*? Não há razão para isso. Trata-se apenas de uma incompreensão do verdadeiro papel do engenheiro de software.

Assim, o engenheiro de software assemelha-se mais ao engenheiro de produção. Ele deve fornecer aos desenvolvedores (inclusive gerentes, analistas e *designers*) as ferramentas e processos que deverão ser usados e será o responsável por verificar se esse uso está sendo feito efetivamente e de forma otimizada. Além disso, caso tais ferramentas e processos apresentem qualquer problema, ele será o responsável por realizar as modificações necessárias, garantindo assim sua contínua melhoria.

O engenheiro de software, portanto, não desenvolve nem especifica software. Ele viabiliza e acompanha o processo de produção, fornecendo e avaliando as ferramentas e técnicas que julgar mais adequadas a cada projeto ou empresa.

É necessário, ainda, distinguir o engenheiro de software do gerente de projeto. O gerente de projeto deve planejar e garantir que ele seja executado de forma adequada dentro dos prazos e orçamento especificados. Mas o gerente de projeto tem uma responsabilidade mais restrita ao projeto em si, e não ao processo de produção. Nesse sentido, ele também é um executor. Utiliza as disciplinas definidas no processo de engenharia de software para gerenciar seu projeto específico, mas não é necessariamente o responsável pela evolução desses processos, nem necessariamente o responsável por sua escolha. Esse papel cabe ao engenheiro de software.

Resumindo, os diferentes papéis poderiam ser caracterizados assim:

- a) O *engenheiro de software* escolhe e, muitas vezes, especifica os processos de planejamento, gerência e produção a serem implementados. Ele acompanha e avalia o desenvolvimento de todos os projetos da empresa para verificar se o processo estabelecido é executado de forma eficiente e efetiva. Caso sejam necessárias mudanças no processo estabelecido, ele as identifica e realiza, garantindo que a equipe adote tais mudanças. Ele reavalia o processo continuamente.
- b) O *gerente de projeto* cuida de um projeto específico, garantindo que os prazos e orçamento sejam cumpridos. Ele segue as práticas definidas no processo de engenharia. É responsável por verificar a aplicação do processo pelos desenvolvedores e, se necessário, reporta-se ao engenheiro de software para sugerir melhorias.
- c) O *analista* é um desenvolvedor responsável pela compreensão do problema relacionado ao sistema que se deve desenvolver, ou seja, pelo levantamento dos requisitos e sua efetiva modelagem. O analista deve, portanto, descobrir o que o cliente precisa (por exemplo, controlar suas vendas, comissões, produtos etc.).
- d) O *designer* deve levar em conta as especificações do analista e propor a melhor tecnologia para produzir um sistema executável para elas. Deve, então, apresentar uma solução para as necessidades do cliente (por exemplo, propor uma solução baseada em *web*, com um banco de dados centralizado acessível por dispositivos móveis etc.).
- e) O *programador* vai construir a solução física a partir das especificações do *designer*. É ele quem gera o produto final, e deve conhecer profundamente a linguagem e o ambiente de programação, bem como as bibliotecas que for usar, além de ter algum conhecimento sobre teste e depuração de software.

Evidentemente, muitas vezes essa divisão de papéis não é observada estritamente nas empresas. Apenas empresas de médio e grande porte podem se dar o luxo de ter um ou mais engenheiros de software dedicados. Mas é importante que se tenha em mente que, ainda que uma pessoa execute mais de um papel nesse processo, esses papéis são distintos.

1.5 Evolução da Engenharia de Software

Os primeiros computadores, construídos na década de 1940, não possuíam software: os comandos eram implantados na máquina a partir de conexões físicas entre os componentes. À medida que se percebeu a necessidade de computadores mais flexíveis, surgiu o software, que consiste em um conjunto de instruções que fazem a máquina produzir algum tipo de processamento. Como o software era um construto abstrato, sua produção não se encaixava perfeitamente em nenhuma das engenharias, nem mesmo na mecânica e na elétrica, que são as mais próximas, por terem relação com as máquinas que efetuam as computações. Surgiu, então, o conceito de engenharia de software, inicialmente referindo-se aos processos para a produção desse tipo de construto abstrato.

Aceita-se que a primeira conferência sobre Engenharia de Software tenha sido a Conferência de Engenharia de Software da OTAN, organizada em Garmish, Alemanha, em 1968 (Bauer, 1968)⁷. Apesar disso, o termo já era usado desde os anos 1950.

O período da década de 1960 até meados da década de 1980 foi marcado pela chamada “crise do software”, durante a qual foram identificados os maiores problemas relacionados à produção de software, especialmente em larga escala. Inicialmente, a crise referenciava especialmente questões relacionadas com orçamento e cronograma de desenvolvimento, mas posteriormente passou também a abranger aspectos de qualidade de software, uma vez que os sistemas, depois de prontos, apresentavam muitos erros, causando prejuízos.

Um exemplo clássico da crise de software dos anos 1960 foi o projeto do sistema operacional OS/360, que utilizou mais de mil programadores. Brooks (1975) afirmou ter cometido um erro que custou milhões à IBM nesse projeto, por não ter definido uma arquitetura estável antes de iniciar o desenvolvimento propriamente dito. Atualmente, a Lei de Brooks afirma que adicionar programadores a um projeto atrasado faz com que ele fique ainda mais atrasado.

Por décadas, a atividade de pesquisa tentou resolver a crise do software. Cada nova abordagem era apontada como uma “bala de prata” para solucionar a crise. Porém, pouco a pouco, chegou-se ao consenso de que tal solução mágica não existia. Ferramentas CASE (*Computer Aided Software Engineering*), especificação formal, processos, componentes etc. foram boas técnicas que ajudaram a engenharia de software a evoluir, mas hoje não se acredita mais em uma solução única e salvadora para os complexos problemas envolvidos com a produção de software.

Os anos 1990 presenciam o surgimento da internet e a consolidação da orientação a objetos como o paradigma predominante na produção de software. A mudança de paradigma e o *boom* da internet mudaram de forma determinante a maneira como o software era produzido. Novas necessidades surgiram e sistemas cada vez mais complexos, acessíveis de qualquer lugar do mundo, substituíram os antigos sistemas *stand-alone*. Com isso, novas preocupações relacionadas à segurança da informação e à proliferação de vírus e *spam* surgiram e passaram a fazer parte da agenda dos desenvolvedores de software.

Nos anos 2000, o crescimento da demanda por software em organizações de pequeno e médio porte levou ao surgimento de soluções mais simples e efetivas para o desenvolvimento de software para essas organizações. Assim surgiram os *métodos ágeis*, que procuram desburocratizar o processo de desenvolvimento e deixá-lo mais adequado a equipes pequenas mas competentes, capazes de desenvolver sistemas sem a necessidade de extensas listas de procedimentos ou de “receitas de bolo”.

Atualmente, a área vem tentando se estabelecer como um corpo de conhecimentos coeso. O surgimento do SWEBOk (IEEE Computer Society, 2004) e sua adoção como padrão internacional em 2006 (ISO/IEC TR 19759) foi um avanço para a sistematização do corpo de conhecimentos da área.

1.6 Tipos de Software do Ponto de Vista da Engenharia

Não existe um processo único e ideal para desenvolvimento de software, porque cada sistema tem suas particularidades. Porém, usualmente, podem-se agrupar os sistemas de acordo com certas características e então definir modelos de processo mais adequados a elas. Do ponto de vista da engenharia de software, os sistemas podem ser classificados da seguinte forma:

- a) *Software básico*: são os compiladores, *drivers* e componentes do sistema operacional.
- b) *Software de tempo real*: são os sistemas que monitoram, analisam e controlam eventos do mundo real.

⁷Disponível em: <homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF>. Acesso em: 21 jan. 2013.

- c) *Software comercial*: são os sistemas aplicados nas empresas, como controle de estoque, vendas etc. Tais sistemas usualmente acessam bancos de dados. São também conhecidos como *sistemas de informação*.
- d) *Software científico e de engenharia*: são os sistemas que utilizam intenso processamento de números.
- e) *Software embutido ou embarcado*: são os sistemas de software presentes em celulares, eletrodomésticos, automóveis etc. Normalmente, tais sistemas precisam trabalhar sob severas restrições de espaço, tempo de processamento e gasto de energia.
- f) *Software pessoal*: são os sistemas usados por pessoas no dia a dia, como processadores de texto, planilhas etc.
- g) *Jogos*: embora existam alguns jogos cujo processamento não é muito complexo, existem também aqueles que exigem o máximo dos computadores em função da qualidade de gráficos e da necessidade de reação em tempo real. Apesar disso, todas as categorias de jogos têm características intrínsecas que extrapolam o domínio da engenharia de software.
- h) *Inteligência artificial*: são os sistemas especialistas, redes neurais e sistemas capazes de alguma forma de aprendizado. Além de serem sistemas independentes, com um tipo de processo de construção próprio, podem também ser embutidos em outros sistemas.

Essa classificação, porém, não é completa, detalhada nem exaustiva, apenas ilustra os diferentes tipos de sistema que são desenvolvidos com o uso de software e, eventualmente, de hardware.

Neste livro, a ênfase estará nos *sistemas de informação*, ou seja, será mostrado basicamente como desenvolver um processo de engenharia de software para sistemas do tipo “comercial”. Apesar disso, algumas técnicas poderão ser aplicadas também ao desenvolvimento de outros tipos de software.

1.7 Princípios da Engenharia de Software

A engenharia de software classicamente apresenta um conjunto de princípios que devem ser usados quando um projeto de desenvolvimento de software for realizado. A ideia é que esses princípios funcionem como boas práticas ou lições aprendidas sobre como desenvolver software. Usualmente não se trata de regras, mas de uma filosofia de desenvolvimento. Entre outros princípios, pode-se citar:

- a) *Decomposição*: um dos mais antigos princípios em desenvolvimento de software é a noção de que o software é um produto complexo construído a partir de partes cada vez mais simples. A decomposição funcional é uma maneira de conceber o software como um conjunto de funções de alto nível (requisitos) que são decompostas em partes cada vez mais simples até chegar a comandos individuais de uma linguagem de programação. Modernamente, essa noção foi substituída pela decomposição em objetos, na qual, em vez de decompor a função original em funções cada vez mais simples, procura-se determinar uma arquitetura de classes de objetos que possa realizar a função.
- b) *Abstração*: outro princípio antigo da engenharia de software é o uso da abstração, que consiste em descrever um elemento em uma linguagem de nível mais alto do que o necessário para sua construção. Um sistema de software, por exemplo, pode ser composto por 100 mil linhas de código (sua representação física concreta), porém, usando-se abstração, talvez o mesmo sistema possa ser descrito por um diagrama ou por umas 50 linhas de texto. A abstração ajuda os interessados no processo de desenvolvimento a entender estruturas grandes e complexas através de descrições mais abstratas.
- c) *Generalização*: o princípio da generalização deu origem à orientação a objetos, em que um objeto pode ser classificado simultaneamente em mais de uma classe. Por exemplo, um cão, além de ser um mamífero, é animal e vertebrado. Esse princípio é usado não só na classificação de dados, mas também em várias outras situações em projetos de desenvolvimento de software, como a arquitetura dos sistemas.
- d) *Padronização*: a criação de padrões (*patterns*) de programação, *design* e análise ajuda a elaborar produtos com qualidade mais previsível. Padrões também são importantes para a capitalização de experiências. Desenvolvedores que não utilizam padrões podem repetir erros que já têm solução conhecida.
- e) *Flexibilização*: uma das qualidades a serem buscadas no desenvolvimento de software é a flexibilização, a fim de que se tenha mais facilidade para acomodar as inevitáveis mudanças nos requisitos. Infelizmente, essa qualidade não pode ser obtida sem certo custo inicial, mas, muitas vezes, esse investimento compensa a economia que se obtém depois.

- f) *Formalidade*: como o software é um construto formal, de modo geral aceita-se que deva ser especificado de maneira também formal. Existem técnicas que iniciam com descrições informais, mas, à medida que caminham para o produto final, elas precisam ser cada vez mais formais. Outras técnicas já iniciam com documentos formais, de alta cerimônia, e tendem a ser menos ambíguas, embora algumas vezes sejam mais difíceis de se compreender.
- g) *Rastreabilidade*: o software é desenvolvido por um processo complexo no qual diferentes documentos e especificações são produzidos e, algumas vezes, derivados uns dos outros. É necessário manter um registro de *traços* entre os diferentes artefatos para que se saiba quando alterações feitas em um deles se refletem em outros. Por exemplo, um requisito que muda tem efeito sobre vários outros artefatos, como modelos de classes, casos de uso ou mesmo código executável.
- h) *Desenvolvimento iterativo*: no início da era do computador, quando o software era mais simples, até se podia conceber o desenvolvimento como um processo de um único ciclo, com início e fim bem definidos. Porém, as técnicas modernas apontam cada vez mais para o desenvolvimento iterativo como uma maneira de lidar com a crescente complexidade do software. Assim, vários ciclos de desenvolvimento são realizados, cada qual com um conjunto de objetivos distintos e cada qual contribuindo para a geração e amadurecimento do produto final.
- i) *Gerenciamento de requisitos*: antigamente, acreditava-se que era possível descobrir de saída quais eram os requisitos de um sistema. Depois, bastava desenvolver o melhor sistema possível que atendesse a esses requisitos. Modernamente, sabe-se que requisitos mudam com muita frequência e que por isso é necessário gerenciar sua mudança como parte integrante do processo de desenvolvimento e evolução de software, e não apenas como um mal necessário.
- j) *Arquiteturas baseadas em componentes*: a componentização do software é uma das formas de se obter reusabilidade e também de lidar melhor com a complexidade.
- k) *Modelagem visual*: o princípio da abstração, por vezes, exige que o software seja entendido de forma visual. A UML (*Unified Modeling Language*) é usada para representar visualmente várias características da arquitetura de um sistema. Cada vez mais, espera-se que, a partir de modelos visuais, seja possível gerar sistemas completos sem passar pelas etapas mais clássicas de programação.
- l) *Verificação contínua da qualidade*: a qualidade de um produto de software não é um requisito que possa ser obtido no final ou ser facilmente melhorado ao final do desenvolvimento. A preocupação com a qualidade e sua garantia deve estar presente ao longo de qualquer projeto de desenvolvimento.
- m) *Controle de mudanças*: não é mais possível gerenciar adequadamente um processo de desenvolvimento sem ter o controle de versões de componentes do software e de sua evolução. Muitas vezes, é necessário voltar atrás em um caminho de desenvolvimento ou ainda bancar o desenvolvimento em paralelo de diferentes versões de um mesmo componente. Sem um sistema de gerenciamento de configuração e mudança, seria muito difícil, senão impossível, fazer esse controle de forma efetiva.
- n) *Gerenciamento de riscos*: é sabido que uma das principais causas de fracasso de projetos de software é o fato de que os planejadores e gerentes podem não estar atentos aos riscos do projeto. Quando um risco se torna um problema, um gerente que já tenha um plano de ação certamente vai se sair melhor do que aquele que é pego de surpresa. Além disso, com o gerenciamento de riscos, é possível realizar ações preventivas de forma a reduzir a probabilidade ou o impacto do risco.

Esses são apenas alguns dos princípios, e muitos outros poderiam ser adicionados. Isso será feito ao longo dos próximos capítulos, quando esses princípios e muitos outros serão mencionados nos diversos tópicos relacionados à área de Engenharia de Software.

PROCESSO E MODELOS DE PROCESSO

Dentre as áreas do SWEBOK, esta primeira parte do livro aborda “Processo de Engenharia de Software” e parte de “Ferramentas e Métodos de Engenharia de Software”.

Inicialmente, o Capítulo 2 define, de forma genérica, o que se entende por processo em Engenharia de Software, o que não deixa de ser semelhante ao conceito de processo em outras áreas da Engenharia e da Administração, pois se trata de um conjunto estruturado de atividades, com entradas, saídas, recursos e responsáveis bem definidos.

Os capítulos seguintes (3 a 5) apresentam os modelos de processo específicos da indústria de software, iniciando, no Capítulo 3, pelos *modelos prescritivos* (por vezes indevidamente chamados de *clássicos*).

O Capítulo 4 apresenta os principais *modelos ágeis*, que se diferenciam dos prescritivos por serem mais “leves”, mas principalmente por terem seu foco nos fatores humanos em vez das descrições detalhadas de atividades, mais típicas dos processos prescritivos.

O Capítulo 5 apresenta o *processo unificado* e suas implementações. Ele é apresentado como um capítulo à parte, primeiramente em razão de sua grande aceitação na indústria, o que tem feito dele um padrão de fato; em segundo lugar, ele não pode ser classificado nem como prescritivo nem como ágil, já que é um *framework* de processo que conta tanto com implementações prescritivas (como RUP e EUP) como com implementações ágeis (como AUP e OpenUP).



Capítulo

2

Processo

Este capítulo conceitua, de forma genérica, o que é um *processo de desenvolvimento de software* (Seção 2.1), sem detalhar este ou aquele modelo de ciclo de vida. Inicialmente, é mostrado que, tipicamente, um processo de desenvolvimento se subdivide em *fases* (Seção 2.2) com objetivos distintos, nas quais determinadas *disciplinas* (como análise de requisitos, implementação e teste) podem ser exercitadas com exclusividade ou predominância (Seção 2.3). Processos usualmente são definidos como conjuntos estruturados de *atividades* (Seção 2.4) para as quais são determinados artefatos de *entrada* e *saída*, papéis de *responsáveis* e *participantes*, além de *recursos* necessários. Atividades podem ser detalhadas pela definição de passos de execução, procedimentos e regras. Um exemplo de *documento descritivo* de uma atividade típica de processo é apresentado na Seção 2.5. Por fim, este capítulo tece alguns comentários sobre a *equipe de processo* (Seção 2.6), que deve ser composta por engenheiros de software de boa capacidade e algumas *normas técnicas* pertinentes (Seção 2.7).

Já foi mencionado que o desenvolvimento de software deve ser encarado como um processo para que possa ter mais qualidade e ser mais previsível e econômico. Mas o que é um processo? Segundo Sommerville (2003), *processo* é um conjunto de atividades e resultados associados que geram um produto de software.

Mas essa definição ainda é muito simples, pois em geral processos não são apenas conjuntos de atividades, mas atividades estruturadas. Além disso, além de atividades e resultados, há mais coisas envolvidas em um processo, como pessoas, recursos, restrições, padrões a serem seguidos etc.

A Wikipédia¹ apresenta uma definição mais apropriada:

Um processo de engenharia de software é formado por um conjunto de passos de processo parcialmente ordenados, relacionados com artefatos, pessoas, recursos, estruturas organizacionais e restrições, tendo como objetivo produzir e manter os produtos de software finais requeridos.

Um processo, então, pode ser entendido como um conjunto de *atividades*:

- a) Interdependentes.
- b) Com responsáveis.
- c) Com entradas e saídas definidas.

Ao longo deste capítulo, o processo de software será caracterizado de acordo com essas definições e seus detalhes.

¹Disponível em: <pt.wikipedia.org/wiki/Processos_de_Engenharia_de_Software> . Acesso em: 21 jan. 2013.

Convém, antes de continuar, distinguir os termos *processo* e *projeto* e a expressão *modelo de processo*.

Projeto é algo que ocorre em um tempo determinado. Consiste na execução concreta de um conjunto de atividades que visam à criação de um produto específico.

Processo é um conjunto de regras que definem como um projeto deve ser executado. No jargão da orientação a objetos, o projeto pode ser considerado uma *instância* de um processo. Um processo normalmente é adotado por uma empresa como um conjunto de regras específicas que seus funcionários devem seguir sempre que trabalharem em um projeto. Assim, quando um projeto precisa ser planejado na empresa, o responsável deve tomar o processo definido e, a partir dele, definir as atividades concretas, prazos e responsáveis.

Já *modelo de processo* é um conjunto de regras mais abstratas que especificam a forma geral de *processos*. Um modelo de processo apresenta uma filosofia, uma forma geral de comportamento com base na qual processos específicos podem ser definidos.

O modelo de processo para as atividades de projeto e desenvolvimento de software também pode ser chamado de *ciclo de vida*. Assim, quando uma empresa decide adotar um processo, deve buscar um modelo e adaptar a filosofia e as práticas recomendadas para criar seu próprio processo. A partir daí, todos os projetos da empresa deverão seguir o processo definido.

Há várias vantagens em definir o desenvolvimento de software como um processo, entre elas:

- a) *O tempo de treinamento pode ser reduzido*: com processos bem definidos e documentados, é mais fácil encaixar novos indivíduos na equipe do que quando não se dispõe deles.
- b) *Produtos podem ser mais uniformizados*: a existência do processo não garante a uniformidade na qualidade dos produtos, mas certamente uma equipe com um processo bem definido tende a ser mais previsível do que a mesma equipe sem processo algum.
- c) *Possibilidade de capitalizar experiências*: pode-se imaginar que um processo de trabalho bem definido poderia tolher o uso da criatividade dos desenvolvedores. Contudo, isso não é verdade, a não ser que a empresa tenha processos engessadores, o que não é bom. Um processo bem gerenciado deve ter embutidos mecanismos para melhoria. Assim, se um desenvolvedor descobrir um meio de fazer as coisas de maneira melhor do que a que está descrita no processo, deve haver meios para incorporar essas alterações.

As seções seguintes vão discorrer sobre a estrutura de processos.

2.1 Fases

Embora a nomenclatura possa variar de um modelo de processo para outro, usualmente se considera que a primeira grande divisão de um processo é a *fase*. Uma fase é um período de tempo no qual determinadas atividades com objetivos bem específicos são realizadas. As *fases* são, então, as grandes divisões dos processos, e normalmente sua quantidade é pequena (menos de 10).

Alguns processos, como o Modelo Cascata (Seção 3.2) e suas variantes, têm fases sequenciais, ou seja, com o passar do tempo o processo de desenvolvimento passa de uma fase a outra, como requisitos, análise, programação, testes e implantação.

Outros modelos podem ter fases cíclicas, ou seja, o desenvolvimento passa repetidamente de uma fase para outra, formando um ciclo repetitivo de fases até a finalização do projeto. Exemplos desse tipo de modelo são o Modelo Espiral (Seção 3.8) e o Modelo de Prototipação Evolucionária (Seção 3.9), além dos modelos ágeis (Capítulo 04).

O Processo Unificado (UP, Capítulo 05) é estruturado em quatro fases (embora algumas variantes tenham até seis fases), que são sequenciais no tempo. Dentro de cada fase, as atividades são organizadas de forma cíclica, ou seja, existem ciclos iterativos dentro das fases, mas elas são sequenciais.

Cada fase de um processo deve ter um macro-objetivo bem estabelecido. Por exemplo, no caso do UP, a fase de *concepção* tem como macro-objetivo uma primeira abordagem sobre o sistema e seus requisitos. A *elaboração* busca aprofundar a análise, detalhar o *design* do sistema e estabilizar sua arquitetura. A *construção* visa produzir código executável e testado. Finalmente, a *transição* visa à instalação e operação do sistema no ambiente final.

2.2 Disciplinas

O termo *disciplina* aparece com mais frequência no Processo Unificado, no qual as disciplinas equivalem grosseiramente às fases do Modelo Cascata, embora com outro significado. *Disciplina* é entendida como um conjunto de atividades ou tarefas correlacionadas, as quais servem a um objetivo específico dentro do processo de desenvolvimento.

Existem, por exemplo, disciplinas relacionadas à produção, como análise de requisitos, modelagem, programação etc., mas também existem disciplinas de apoio, como gerência de projeto, ambiente e gerência de configuração. Uma implementação do UP (EUP) também apresenta disciplinas relacionadas à empresa como um todo.

As disciplinas usualmente são compostas por tarefas ou atividades que se organizam em um grafo de dependências, que estabelece em que ordem (se for o caso) as atividades devem ser executadas.

No Processo Unificado, todas as disciplinas são exercitadas em todas as fases, cada uma com maior ou menor intensidade. Outros ciclos de vida não utilizam explicitamente o conceito de *disciplina*, sendo organizados normalmente em torno de tarefas ou atividades a serem executadas em cada fase.

2.3 Atividades ou Tarefas

A maioria dos processos de software é organizada em torno de *tarefas*, às vezes também chamadas de *atividades*. Toda atividade tem um objetivo principal estabelecido e visa criar ou produzir uma mudança de estado visível em um ou mais artefatos durante a execução de um projeto.

As atividades devem ter *entradas* e *saídas* bem definidas. Uma atividade toma artefatos de entrada e produz como saída novos artefatos e/ou promove uma modificação bem definida nos artefatos de entrada.

Atividades também devem ter identificados os responsáveis e participantes. *Responsáveis* são as pessoas que devem realizar a atividade e responder pela sua conclusão. Já os *participantes* agem apenas em resposta à iniciativa dos responsáveis. O ideal é que o responsável seja uma única pessoa (ou cargo). Por exemplo, uma atividade de levantamento de requisitos terá como responsável um analista e como participantes os clientes e usuários.

Cada atividade também necessita de um conjunto de *recursos* alocados, não apenas recursos humanos, que já estarão previstos na forma de responsáveis ou participantes, mas recursos físicos, como horas de computador, licenças de software, papel, passagens etc.

As atividades de um processo normalmente são descritas por um documento. Esse documento poderá ter as seguintes seções:

a) Cabeçalho:

- Nome do processo.
- Nome da fase (se cabível).
- Nome da atividade.
- Versão do documento.
- Responsável (cargo).
- Participantes (opcional).
- Artefatos de entrada (opcional).
- Artefatos de saída.
- Recursos alocados (opcional).

b) Corpo contendo o detalhamento da atividade.

Mais adiante, será visto um *template* para o documento de descrição e detalhamento de atividade. Nem sempre as atividades são executadas como uma sequência estrita. Nesses casos, pode ser interessante apresentá-las na forma de um algoritmo ou mesmo de um fluxograma ou diagrama de atividades UML.

Algumas vezes, as atividades podem ter descrições distintas, dependendo da fase.

2.3.1 ARTEFATOS

Artefatos são quaisquer documentos que puderem ser produzidos durante um projeto de desenvolvimento de software, incluindo diagramas, programas, documentos de texto, desenhos, contratos, projetos, planos etc.

Alguns modelos de processo (como UP) determinam que cada artefato tenha um dono, que é o único que pode modificá-lo ou permitir sua modificação.

Outros modelos (como XP) determinam que artefatos não tenham dono e que possam ser modificados à vontade por qualquer desenvolvedor, desde que exista uma boa razão para isso.

Em qualquer um dos casos, é importante que todos os artefatos estejam submetidos a um sistema de controle de versão (Capítulo 10) para que eventuais mudanças indevidas possam ser desfeitas.

2.3.2 RESPONSÁVEIS E PARTICIPANTES

Os *responsáveis* são perfis de pessoas ou cargos que respondem pela realização de uma atividade. Na prática, é interessante que qualquer atividade tenha um único responsável. Quando existem vários responsáveis pela mesma atividade, pode ocorrer de ninguém se sentir realmente responsável por ela.

Na descrição de um processo, as atividades devem ser atribuídas a *perfis* ou *cargos*, e não a pessoas. Apenas quando o processo for usado em um projeto concreto é que deve haver atribuições de atividades a pessoas. Por exemplo, uma atividade de análise de requisitos terá como responsável um analista, mas não é o caso de nomeá-lo na descrição da atividade que compõe a documentação do processo. A atribuição de atividades a pessoas deve ser feita durante o planejamento ou a gerência de um projeto.

Os *participantes* são todas as outras pessoas (perfis ou cargos) que precisam participar de alguma atividade para que ela seja concluída. Não são necessariamente responsáveis pela atividade, mas precisam participar dela. Por exemplo, o cliente deve participar da atividade de levantamento de requisitos, mas o responsável por ela é o analista.

2.3.3 RECURSOS

Uma atividade, para ser executada, pode demandar recursos. Existem *recursos humanos* e *recursos físicos*. Em geral, os recursos humanos são classificados à parte dos recursos físicos. Os recursos humanos são associados às atividades nos papéis de responsáveis e participantes. Os recursos físicos, de outro lado, dividem-se em dois grupos: *consumíveis* e *não consumíveis*.

Não se devem confundir os recursos com as entradas de uma atividade. *Entradas* são artefatos que servirão de fonte de informação ou que serão transformados na atividade para produzir os artefatos de saída. Já os *recursos* são ferramentas ou insumos usados na atividade.

Pode-se dizer que as entradas são específicas ao projeto, enquanto os recursos são genéricos. Assim, um *template* de documento é um recurso, mas um diagrama de classes de *design* é uma entrada. Uma ferramenta CASE é um recurso, mas um relatório de *status* de projeto gerado pela ferramenta é uma entrada.

2.3.3.1 Recursos Consumíveis

Recursos consumíveis são aqueles que são gastos quando usados. Por exemplo, folhas de papel, passagens etc. Esses recursos normalmente são alocados em determinada quantidade.

Por exemplo, para realizar uma reunião do projeto, podem-se alocar recursos consumíveis, como passagens e diárias para participantes de outras cidades, folhas de papel pra anotações, biscoitos e café. Nenhum recurso consumível pode ser reaproveitado em outra atividade.

2.3.3.2 Recursos Não Consumíveis

Recursos não consumíveis podem ser alocados inúmeras vezes para várias atividades, porém, normalmente não podem ser alocados para mais de uma atividade de cada vez. Exemplos de recursos não consumíveis são o software e o hardware. Por exemplo, enquanto um computador estiver sendo usado em uma atividade, não poderá ser simultaneamente usado em outra. Contudo, depois de liberado, poderá ser usado novamente.

Recursos não consumíveis podem sofrer desgaste e depreciação ao longo do tempo e um dia deixarem de ser aproveitáveis. Por exemplo, computadores podem parar de funcionar e versões de software podem ficar desatualizadas. Mas isso é gerenciado pelo processo de administração de recursos físicos da empresa como um todo, e normalmente

não é dentro de um projeto que se tenta determinar se o tempo de uso de um recurso não consumível já expirou. Então, para efeito de processo de desenvolvimento de software, esses recursos podem ser considerados inesgotáveis.

2.4 Detalhamento de Atividades

Atividades podem ser detalhadas em *passos*, complementadas por *procedimentos* e restringidas por *regras*.

2.4.1 PASSOS

As atividades de um processo podem ser detalhadas em *passos* individuais. Toda atividade necessita de uma descrição, que deve dizer, de modo simples e direto, o que precisa ser feito para que ela seja realizada.

Basicamente, deve-se informar como cada um dos artefatos de saída é produzido a partir dos artefatos de entrada. Isso deve ser feito usando-se um discurso essencial, ou seja, sem entrar em detalhes quanto às tecnologias a serem usadas. Detalhes de tecnologia devem ficar restritos aos procedimentos associados à atividade.

2.4.2 PROCEDIMENTOS

Uma atividade é descrita em termos gerais ou essenciais através de seus passos, que são isentos de tecnologia. Mas, por vezes, pode ser necessário informar como realizar determinados passos com uma tecnologia específica. Os *procedimentos* representam, então, uma realização tecnológica para o passo essencial definido. Para cada tecnologia poderá haver uma descrição de procedimentos distinta associada.

O procedimento é uma explicação adicional à atividade, indicando como realizá-la com as ferramentas e a tecnologia disponíveis. Uma atividade pode ser descrita de maneira semelhante a um caso de uso essencial (Wazlawick, 2011), ou seja, mencionando-se apenas o que deve ser feito sem detalhar a tecnologia a ser usada. Já o procedimento equivale a um caso de uso real, que detalha a atividade com uma tecnologia específica.

Pode haver mais de uma possibilidade tecnológica para realizar uma atividade. Por exemplo, pode haver disponíveis na empresa duas ferramentas CASE. Assim, uma atividade que consiste em realizar modelagem conceitual, por exemplo, terá uma única descrição, *independentemente de ferramenta*, mas haverá pelo menos dois procedimentos associados à atividade, um para cada ferramenta CASE disponível.

Se houver mudança de tecnologia na empresa, podem-se manter os procedimentos antigos para registro e, ao mesmo tempo, acrescentar os novos procedimentos. O interessante é observar que, como a descrição da atividade em si é feita no nível essencial, então ela deve valer para qualquer tecnologia possível.

2.4.3 REGRAS

A realização de uma atividade pode ainda ser condicionada por *regras* ou *restrições*, que podem se referir a passos, recursos, artefatos etc. Por exemplo, a atividade de escrever o sumário executivo do projeto pode ter como restrição o fato de que esse artefato não deve ter mais de duas páginas.

Pode-se fazer um paralelo entre atividades e regras com requisitos funcionais e não funcionais. As atividades correspondem aos requisitos funcionais (são coisas que devem ser executadas) e as regras aos requisitos não funcionais (a maneira como as coisas devem ser executadas) (Wazlawick, 2011).

2.5 Template e Exemplo de Documento de Atividade

Esta seção apresenta uma sugestão de *template* para documento descritivo de atividade e um exemplo. O *template* é apresentado apenas como referência, sendo que inúmeros outros modelos podem ser usados e seguidos. O *template* é mostrado na [Figura 2.1](#).

Evidentemente, trata-se apenas de uma proposta. Pode-se discutir, por exemplo, a praticidade de descrever os procedimentos em diferentes tecnologias usando-se cores variadas. Isso pode ser bastante útil na visualização, porque normalmente trabalha-se com uma tecnologia de cada vez e as informações sobre outras tecnologias são

| | |
|-------------------------------|---|
| Processo: | <nome do processo> |
| Fase: | <número e nome da fase> |
| Atividade: | <número e nome da atividade> |
| Versão: | <histórico de versões do documento> |
| Responsável: (obrigatório) | <cargo ou pessoa> |
| Participantes: (opcional) | <cargo ou papel 1> <cargo ou papel 2> ... |
| Entradas: (opcional) | <artefato 1> <artefato 2> ... |
| Saídas: (obrigatório) | <artefato 1> <artefato 2> ... |
| Recursos: (opcional) | <recurso 1> <recurso 2> ... |
| Passos: | |
| <passo 1>: | <descrição> <procedimento segundo tecnologia 1> <procedimento segundo tecnologia 2> ... • <regra 1> • <regra 2> ... |
| <passo 2>: | <descrição> <procedimento segundo tecnologia 1> <procedimento segundo tecnologia 2> ... • <regra 1> • <regra 2> ... |
| ... | ... |

Figura 2.1 Um template de atividade de projeto.

simplesmente inúteis nesse caso. Como em geral há poucas tecnologias possíveis (apenas uma ou duas), não haverá problemas para distinguir as cores.

A Figura 2.2 apresenta um exemplo de documento de descrição de atividade preenchido com uma atividade de captura de requisitos em um processo personalizado fictício baseado no Processo Unificado.

Observa-se que, no caso dos procedimentos suportados por ferramentas, é importante anotar também a versão da ferramenta para a qual esse procedimento foi escrito, visto que de uma versão para outra ele pode mudar. Cada vez que a ferramenta for atualizada no ambiente de trabalho, deve-se revisar se os procedimentos continuam os mesmos e registrar o novo procedimento, se for o caso. Alguns modelos de processo, como RUP, chamam os procedimentos específicos de ferramentas de “mentores de ferramentas” (Seção 5.3.1.5).

Uma pergunta que pode ser feita por quem ler a descrição de atividade da Figura 2.2 é: “Está claro o suficiente?”. O descriptivo de uma atividade não deve ser detalhado a ponto de ser cansativo para um analista que tenha alguma noção do que está fazendo. Porém, também não pode ser tão genérico a ponto de dois analistas produzirem resultados totalmente diferentes a partir dele. É necessário que cada passo esteja claramente definido, e quem vai determinar se isso está claro o suficiente são as pessoas que vão usar essa descrição de atividade.

Se os usuários desse documento acharem que alguma parte não está suficientemente clara, devem solicitar mudanças ao engenheiro de software que cuida do processo.

O documento de processo não é estático. Ele vai evoluindo com o passar do tempo e deve ser mantido sob controle de versões (Capítulo 10). Em empresas com maior maturidade, pode-se dizer que ele vai sendo sistematicamente *otimizado*.

| | |
|----------------|--|
| Processo: | MP – Meu Processo |
| Fase: | 1. Concepção |
| Atividade: | 1.3 Captura de Requisitos a partir das Entrevistas |
| Versão: | 1.0 inicial |
| Responsável: | Analista de requisitos |
| Participantes: | - |
| Entradas: | 1. Transcrição de entrevistas com o cliente. 2. Sumário executivo do projeto. 3. Definição de escopo do projeto. |
| Saídas: | 1. Documento de requisitos iniciais. |
| Recursos: | 1. Template de documento de requisitos. 2. Ferramenta CASE (EA v6.0 ou VP v.8.3). |
| Passos: | <p>1. Listar requisitos funcionais dos candidatos. EA v6.0: Criar diagrama de requisitos e criar uma caixa para cada requisito candidato preenchendo o texto do requisito no campo “description” VP v8.3: Criar um diagrama de requisitos e uma classe estereotipada como <>requirement<> para cada requisito, preenchendo o texto do requisito no atributo “text”, e preenchendo o atributo “kind” com “functional”. • Numerar os requisitos funcionais como RF01, RF02, ... • Iniciar sempre com verbo no infinitivo.</p> <p>2. Listar requisitos suplementares e não funcionais. EA v6.0: Criar requisitos suplementares em um pacote separado dos funcionais. Indicar os requisitos não funcionais após o texto do requisito funcional associado, indicado pela marca “RESTRICOES.”. VP v8.3: Criar requisitos suplementares em um pacote separado. Criar requisitos não funcionais, como classes estereotipadas do diagrama com atributo “kind” preenchido com o tipo do requisito (interface, segurança, ...) • Associar requisitos não funcionais a algum requisito funcional. • Classificar requisitos suplementares pelo seu tipo: interface, segurança, tolerância a falhas, performance etc. • Não criar requisitos desnecessários.</p> <p>3. Agrupar requisitos funcionais em pacotes. • Não permitir que mais de 20 requisitos estejam em cada pacote, a não ser em casos em que se trate efetivamente de requisitos altamente coesos. • Agrupar os requisitos em pacotes por afinidade, ou seja, de modo que os requisitos mais próximos sejam aqueles que tratam dos mesmos objetos. • Requisitos do tipo inserir, alterar, remover e consultar sobre um objeto devem ser agrupados em um único requisito “gerenciar”, estereotipado como <>crud<>.</p> <p>4. Gerar o documento de requisitos. EA v6.0: Usar o gerador de documentação acessível a partir do menu superior. VP v8.3: Usar a opção “generate report” disponível no menu superior. • Devem ser geradas uma versão PDF para impressão e uma versão HTML, que ficará on-line na intranet do projeto.</p> |

Figura 2.2 Um exemplo de atividade descrita de acordo com o template apresentado.

De outro lado, é altamente recomendável que tais documentos sejam elaborados como hipertextos que o leitor possa ler no nível de detalhe que lhe interessa no momento. Por exemplo, apenas os passos das atividades, sem as descrições, ou as descrições sem as regras, ou ainda as descrições, regras e procedimentos referentes a apenas uma tecnologia. Desenvolvedores mais experientes possivelmente precisarão apenas lembrar-se da sequência de passos e fazer um *checklist* dos artefatos, ao passo que desenvolvedores iniciantes, ou que estão executando processos novos, precisarão de maior detalhamento em relação às atividades.

Uma das ferramentas bastante usadas para produzir documentos de atividade como hipertextos, gratuita, é o *OpenWiki*².

2.6 Equipe de Processo

As organizações devem ter *equipes de processo* constituídas por um ou mais engenheiros de software, que serão responsáveis pela manutenção, avaliação e otimização do processo.

²Disponível em: <www.openwiki.com/ow.asp?OpenWiki>. Acesso em: 21 jan. 2013.

O *Software Engineering Institute*³ (SEI) publicou um documento, disponível *on-line*, sobre como estabelecer equipes de processo de engenharia de software que podem ser de grande valia como referência (Fowler & Rifkin, 1990)⁴. Segundo esse documento, a equipe de processo é o ponto focal da melhoria de processos em uma empresa. O tamanho do grupo deve variar entre 1 e 3% do número de profissionais da empresa ligados ao desenvolvimento de software, e ele centraliza e capitaliza o esforço colaborativo dos mais diferentes agentes no sentido da melhoria contínua do processo adotado na empresa.

Organizações muito pequenas poderão ter um funcionário alocado no processo em tempo parcial.

2.7 Norma NBR ISO/IEC 12207

Pode-se agora perguntar: quais outros processos, tarefas ou atividades devem ser formalmente definidos quando se está estabelecendo um processo de desenvolvimento de software?

Existe uma norma técnica denominada ISO/IEC 12207:2008, adotada internacionalmente (inclusive no Brasil, como NBR), a qual estabelece definições e padrões referentes a vários processos relacionados com a indústria de software. Ela estabelece processos, atividades e tarefas que devem ser aplicados durante a aquisição, o fornecimento, o desenvolvimento, a operação, a manutenção e o descarte de software.

A norma pode ser adquirida no *site* da ISO⁵. Existe também uma norma, a IEEE 12207, que consiste na adoção e adaptação da ISO/IEC 12207 pelo IEEE (*Institute of Electrical and Electronic Engineers*). Esse documento pode ser adquirido diretamente no *site* do IEEE⁶. Gray (1999)⁷ apresenta um comparativo entre as duas normas.

No Brasil, a ABNT adotou a norma ISO/IEC 12207, transformando-a também em padrão brasileiro: a NBR ISO/IEC 12207:2009. Essa norma pode ser adquirida no *site* da ABNT⁸.

A ISO/IEC 12207 e suas adaptações apresentam definições e conceitos que são independentes do ciclo de vida escolhido e, portanto, podem ser aplicados a variados contextos.

Para efeito de organização, a 12207 divide os processos em quatro grandes famílias:

- a) *Processos fundamentais*, que são necessários para que um software seja construído e executado.
- b) *Processos de apoio*, que auxiliam outros processos (garantindo qualidade, por exemplo), mas não são fundamentais.
- c) *Processos organizacionais*, que são usados no contexto da organização para permitir o melhor acompanhamento e gerenciamento dos projetos.
- d) *Processo de adaptação*, em que a norma estabelece como pode ser aplicada a uma organização ou a um projeto específico.

Os *processos fundamentais* são os mais fortemente relacionados ao ciclo de vida do software. Esses processos constituem-se em:

- a) *Aquisição*: visa à obtenção do produto ou serviço relacionado à informática que satisfaça as necessidades da empresa.
- b) *Fornecimento*: seu objetivo é fornecer um produto ou serviço a terceiros.
- c) *Desenvolvimento*: é definido como o processo de transformar um conjunto de requisitos em um produto executável.
- d) *Operação*: tem como objetivo iniciar e manter o produto operando em seu local definitivo, bem como prestar serviços aos usuários.
- e) *Manutenção*: seu propósito é modificar o produto, removendo erros e adequando-o a novos contextos (Capítulo 10).

³Disponível em: <www.sei.cmu.edu>. Acesso em: 21 jan. 2013.

⁴Disponível em: <www.sei.cmu.edu/reports/90tr024.pdf>. Acesso em: 21 jan. 2013.

⁵Disponível em: <http://www.iso.org/iso/catalogue_detail.htm?csnumber=43447>. Acesso em: 21 jan. 2013.

⁶Disponível em: <www.ieee.org>. Acesso em: 21 jan. 2013.

⁷Disponível em: <www.abelia.com/docs/122_016.pdf>. Acesso em: 21 jan. 2013.

⁸Disponível em: <www.abntcatalogo.com.br/norma.aspx?ID=38643>. Acesso em: 21 jan. 2013.

Os *processos de apoio* têm como objetivo apoiar os processos fundamentais, mas não são eles que compõem as atividades de desenvolvimento propriamente ditas. Esses processos são:

- a) *Documentação*: tem como propósito criar e manter informações sobre o produto e o processo de desenvolvimento.
- b) *Gerência de configuração*: seu objetivo é gerenciar e manter a consistência entre todas as versões dos produtos do trabalho, de forma a manter também sua integridade (Capítulo 10).
- c) *Garantia de qualidade*: visa garantir que os produtos e serviços estejam em conformidade com normas e padrões predefinidos, sendo consistentes em relação aos requisitos (Capítulo 11).
- d) *Verificação*: tem como propósito garantir ou confirmar que os produtos refletem os requisitos especificados.
- e) *Validação*: objetiva garantir ou confirmar que os requisitos especificados são os realmente desejados pelo cliente.
- f) *Revisão conjunta*: visa manter um entendimento comum entre os diversos interessados a respeito do produto, do processo ou do serviço.
- g) *Auditoria*: seu propósito é prover uma avaliação, independentemente dos produtos e processos.
- h) *Resolução de problemas*: visa assegurar que todos os problemas levantados sejam resolvidos.

Os *processos organizacionais* garantem o funcionamento da organização. São eles:

- a) *Gerência*: objetiva organizar e controlar a realização dos projetos, bem como seu desempenho (Capítulo 8).
- b) *Infraestrutura*: visa manter um ambiente de trabalho adequado.
- c) *Melhoria*: tem como propósito permitir que os processos sejam continuamente adaptados, visando à otimização do trabalho.
- d) *Treinamento ou recursos humanos*: seu objetivo é manter os recursos humanos capacitados para o melhor desempenho possível de suas funções.

O *processo de adaptação* definido pela norma indica como ela pode ser aplicada a diferentes empresas, já que de uma para a outra podem variar a cultura organizacional, o modelo de ciclo de vida utilizado no processo de desenvolvimento e outros fatores.

Pode-se mencionar ainda a norma ISO/IEC 15504, que será discutida em detalhes no Capítulo 12, por tratar de qualidade e níveis de capacidade em processo de software. A norma 15504, também conhecida como SPICE, é considerada uma evolução da ISO/IEC 12207.

Modelos de Processo Prescritivos

Este capítulo apresenta os modelos de processo prescritivos, ou seja, aqueles que se baseiam em uma descrição de como as atividades são feitas. Inicialmente é apresentado o *antimodelo* por excelência, que é *Codificar e Consertar* (Seção 3.1), que consiste na absoluta falta de processo. O *Modelo Cascata* (Seção 3.2) introduz a noção de fases bem definidas e a necessidade de documentação ao longo do processo, e não apenas para o código final. O *Modelo Sashimi* (Seção 3.3) relaxa a restrição sobre o início e o fim estanques das fases do Modelo Cascata. O *Modelo V* (Seção 3.4) é uma variação do Modelo Cascata, que enfatiza a importância dos testes em seus vários níveis. O *Modelo W* (Seção 3.5) enriquece o Modelo V com um conjunto de fases de planejamento de testes em paralelo com as atividades de análise, e não apenas no final do processo. O *Modelo Cascata com Subprojetos* (Seção 3.6) introduz a possibilidade de dividir para conquistar, em que subprojetos podem ser desenvolvidos em paralelo ou em momentos diferentes, e o *Modelo Cascata com Redução de Risco* (Seção 3.7) enfatiza a necessidade de tratar inicialmente os maiores riscos e incertezas do projeto antes de se iniciar um processo de desenvolvimento com fases bem definidas. O *Modelo Espiral* (Seção 3.8) é uma organização de ciclo de vida voltada ao tratamento de risco, iteratividade e prototipação. A *Prototipação Evolucionária* (Seção 3.9) é uma técnica que pode ser entendida como um modelo independente ou parte de outro modelo em que protótipos cada vez mais refinados são apresentados ao cliente para que o entendimento sobre os requisitos evolua de forma suave e consistente. O *Modelo Entrega em Estágios* (Seção 3.10) estabelece que partes do sistema já prontas podem ser entregues antes de o projeto ser finalizado e que isso possa ser planejado. O *Modelo Orientado a Cronograma* (Seção 3.11) indica que se podem priorizar requisitos de forma que, se o tempo disponível acabar antes do projeto, pelo menos os requisitos mais importantes terão sido incorporados. A *Entrega Evolucionária* (Seção 3.12) é um misto de Prototipação Evolucionária e Entrega em Estágios em que se pode adaptar o cronograma de desenvolvimento a necessidades de última hora, em função de prioridades definidas. Os *modelos orientados a ferramentas* (Seção 3.14) são modelos específicos de ferramentas CASE e geradores de código aplicados com essas ferramentas. Por fim, o capítulo apresenta as *Linhas de Produto de Software* (Seção 3.14), que são uma abordagem moderna para a reusabilidade planejada em nível organizacional.

Para que o desenvolvimento de sistemas deixe de ser artesanal e aconteça de forma mais previsível e com maior qualidade, é necessário que se compreenda e se estabeleça um processo de produção.

Processos normalmente são construídos de acordo com modelos ou estilos. *Modelos de processo* aplicados ao desenvolvimento de software também são chamados de “ciclos de vida”. Pode-se afirmar que existem duas grandes famílias de modelos: os *prescritivos*, abordados neste capítulo, e os *ágeis*, que serão apresentados no Capítulo 4.

O modelo prescritivo mais emblemático é o *Waterfall* ou *Cascata* (com suas variações), cuja característica é a existência de fases bem definidas e sequenciais. Também pode-se citar o Modelo Espiral, cuja característica é a realização de ciclos de prototipação para a redução de riscos de projeto.

Cada um dos ciclos de vida, sejam aplicados ou não no desenvolvimento de software atualmente, trouxe uma característica própria. As características inovadoras dos diferentes modelos foram, em grande parte, capitalizadas pelo Processo Unificado descrito no Capítulo 5. Pode-se resumir da seguinte forma a contribuição de cada um dos modelos listados neste livro:

- a) *Codificar e consertar*: é considerado o modelo *ad-hoc*, ou seja, aquele que acaba sendo usado quando não se utiliza conscientemente nenhum modelo. Sendo assim, ele também não traz nenhuma contribuição, sendo considerado o “marco zero” dos modelos.
- b) *Cascata*: introduz a noção de que o desenvolvimento de software ocorre em fases bem definidas e de que é necessário produzir não apenas um código executável, mas também documentos que ajudem a visualizar o sistema de forma mais abstrata do que o código.
- c) *Sashimi*: derivado do Modelo Cascata, introduz a noção de que as fases do processo de desenvolvimento não são estanques, mas que em determinado momento ele pode estar simultaneamente em mais de uma dessas fases.
- d) *Cascata com subprojetos*: introduziu a divisão do projeto em subprojetos de menor porte – seu lema é “dividir para conquistar”. Os projetos menores podem ser desenvolvidos em paralelo por várias equipes ou por uma única equipe, em diferentes momentos, o que se assemelha ao desenvolvimento iterativo em ciclos. Porém, nesse modelo, a integração costuma ocorrer apenas no final do desenvolvimento dos subprojetos.
- e) *Cascata comma redução de risco e espiral*: introduzem a noção de que riscos são fatores determinantes para o sucesso de um projeto de software e, portanto, devem ser os primeiros aspectos analisados. Assim, uma espiral de desenvolvimento se inicia onde, a cada ciclo, um ou mais riscos são resolvidos ou minimizados. Apenas ao final desses ciclos de redução de risco é que o desenvolvimento propriamente dito se inicia.
- f) *Modelo V e modelo W*: enfatizam a importância do teste no desenvolvimento de software e indicam que essa deve ser uma preocupação constante, e não apenas uma etapa colocada ao final do processo de desenvolvimento.
- g) *Modelo orientado a cronograma*: introduziu a noção de que se pode trabalhar prioritariamente com as funcionalidades mais importantes, deixando as menos importantes para o final. Assim, se houver atrasos e o prazo for rígido, pelo menos as funcionalidades mais importantes serão entregues no prazo.
- h) *Entrega em estágios*: introduz a noção de que é possível planejar e entregar partes prontas do sistema antes do final do projeto.
- i) *Prototipação evolucionária*: propõe o uso de protótipos para ajudar na compreensão da arquitetura, da interface e dos requisitos do sistema, o que é extremamente útil quando não é possível conhecer bem esses aspectos *a priori*.
- j) *Entrega evolucionária*: combina a prototipação evolucionária com a entrega em estágios, mostrando que é possível fazer um planejamento adaptativo em que, a cada nova iteração, o gerente de projeto decide se vai acomodar as requisições de mudança que surgiram ao longo do projeto ou manter-se fiel ao planejamento inicial.
- k) *Modelos orientados a ferramentas*: é uma família de modelos cuja única semelhança costuma consistir no fato de que eles são indicados para uso com ferramentas específicas.
- l) *Linhas de produto de software*: é um tipo de processo que se aplica somente se a empresa vai desenvolver uma família de produtos semelhantes. A linha de produto de software permite, então, que a reusabilidade de componentes seja efetivamente planejada, e não apenas fortuita.
- m) *Modelos ágeis*: é uma família de modelos cujo foco está nos fatores humanos, e não na descrição de tarefas, o que os diferencia dos modelos prescritivos.
- n) *Processo unificado*: busca capitalizar e aprimorar todas as características dos modelos anteriores, consistindo de um modelo iterativo, focado em riscos, que valoriza o teste e a prototipação, entre outras características. Ele possui diferentes implementações, como a clássica RUP, as ágeis AUP e OpenUP, a orientada a ferramentas OUM e a específica para sistemas de grande porte, RUP-SE.

Portanto, existem vários tipos de ciclos de vida. Alguns vêm caindo em desuso, enquanto outros evoluem a partir deles. O engenheiro de software deve escolher o que for mais adequado a sua equipe e ao projeto que ele vai desenvolver. Se escolher bem, terá um processo eficiente, baseado em padrões e lições aprendidas e com possibilidade de capitalizar experiências. O controle será eficiente, e os riscos, erros e retrabalho serão minimizados.

Entretanto, se o engenheiro de software escolher um modelo inadequado para sua realidade, poderá gerar trabalho repetitivo e frustrante para a equipe, o que, aliás, pode acontecer também quando não se escolhe modelo algum.

Projetos diferentes têm necessidades diferentes. Assim, não há um modelo que seja sempre melhor do que os outros. Nem mesmo o processo unificado se adapta bem a sistemas cujos requisitos não sejam baseados em casos de uso, como o software científico, os jogos e os compiladores. Para escolher um ciclo de vida, ou pelo menos as características de processo necessárias para seu projeto e empresa, o engenheiro de software pode tentar responder às seguintes perguntas:

- a) *Quão bem os analistas e o cliente podem conhecer os requisitos do sistema?* O entendimento sobre o sistema poderá mudar à medida que o desenvolvimento avançar? Se os requisitos são estáveis, pode-se trabalhar com modelos mais previsíveis, como o Modelo Cascata ou o Modelo V. Requisitos instáveis ou mal compreendidos, porém, exigem ciclos de redução de risco e modelos baseados em prototipação, espiral ou ainda métodos ágeis.
- b) *Quão bem é compreendida a arquitetura do sistema?* É provável que sejam feitas grandes mudanças de rumo ao longo do projeto? Uma arquitetura bem compreendida e estável permite o uso de modelos como o Cascata com Subprojetos, mas arquiteturas mal compreendidas precisam de protótipos e de um desenvolvimento iterativo para reduzir o risco.
- c) *Qual o grau de confiabilidade necessário em relação ao cronograma?* O Modelo Orientado a Cronograma, o Processo Unificado e os Métodos Ágeis prezam o cronograma, ou seja, na data definida esses modelos deverão permitir a entrega de alguma funcionalidade (possivelmente não toda, mas alguma coisa estará pronta). Já os modelos Cascata, Espiral e Prototipação são bem menos previsíveis em relação ao cronograma.
- d) *Quanto planejamento é efetivamente necessário?* Os modelos prescritivos costumam privilegiar o planejamento com antecedência. Já os modelos ágeis preferem um planejamento menos detalhado e a adaptação às condições do projeto à medida que ele vai evoluindo. O Processo Unificado faz um planejamento genérico a longo prazo e um planejamento detalhado para o próximo ciclo iterativo que vai iniciar.
- e) *Qual é o grau de risco que esse projeto apresenta?* Existem ciclos de vida especialmente voltados à minimização dos riscos de projeto nos primeiros instantes, entre eles Modelo Espiral, Cascata com Redução de Risco, Métodos Ágeis e o Processo Unificado.
- f) *Existe alguma restrição de cronograma?* Se a data de entrega do sistema é definitiva e inadiável, o Modelo Orientado a Cronograma ou uma variante deste deveria ser escolhida.
- g) *Será necessário entregar partes do sistema funcionando antes de terminar o projeto?* Alguns ciclos de vida, como Cascata com Subprojetos, preveem a integração do software apenas no final do desenvolvimento. Já os Métodos Ágeis e o Processo Unificado sugerem que se pratique a integração contínua, o que pode viabilizar a entrega de partes do sistema ao longo do processo de desenvolvimento.
- h) *Qual é o grau de treinamento e adaptação necessário para a equipe poder utilizar o ciclo de vida que parece mais adequado ao projeto?* Nenhum ciclo de vida, exceto Codificar e Consertar, é trivial. Todos exigem certa dose de preparação da equipe. Porém, os prescritivos, por definirem as tarefas detalhadamente, podem ser mais adequados a equipes inexperientes, enquanto os métodos ágeis, focados em valores humanos, usualmente necessitam de desenvolvedores mais experientes.
- i) *Será desenvolvido um único sistema ou uma família de sistemas semelhantes?* Caso mais de dois sistemas semelhantes sejam desenvolvidos, pode ser o caso de investir em uma linha de produtos de software, que permite lidar com as partes em comum e as diferenças entre os sistemas, aplicando o reúso de forma planejada e institucionalizada.
- j) *Qual o tamanho do projeto?* Projetos que possam ser realizados por equipes pequenas (de até 8 ou 10 desenvolvedores) adequam-se melhor aos modelos ágeis, enquanto os projetos de grande porte precisam de processos mais formais, como *RUP-SE*. A família *Crystal* (Seção 4.5), de outro lado, é um modelo de processos que se adapta ao tamanho do projeto.

Em geral, o que se observa é que é mais útil escolher um modelo de processo simples, mas executá-lo coerentemente e de forma bem gerenciada, do que escolher um modelo sofisticado, porém executá-lo e gerenciá-lo mal.

3.1 Codificar e Consertar

O Modelo Codificar e Consertar (*Code and Fix*) tem uma filosofia muito simples, mostrada na Figura 3.1.

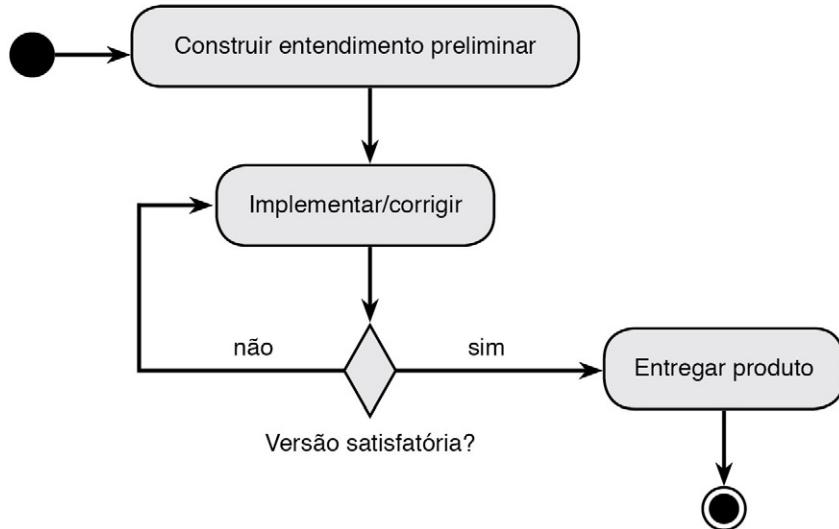


Figura 3.1 Modelo Codificar e Consertar¹.

Em resumo, esse modelo consiste em:

- a) Construir com o cliente um entendimento preliminar sobre o sistema que deve ser desenvolvido.
- b) Implementar uma primeira versão desse sistema.
- c) Interagir com o cliente de forma a corrigir a versão preliminar até que esta satisfaça o cliente.
- d) Fazer testes e corrigir os erros inevitáveis.
- e) Entregar o produto.

Pode-se dizer que se trata de uma forma bastante ingênua de modelo de processo e também que sequer parece um, pois não há previsibilidade em relação às atividades e aos resultados obtidos. Esse modelo é usado porque é simples, não porque funciona bem. Muitos projetos reais, mesmo dirigidos por outros modelos de ciclo de vida, por vezes caem na prática de Codificar e Consertar, por causa da pressão do cronograma.

Empresas que não usam modelo de processo algum possivelmente utilizam o Modelo Codificar e Consertar por *default*.

Possivelmente, esse modelo é bastante usado em empresas de pequeno porte, mas deve ser entendido mais como uma maneira de pressionar os programadores (*code rush*) do que como um processo organizado (McConnell, 1996)². Se o sistema não satisfaz o cliente, cobra-se do programador uma versão funcional adequada no menor tempo possível.

Apesar de tudo, esse modelo ainda pode ser usado quando se trata de desenvolver sistemas muito pequenos em intervalos de poucos dias. Também pode ser usado para implementar sistemas que serão descartados, como provas de conceito ou protótipos (na abordagem de prototipação *throw-away*). Suas vantagens são:

- a) Não se gasta tempo com documentação, planejamento ou projeto: vai-se direto à codificação.
- b) O progresso é facilmente visível à medida que o programa vai ficando pronto.
- c) Não há necessidade de conhecimentos ou treinamento especiais. Qualquer pessoa que programe pode desenvolver software com esse modelo de ciclo de vida.

Um dos problemas com essa técnica é que o código, que já sofreu várias correções, fica cada vez mais difícil de ser modificado (Capítulo 14). Além disso, aquilo de que o cliente menos precisa é um código ruim produzido rapidamente – ele precisa ter certas necessidades atendidas. Essas necessidades são levantadas na fase de requisitos, que, se for feita às pressas, deixará de atingir seus objetivos. Outras desvantagens são:

¹Todos os diagramas do livro seguem a notação UML, exceto quando o contrário for explicitamente informado.

²Disponível em: <my.safaribooksonline.com/9780735634725>. Acesso em: 21 jan. 2013.

- a) É muito difícil avaliar a qualidade e os riscos do projeto.
- b) Se no meio do projeto a equipe descobrir que as decisões arquiteturais estavam erradas, não há solução, a não ser começar tudo de novo.

Além disso, usar o Modelo Codificar e Consertar sem um plano de teste sistemático tende a produzir sistemas instáveis e com grande probabilidade de conter erros.

3.2 Modelo Cascata

O *Modelo Cascata* (*Waterfall* ou *WFM*) começou a ser definido nos anos 1970 e apresenta um ciclo de desenvolvimento bem mais detalhado e previsível do que o Modelo Codificar e Consertar. O Modelo Cascata é considerado o “avô” de todos os ciclos de vida e baseia-se na filosofia *BDUF* (*Big Design Up Front*): ela propõe que, antes de produzir linhas de código, deve-se fazer um trabalho detalhado de análise e projeto, de forma que, quando o código for efetivamente produzido, esteja o mais próximo possível dos requisitos do cliente.

O modelo prevê uma atividade de revisão ao final de cada fase para que se avalie se o projeto pode passar à fase seguinte. Se a revisão mostrar que o projeto não está pronto para passar à fase seguinte, deve permanecer na mesma fase (Ellis, 2010)³.

O Modelo Cascata é dirigido por documentação, já que é ela que determina se as fases foram concluídas ou não.

Boehm (1981) apresenta uma série de marcos (*milestones*) que podem ser usados para delimitar as diferentes fases do Modelo Cascata:

- a) Início da fase de planejamento e requisitos. Marco LCR, *Life-cicle Concept Review*, completar a revisão dos conceitos do ciclo de vida:
 - Arquitetura de sistema aprovada e validada, incluindo questões básicas de hardware e software.
 - Conceito de operação aprovado e validado, incluindo questões básicas de interação humano-computador.
 - Plano de ciclo de vida de alto nível, incluindo marcos, recursos, responsabilidades, cronogramas e principais atividades.
- b) Fim da fase de planejamento e requisitos – início da fase de design de produto. Marco SRR, *Software Requirements Review*, completar a revisão dos requisitos do software:
 - Plano de desenvolvimento detalhado: detalhamento de critérios de desenvolvimento de marcos, orçamento e alocação de recursos, organização da equipe, responsabilidades, cronograma, atividades, técnicas e produtos a serem usados.
 - Plano de uso detalhado: contraparte para os itens do plano de desenvolvimento como treinamento, conversão, instalação, operações e suporte.
 - Plano de controle de produto detalhado: plano de gerenciamento de configuração, plano de garantia de qualidade, plano geral de V&V (verificação e validação), excluindo detalhes dos planos de testes.
 - Especificações de requisitos de software aprovadas e validadas: requisitos funcionais, de performance e especificações de interfaces validadas em relação a completude, consistência, testabilidade e exequibilidade.
 - Contrato de desenvolvimento (formal ou informal) aprovado com base nos itens anteriores.
- c) Fim da fase de *design* de produto – início da fase de *design* detalhado. Marco PDR, *Product Design Review*, completar a revisão do *design* do produto:
 - Especificação do *design* do produto de software verificada.
 - Hierarquia de componentes do programa, interfaces de controle e dados entre as unidades (uma *unidade* de software realiza uma função bem definida, pode ser desenvolvida por uma pessoa e costuma ter de 100 a 300 linhas de código).
 - Estruturas de dados lógicas e físicas detalhadas em nível de seus campos.
 - Orçamento para recursos de processamento de dados (incluindo especificações de eficiência de tempo, capacidade de armazenamento e precisão).
 - Verificação do *design* com referência a completude, consistência, exequibilidade e rastreabilidade dos requisitos.

³Disponível em: <www.eng.uwi.tt/depts/civil/pgrad/proj_mgmt/courses/prmg6009/Notes%20PDF/04_L3.pdf>. Acesso em: 21 jan. 2013.

- Identificação e resolução de todos os riscos de alta importância.
 - Plano de teste e integração preliminar, plano de teste de aceitação e manual do usuário.
- d) Fim da fase de *design* detalhado – início da fase de codificação e teste de unidade. Marco CDR, *Critical Design Review*, completar o *design* e revisar aspectos críticos dele das unidades:
- Especificação de *design* detalhado revisada para cada unidade.
 - Para cada rotina (menos de 100 instruções) dentro de uma unidade, especificar nome, propósito, hipóteses, tamanho, sequência de chamadas, entradas, saídas, exceções, algoritmos e fluxo de processamento.
 - Descrição detalhada da base de dados.
 - Especificações e orçamentos de *design* verificados em relação a completude, consistência e rastreabilidade dos requisitos.
 - Plano de teste de aceitação aprovado.
 - Manual do usuário e rascunho do plano de teste e integração completados.
- e) Fim da fase de codificação e teste de unidade – início da fase de integração e teste. Marco UTC, *Unit Test Criteria*, satisfação dos critérios de teste de unidade:
- Verificação de todas as unidades de computação usando-se não apenas valores nominais, mas também valores singulares e extremos (Seção 13.5.2).
 - Verificação de todas as entradas e saídas unitárias, incluindo mensagens de erro.
 - Exercício de todos os procedimentos executáveis e todas as condições de teste (Seção 13.4).
 - Verificação de conformação a padrões de programação.
 - Documentação em nível de unidade completada.
- f) Fim da fase de integração e teste – início da fase de implantação. Marco SAR, *Software Acceptance Review*, completar a revisão da aceitação do software:
- Testes de aceitação do software satisfeitos.
 - Verificação da satisfação dos requisitos do software.
 - Demonstração de performance aceitável acima do nominal, conforme especificado.
 - Aceitação de todos os produtos do software: relatórios, manuais, especificações e bases de dados.
- g) Fim da fase de implantação – início da fase de operação e manutenção. Marco SyAR, *System Acceptance Review*, completar a revisão da aceitação do sistema:
- Satisfação do teste de aceitação do sistema.
 - Verificação da satisfação dos requisitos do sistema.
 - Verificação da prontidão operacional de software, hardware, instalações e pessoal.
 - Aceitação de todas as entregas relacionadas ao sistema: hardware, software, documentação, treinamento e instalações.
 - Todas as conversões especificadas e atividades de instalação foram completadas.
- h) Fim da fase de operação e manutenção. Corresponde à aposentadoria do sistema:
- Foram completadas todas as atividades do plano de aposentadoria: conversão, documentação, arquivamento e transição para um novo sistema.

As fases e marcos, conforme apresentados por Boehm, são resumidos na [Figura 3.2](#).

As ideias fundamentais do Modelo Cascata são coerentes, em uma primeira abordagem, e podem gerar benefícios relevantes:

- a) A existência de fases bem definidas ajuda a detectar erros cedo; dessa forma, é mais barato corrigi-los.
- b) O modelo procura promover a estabilidade dos requisitos; assim, o projeto só segue em frente quando os requisitos são aceitos.
- c) Funciona bem com projetos nos quais os requisitos são bem conhecidos e estáveis, já que esse tipo de projeto se beneficia de uma abordagem organizada e sistemática.
- d) Funciona bem quando a preocupação com a qualidade está acima das preocupações com custo ou tempo de desenvolvimento, visto que a eliminação de mudanças ao longo do projeto minimiza uma conhecida fonte de erros.
- e) É adequado para equipes tecnicamente fracas ou inexperientes, pois dá estrutura ao projeto, servindo de guia e evitando esforço inútil.

Um dos problemas com essa abordagem é que, em geral, é fácil verificar se o código funciona direito, mas não é tão fácil verificar se modelos e projetos estão bem escritos. Para ser efetivamente viável, esse tipo de ciclo de vida

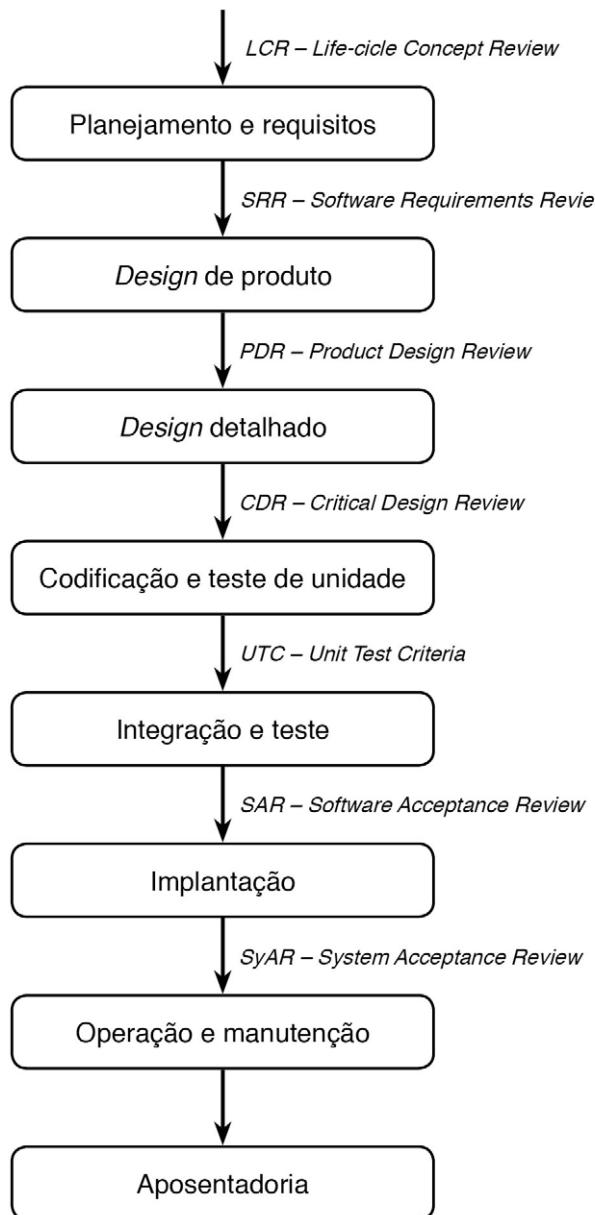


Figura 3.2 Fases e marcos do Modelo Cascata, de acordo com Boehm.

necessitaria de ferramentas de análise automatizada de diagramas e documentos para verificar sua exatidão, mas tais ferramentas ainda são bastante limitadas.

O Modelo Cascata também tem sido um dos mais criticados da história, especialmente pelos adeptos dos modelos ágeis, que valorizam princípios diametralmente opostos aos desse modelo.

Ellis (2010) aponta uma série de problemas com o Modelo Cascata:

- a) *Não produz resultados tangíveis até a fase de codificação*, exceto para as pessoas familiarizadas com as técnicas de documentação, que poderão ver significado nos documentos.
- b) *É difícil estabelecer requisitos completos antes de começar a codificar*: hoje, o desenvolvimento de software é entendido mais como um processo de amadurecimento do que como uma construção que pode ser baseada em um projeto detalhado desde o início. É natural que alguns requisitos só sejam descobertos durante o desenvolvimento de um projeto de software.

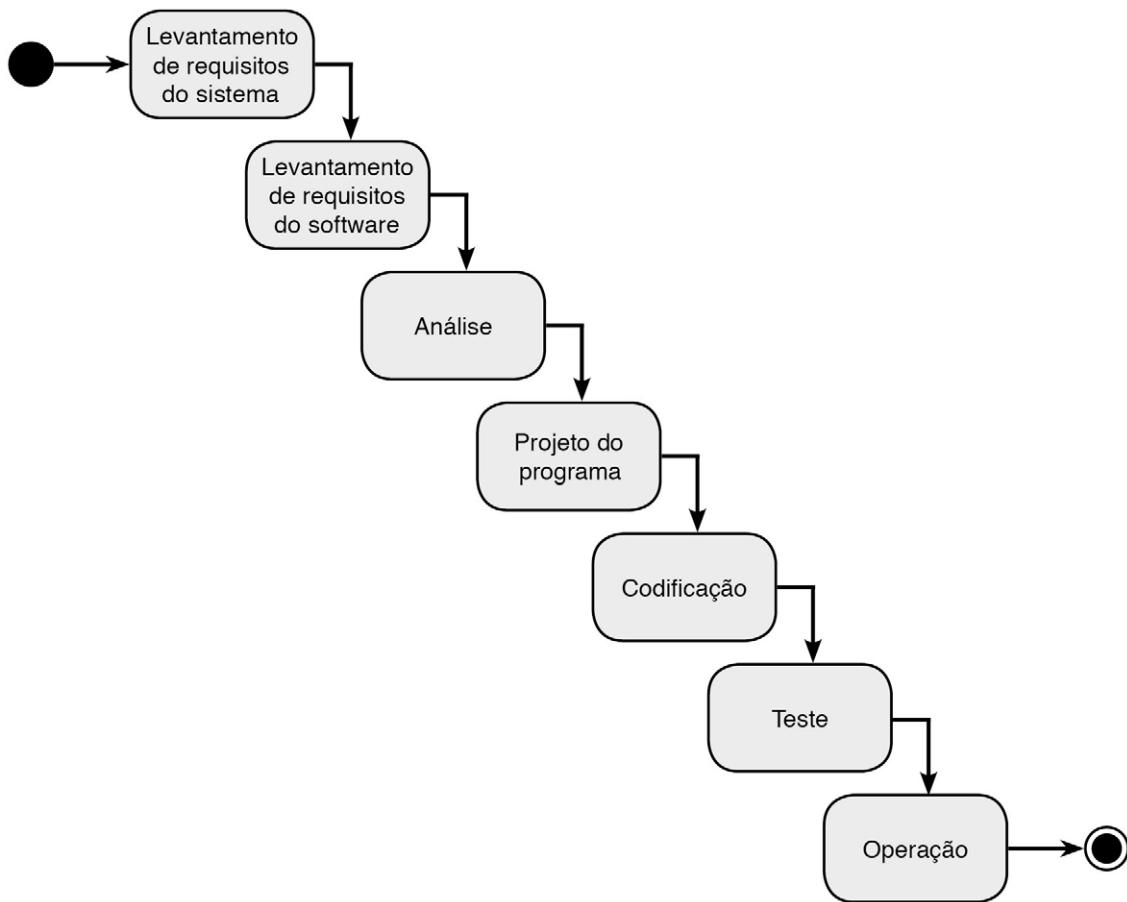


Figura 3.3 Modelo Cascata original (1970).

- c) *Desenvolvedores sempre reclamam que os usuários não sabem expressar aquilo de que precisam:* como os usuários não são especialistas em computação, muitas vezes não mencionam os problemas mais óbvios, que só aparecerão quando o produto estiver em operação.
- d) *Não há flexibilidade com requisitos:* voltar atrás para corrigir requisitos mal estabelecidos ou que mudaram é bastante trabalhoso.

O Modelo Cascata é estritamente sequencial. Sua criação é atribuída a Royce (1970)⁴, que o apresentou pela primeira vez, embora não usasse, na época, a expressão *Waterfall* para designá-lo (Figura 3.3).

O mais irônico nessa questão é que Royce apresentou justamente esse modelo como algo que *não* poderia ser seguido. Ele comenta que, embora acreditasse no modelo como filosofia de projeto organizado, achava sua implementação (conforme mostrado na figura) bastante arriscada, já que apenas na fase de teste vários aspectos do sistema seriam experimentados na prática pela primeira vez. Dessa forma, ele acreditava (e isso se confirma na prática) que, após a fase de testes, muito retrabalho seria necessário para alterar os requisitos e, a partir deles, todo o projeto.

Na sequência de seu artigo, Royce busca apresentar sugestões que diminuam a fragilidade desse modelo. Inicialmente, ele propõe que problemas encontrados em uma fase podem ser resolvidos retornando-se à fase anterior para efetuar as correções. Por exemplo, problemas na codificação poderiam ser resolvidos se o projeto fosse refeito. O modelo resultante, muitas vezes apresentado como ciclo de vida Cascata Dupla, é mostrado na Figura 3.4.

Novamente, Royce apresenta esse modelo como algo que não poderia funcionar bem na prática. Ele diz que, com alguma sorte, as interações entre as fases poderiam ser feitas como na Figura 3.4. Mas, como mostra a Figura 3.5,

⁴Disponível em: <www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf>. Acesso em: 21 jan. 2013.

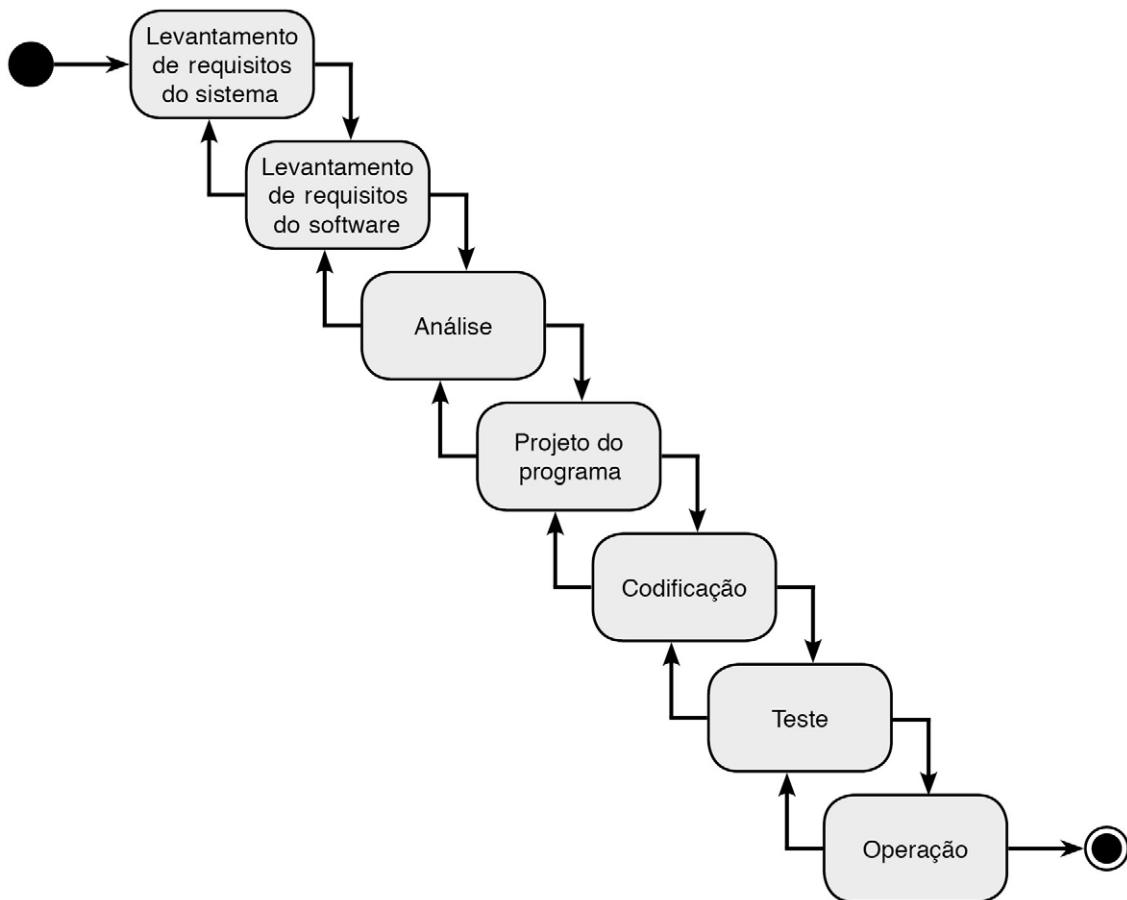


Figura 3.4 Como as interações entre fases poderiam ser (Modelo Cascata Dupla).

não é isso o que acaba acontecendo. De acordo com essa figura, problemas encontrados em uma fase algumas vezes não foram originados na fase imediatamente anterior, mas várias fases antes. Assim, nem o Modelo Cascata Dupla nem o Modelo *Sashimi* (Seção 3.3) apresentam uma solução satisfatória para esse problema.

Então, para contornar o problema de retornar às fases anteriores, Royce apresenta cinco propostas, que visam produzir maior estabilidade dentro das fases do modelo, minimizando a necessidade de retornos:

- Inserir uma fase de design entre o levantamento dos requisitos e sua análise:* se designers que conhecem as limitações dos sistemas computacionais puderem verificar os requisitos antes de os analistas iniciarem seus trabalhos, poderão adicionar preciosos requisitos suplementares referentes às limitações físicas do sistema. Nos modelos modernos, os ciclos de redução de risco permitem realizar essa atividade.
- Produzir documentação:* nas fases iniciais, a documentação é o produto esperado e deve ser feita com a mesma qualidade com que se procura fazer o produto. Caso se trabalhe com a filosofia de gerar código automaticamente, os modelos deverão ser tão precisos quanto o código seria.
- Fazer duas vezes:* sugere-se que o produto efetivamente entregue ao cliente seja a segunda versão produzida. Ou seja, executam-se as fases do Modelo Cascata duas vezes, usando o conhecimento aprendido na primeira rodada para gerar um produto melhor na segunda vez. Essa ideia foi incorporada aos ciclos baseados em prototipação e iterações.
- Planejar, controlar e monitorar o teste:* testes devem ser sistemáticos e realizados por especialistas, já que são essenciais para o sucesso do sistema. Hoje, o Modelo V (Seção 3.4), o Modelo W (Seção 3.5), o Processo Unificado (Capítulo 5) e os modelos ágeis (Capítulo 4) apresentam a disciplina de teste (Capítulo 13) como algo fundamental no projeto do software.
- Envolver o cliente:* é importante envolver o cliente formalmente no processo, e não apenas na aceitação do produto final. Os modelos ágeis, especialmente, consideram o cliente parte da equipe de desenvolvimento.

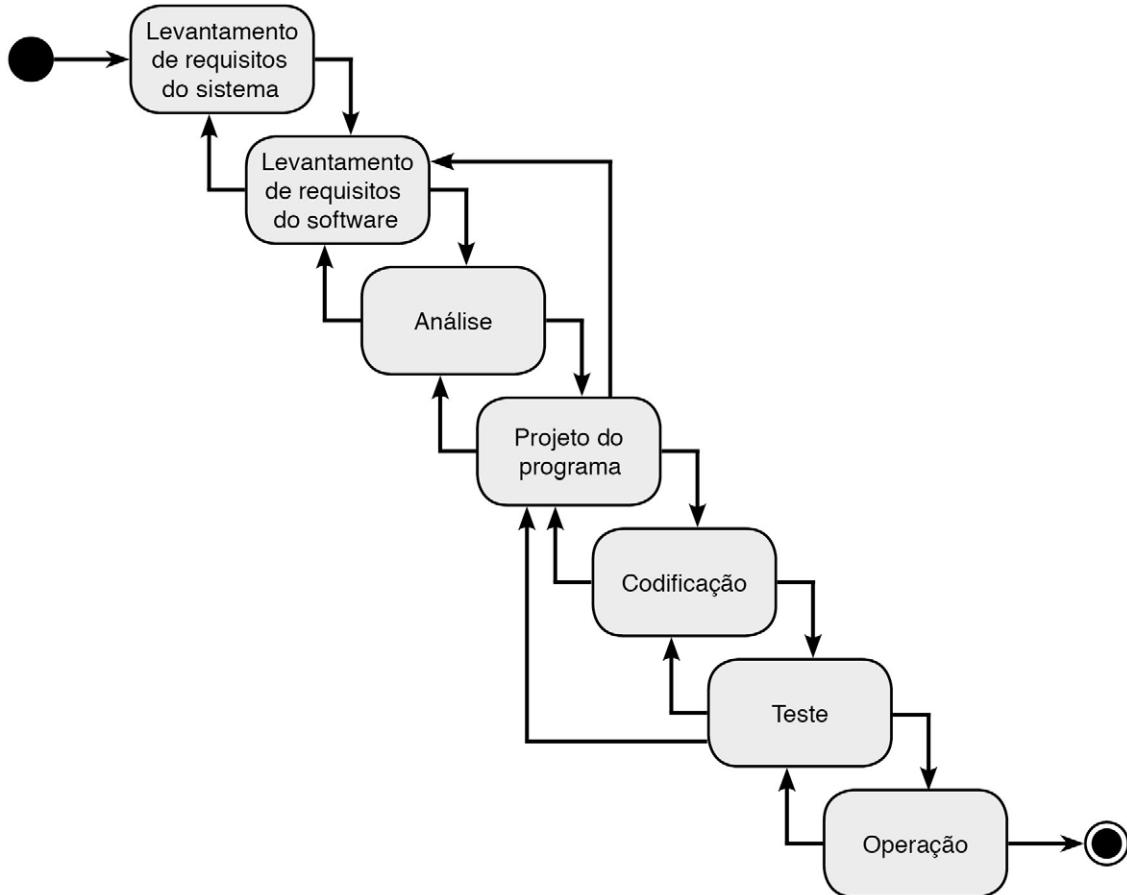


Figura 3.5 Como as interações entre fases do Modelo Cascata acabam acontecendo na prática.

O Modelo Cascata, na sua forma mais simples, é impraticável. Algumas variações foram propostas ao longo do tempo para permitir a aplicação desse tipo de ciclo de vida em processos de desenvolvimento de software reais. Algumas dessas variações são apresentadas nas seções seguintes.

3.3 *Sashimi* (Cascata Entrelaçado)

O modelo conhecido como *Sashimi* (DeGrace & Stahl, 1990), ou *Cascata Entrelaçado* (*Overlapped Waterfall*), é uma tentativa de atenuar a característica BDUF do Modelo Cascata. Em vez de cada fase produzir documentação completa para a fase seguinte, o Modelo *Sashimi* propõe que cada fase continue tratando as questões da fase anterior e procure iniciar o tratamento de questões da fase seguinte.

O diagrama de atividades da UML não é adequado para representar as atividades que se entrelaçam no Modelo *Sashimi* em razão de seu paralelismo difuso. Classicamente, ele tem sido representado como na Figura 3.6, com aparência de comida japonesa, composta por cortes de peixe sobrepostos, de onde vem seu nome.

A ideia do Modelo *Sashimi*, de que cada fase se entrelaça apenas com a anterior e a posterior, entretanto, vai contra a observação de Royce, representada na Figura 3.5. Segundo essa observação, não seria suficiente entrelaçar fases contíguas, pois pode-se necessitar retornar a outras fases anteriores a elas.

Em função disso, uma das evoluções mais importantes do Modelo *Sashimi* é o Modelo *Scrum* (Seção 4.1), que consiste em levar a ideia das fases entrelaçadas ao extremo pela redução do processo a uma única fase, na qual todas as fases do Modelo Cascata seriam realizadas paralelamente por profissionais especializados trabalhando em equipe.

O Modelo *Sashimi*, porém, tem o mérito de indicar que a fase de análise de requisitos só estará completa depois que as questões referentes ao *design* da arquitetura tiverem sido consideradas, e assim por diante.

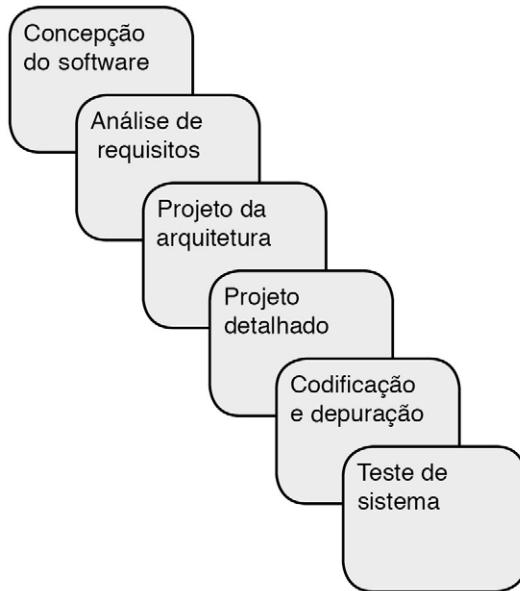


Figura 3.6 Ciclo de vida Sashimi.

Esse estilo de processo é adequado se o engenheiro de software avaliar que poderá obter ganhos de conhecimento sobre o sistema ao passar de uma fase para outra. O modelo também provê uma substancial redução na quantidade de documentação, pois as equipes das diferentes fases trabalharão juntas boa parte do tempo.

Entre os problemas do modelo está o fato de que é mais difícil definir marcos (*milestones*), pois não fica muito claro quando uma fase termina e outra começa. Além disso, a realização de atividades paralelas com esse modelo pode levar a falhas de comunicação, à aceitação de hipóteses erradas e à ineficiência no trabalho.

3.4 Modelo V

O *Modelo V* (*V Model*) é uma variação do Modelo Cascata. Ele prevê uma fase de validação e verificação para cada fase de construção. O Modelo V pode ser usado com projetos que tenham requisitos estáveis e dentro de um domínio conhecido (Lenz & Moeller, 2004)⁵.

O Modelo V é sequencial e, como o Modelo Cascata, é dirigido por documentação. A Figura 3.7 mostra o diagrama de atividades do Modelo V, com as dependências de verificação indicadas entre as atividades.

Na *fase de requisitos*, a equipe e o cliente eliciam e elaboram o documento de requisitos, que é aprovado conjuntamente, na forma de um contrato de desenvolvimento. Uma estimativa de esforço (Capítulo 7) também deve ser produzida nessa fase.

Na *fase de design arquitetural*, a equipe organiza os requisitos em unidades funcionais coesas, definindo como as diferentes partes arquiteturais do sistema vão se interconectar e colaborar. Nessa fase deve ser produzido um documento de especificação funcional do sistema, e as estimativas de esforço podem ser revistas.

Na *fase de design detalhado*, a equipe vai aprofundar a descrição das partes do sistema e tomar decisões sobre como elas serão implementadas. Essa fase produz o documento de especificação detalhado dos componentes do software.

Na *fase de implementação*, o software é implementado de acordo com a especificação detalhada.

A *fase de teste de unidade* tem como objetivo verificar se todas as unidades se comportam como especificado na fase de *design detalhado*. O projeto só segue em frente se todas as unidades passam em todos os testes.

A *fase de teste de integração* tem como objetivo verificar se o sistema se comporta conforme a especificação do *design arquitetural*.

⁵Disponível em: <books.google.com.br/books?id=64XC28ZMhdEC&printsec=frontcover#v=onepage&q=&f=false>. Acesso em: 21 jan. 2013.

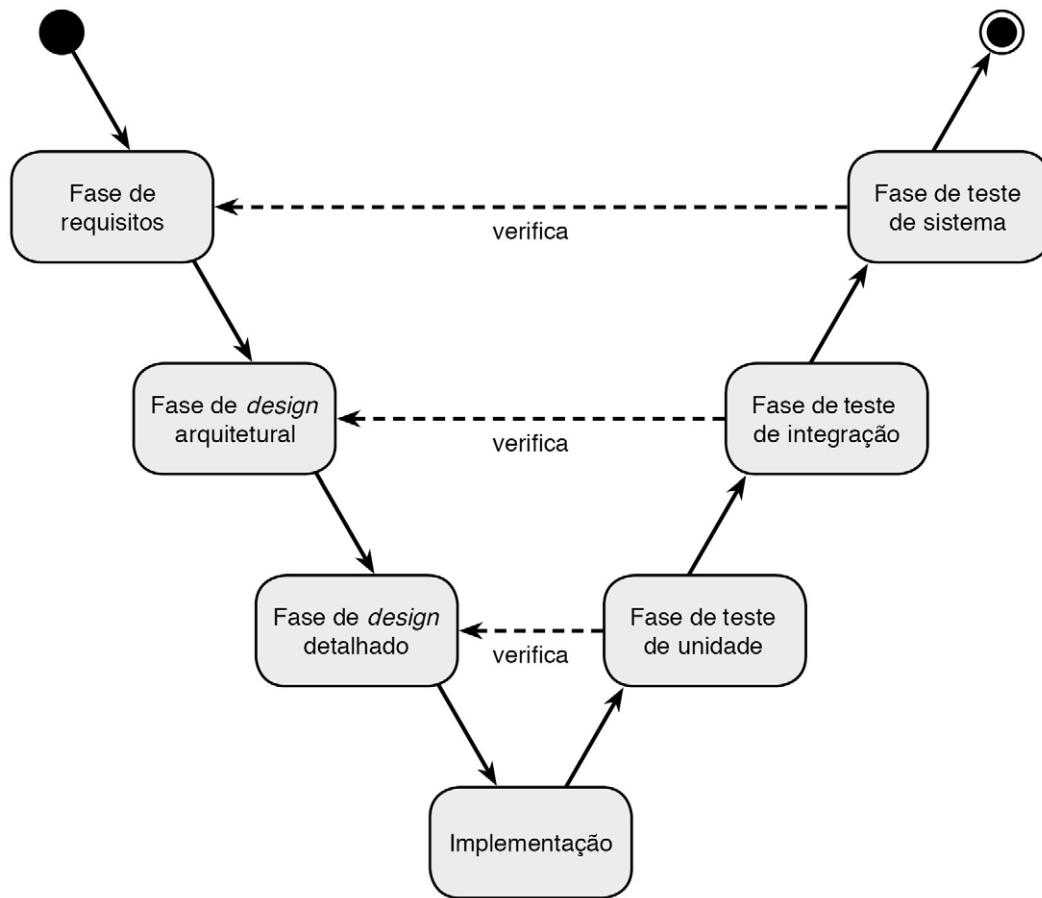


Figura 3.7 Modelo V.

Finalmente, a *fase de teste de sistema* verifica se o sistema satisfaz os requisitos especificados. Se o sistema passar nesses testes, estará pronto para ser entregue ao cliente.

A principal característica do Modelo V está na sua ênfase nos testes e validações simétricos ao *design*. Essas etapas, porém, podem ser incorporadas a outros modelos de processo.

Os pontos negativos desse modelo são os mesmos do Modelo Cascata puro, entre eles o fato de que mudanças nos requisitos geram muito retrabalho. Além disso, em muitos casos, os documentos produzidos no lado esquerdo do V são ambíguos e imprecisos, impedindo ou dificultando os testes necessários representados no lado direito.

3.5 Modelo W

Spillner (2002)⁶ apresenta uma variação ao Modelo V ainda mais voltada à área de testes: o *Modelo W*. A principal motivação para essa variação está na observação de que há uma divisão muito estreita entre as atividades construtivas do lado esquerdo do V e as atividades de teste no lado direito. Spillner propõe que o planejamento dos testes se inicie durante a fase construtiva, mesmo sendo executado depois, e que o lado direito do V não seja considerado apenas um conjunto de atividades de testes, mas também de reconstrução.

A figura resultante desse modelo assemelha-se graficamente a uma letra W (Figura 3.8), em que as atividades exteriores são construtivas e as interiores são atividades de teste.

Uma das questões colocadas já na fase de requisitos diz respeito ao fato de eles serem ou não testáveis. Apenas requisitos que possam ser testados são aceitáveis ao final dessa fase. A mesma questão é colocada em relação à arquitetura na fase de *design* arquitetural. Arquiteturas simples devem ser fáceis de testar, caso contrário, talvez a arquitetura

⁶Disponível em: <www.stickyminds.com/getfile.asp?ot=XML&id=3572&fn=XDD3572filelistfilename1%2Epdf>. Acesso em: 21 jan. 2013.

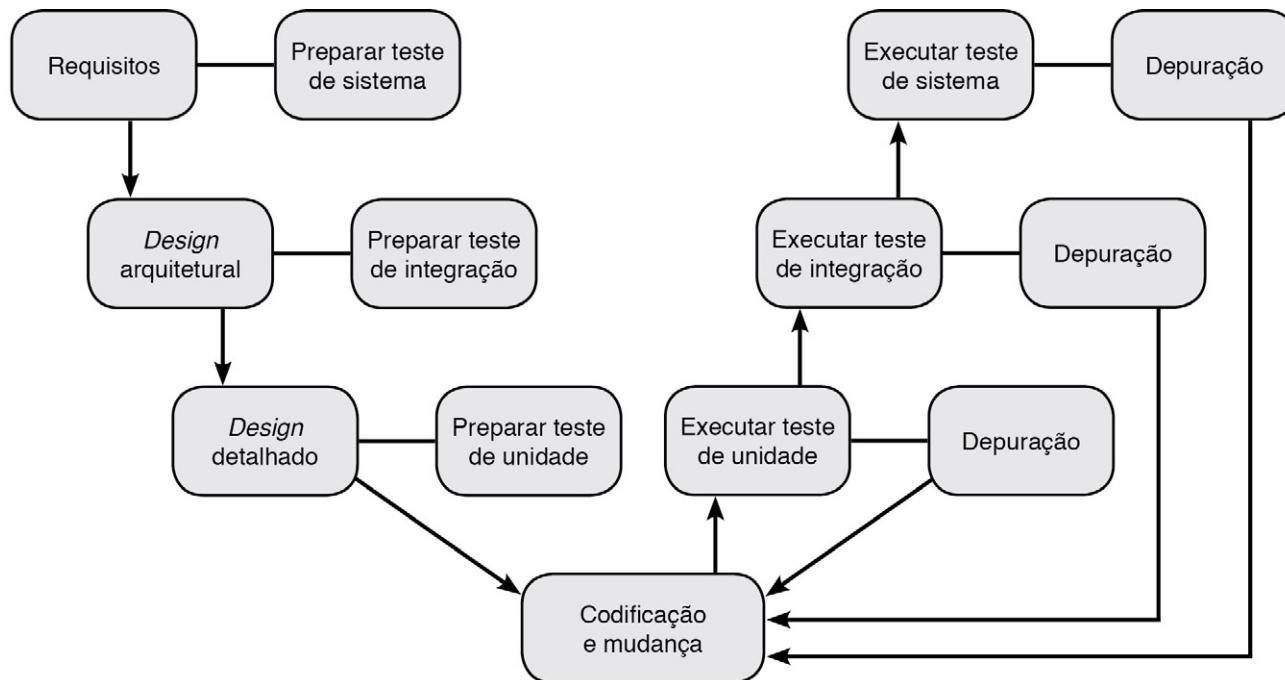


Figura 3.8 Modelo W^{7,8}.

seja demasiadamente complexa e necessite ser refatorada. Na fase de projeto detalhado, a mesma questão se coloca em relação às unidades. Unidades coesas são mais fáceis de testar.

Spillner argumenta que envolver os responsáveis pelos testes já nas fases iniciais de desenvolvimento faz com que mais erros sejam detectados mais cedo e *designs* excessivamente complexos sejam simplificados.

O Modelo W, assim, incorpora o teste nas atividades de desenvolvimento desde seu início, e não apenas nas fases finais. Tal característica também é fortemente utilizada pelos métodos ágeis, que preconizam que o caso de teste deve ser produzido antes do código que será testado.

3.6 Cascata com Subprojetos

O Modelo Cascata com Subprojetos (*Waterfall with Subprojects*) permite que algumas fases do Modelo Cascata sejam executadas em paralelo. Após a fase de *design* da arquitetura, o projeto pode ser subdividido de forma que vários subsistemas sejam desenvolvidos em paralelo por equipes diferentes ou pela mesma equipe em momentos diferentes.

A Figura 3.9 mostra um diagrama de atividades UML que representa o ciclo de vida Cascata com Subprojetos.

Esse modelo é bem mais razoável de se utilizar do que o Modelo Cascata puro, visto que o fato de quebrar o sistema em subsistemas menores permite que subprojetos mais rápidos e fáceis de gerenciar sejam realizados. Essa técnica explora melhor as potencialidades de modularidade do projeto e, com ela, o progresso é visto mais facilmente, porque podem-se produzir várias entregas de partes funcionais do sistema à medida que elas ficam prontas.

A maior dificuldade relacionada a esse modelo está na possibilidade de surgirem interdependências imprevistas entre os subsistemas. O *design* arquitetônico deve ser bem feito, de forma a minimizar tais problemas. Além disso, esse modelo exige maior capacidade de gerência para impedir que sejam criadas inconsistências entre os subsistemas.

Além disso, a integração final de todos os subsistemas pode ser um problema, caso as interdependências não tenham sido adequadamente gerenciadas. Modelos de desenvolvimento ágeis preferem a integração contínua de pequenos pacotes de funcionalidade a grandes fases de integração no final do desenvolvimento.

⁷Disponível em: <www.informatik.hs-bremen.de/spillner/WWW-Talks/Valencia.html>. Acesso em: 21 jan. 2013.

⁸As linhas sem setas ligam atividades que são feitas em paralelo.

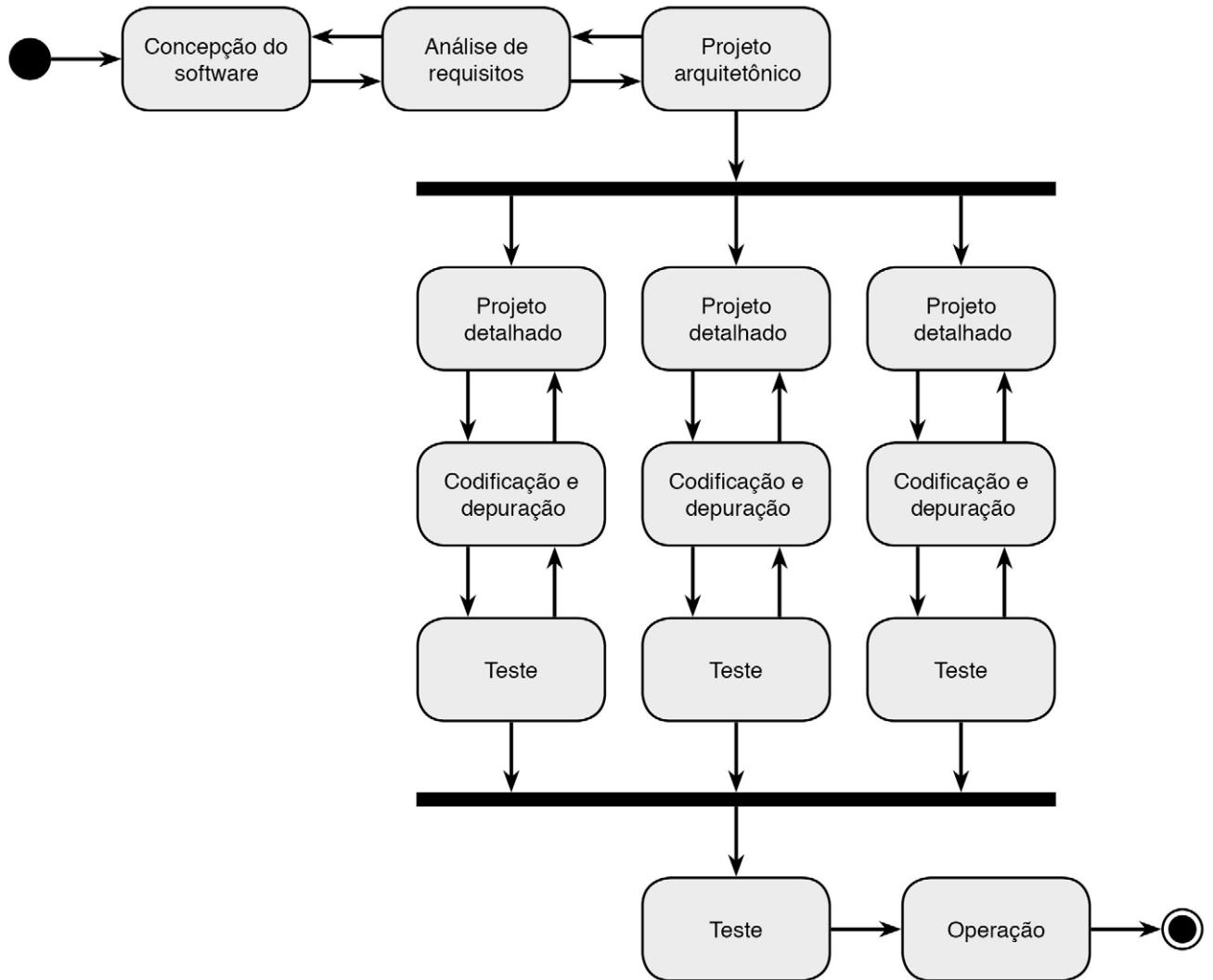


Figura 3.9 Modelo Cascata com Subprojetos.

Apesar disso, esse modelo oficializou uma prática bastante razoável em desenvolvimento de sistemas, que é “dividir para conquistar”, pois é relativamente mais fácil conduzir um processo de desenvolvimento de vários subsistemas parcialmente dependentes do que o de um grande sistema completo. Inclusive, o desenvolvimento de subprojetos sequencialmente pode ser entendido como um ancestral do desenvolvimento iterativo, uma das grandes características dos métodos ágeis e do Processo Unificado.

3.7 Cascata com Redução de Risco

O *Modelo Cascata com Redução de Risco* (*Waterfall with Risk Reduction*) procura resolver um dos principais problemas do *BDUF*, que é a dificuldade de ter uma boa definição dos requisitos do projeto nas fases iniciais. Esse modelo, basicamente, acrescenta uma fase de redução de riscos antes do início do processo em cascata.

O objetivo do modelo é a redução do risco com os requisitos. A tônica é a utilização de técnicas que garantam que os requisitos serão os mais estáveis possíveis. Algumas das técnicas utilizadas durante a fase espiral são:

- Desenvolver protótipos de interface com o usuário*: nesse caso, o modelo por vezes é chamado de “Cascata com Prototipação”. Essa técnica realiza uma das sugestões de Royce: fazer duas vezes. A elaboração de um protótipo antes de comprometer recursos com o desenvolvimento de um sistema real permite que questões relacionadas a requisitos e organização da arquitetura do sistema sejam analisadas e resolvidas.

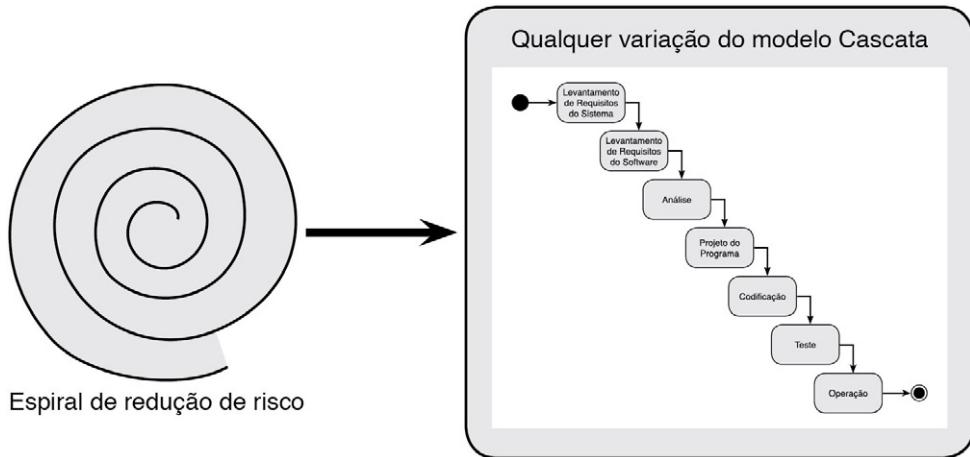


Figura 3.10 Modelo Cascata com Redução de Risco.

- b) *Desenvolver storyboards com o usuário:* a técnica de *storyboard* (Gomes et al., 2007)⁹ utiliza imagens para descrever situações de uso do sistema. É similar à técnica de cenários (Jacobson, 1995), mas em geral os cenários são apenas resultado de dinâmica de grupo e documentados por texto.
- c) *Conduzir vários ciclos de entrevistas com o usuário e o cliente:* em vez de realizar apenas uma entrevista e obter os requisitos a partir dela, a técnica específica que o cliente deve sempre receber um retorno sobre os requisitos levantados a cada entrevista e fazer uma validação destes. Durante o processo de validação, uma nova entrevista poderá esclarecer aspectos confusos ou detalhar aspectos ainda muito gerais.
- d) *Filmar os usuários utilizando os sistemas antigos, sejam eles informatizados ou não:* a análise dos filmes poderá ajudar a compreender os fluxos de trabalho dos usuários. A vantagem dos filmes reais sobre os *storyboards* e os cenários está no fato de que, normalmente, a atuação nos filmes não pode ser supersimplificada nem falsificada pelos usuários. A desvantagem está no fato de que nem sempre surgem com frequência os fluxos de exceção ou variantes de um processo do usuário.
- e) *Utilizar extensivamente quaisquer outras práticas de elicitação de requisitos:* a área de levantamento ou elicitação de requisitos é, por si só, uma área de pesquisa altamente frutífera dentro da Engenharia de Software. Quaisquer técnicas novas ou antigas que possam ajudar o analista a compreender e a modelar adequadamente os requisitos é bem-vinda.

A Figura 3.10 apresenta graficamente o Modelo Cascata com Redução de Risco.

A espiral de redução de risco é uma fase extra que pode ser colocada à frente de qualquer variação do Modelo Cascata: *Sashimi*, Cascata com Subprojetos, Modelo V etc. Ela não precisa ficar restrita aos requisitos do projeto; pode ser aplicada a quaisquer outros riscos identificados (Capítulo 7). Esse tipo de modelo é adequado a projetos com um grande número de riscos significativos. A espiral de redução de risco é uma forma de a equipe garantir alguma estabilidade ao projeto antes de comprometer um grande número de recursos na sua execução.

Como principais desvantagens podem-se citar a dificuldade de definir um cronograma preciso para a fase espiral, bem como as demais desvantagens do Modelo Cascata e suas variantes.

3.8 Modelo Espiral

O *Modelo Espiral* (*Spiral*) foi originalmente proposto por Boehm (1986) e é fortemente orientado à redução de riscos. A proposta de Boehm não foi a primeira a apresentar a ideia de ciclos iterativos, mas foi a primeira a realmente explicar por que as iterações eram necessárias. O projeto é dividido em subprojetos, cada qual abordando um ou mais elementos de alto risco, até que todos os riscos identificados tenham sido tratados.

Pode-se dizer que, de certa forma, o Modelo Espiral também atende a uma das recomendações de Royce (1970), que propunha que o projeto fosse desenvolvido pelo menos duas vezes para que as lições aprendidas na primeira

⁹Disponível em: <www.cin.ufpe.br/~asg/publications/files/WDDS_Final.pdf>. Acesso em: 21 jan. 2013.

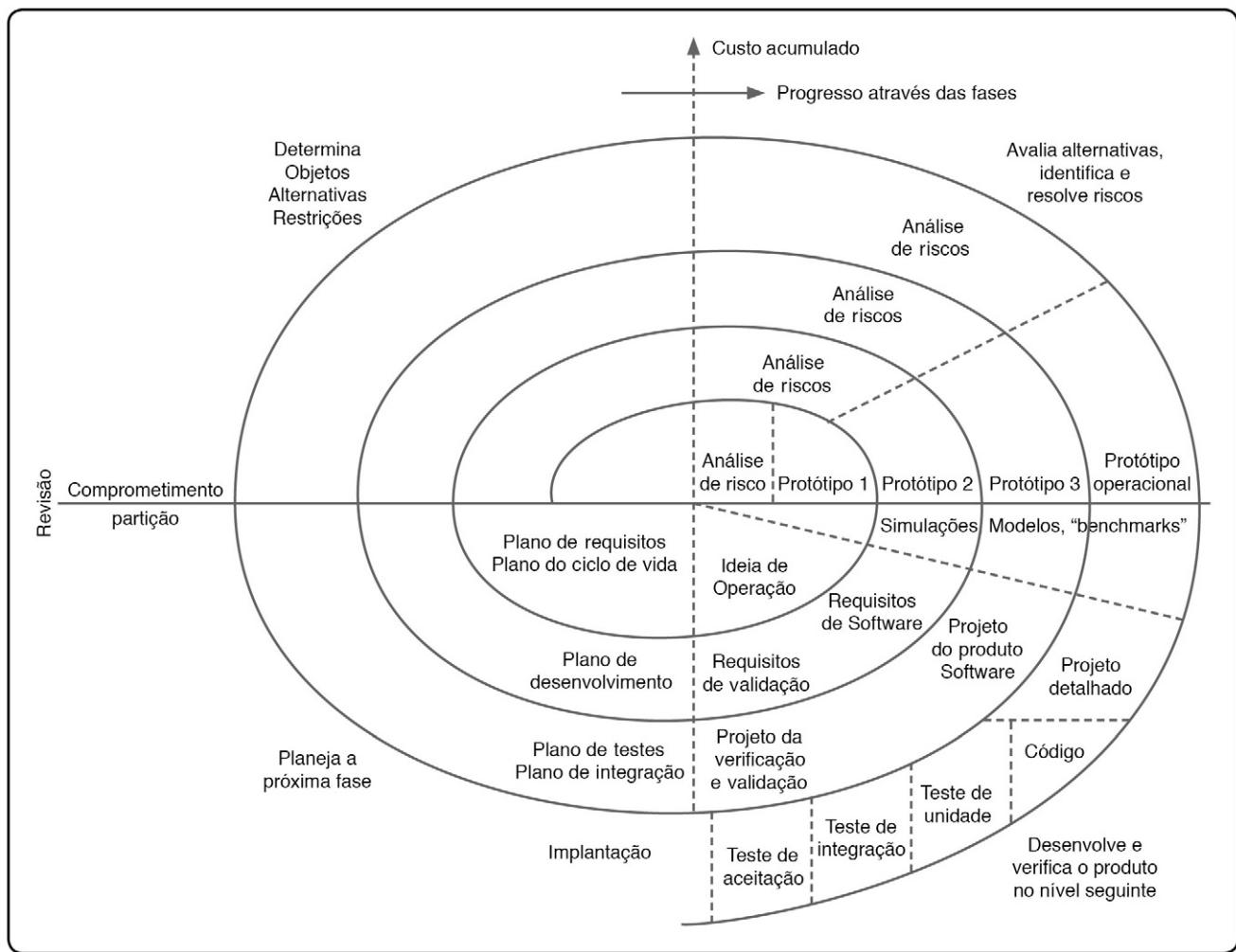


Figura 3.11 Ciclo de vida Espiral¹⁰.

vez pudessem ser aproveitadas na segunda. O Modelo Espiral é uma forma de realizar essas iterações de forma mais organizada, iniciando com pequenos protótipos e avançando para projetos cada vez maiores.

O conceito de risco é definido de maneira abrangente e pode envolver desde requisitos mal compreendidos até problemas tecnológicos, incluindo arquitetura, desempenho, dispositivos eletromecânicos etc. (Capítulo 7).

Depois que os principais riscos foram mitigados, o Modelo Espiral prossegue de forma semelhante ao Modelo Cascata ou uma de suas variantes. A Figura 3.11 apresenta a definição clássica desse ciclo, a partir da qual seu nome foi escolhido.

A ideia desse ciclo é iniciar com miniprojetos, abordando os principais riscos, e então expandir o projeto através da construção de protótipos, testes e replanejamento, de forma a abranger os riscos identificados. Após a equipe ter adquirido um conhecimento mais completo dos potenciais problemas com o sistema, passará a desenvolver um ciclo final semelhante ao do Modelo Cascata.

No início do processo espera-se que a equipe explore os riscos, construa um plano para gerenciar os riscos, planeje e concorde com uma abordagem para o ciclo seguinte. Cada volta no ciclo (ou iteração) faz o projeto avançar um nível em entendimento e mitigação de riscos.

Cada iteração do modelo envolve seis passos:

- Determinar inicialmente os objetivos, alternativas e restrições relacionadas à iteração que vai se iniciar.
- Identificar e resolver riscos relacionados à iteração em andamento.

¹⁰Disponível em: <www2.dem.inpe.br/ijar/CicloVidaSoftPrado.html>. Acesso em: 21 jan. 2013.

- c) Avaliar as alternativas disponíveis. Nessa fase, podem ser utilizados protótipos para verificar a viabilidade de diferentes alternativas.
- d) Desenvolver os artefatos (possivelmente entregas) relacionados a essa iteração e certificar-se de que estão corretos.
- e) Planejar a próxima iteração.
- f) Obter concordância em relação à abordagem para a próxima iteração, caso se resolva realizar uma.

Uma das vantagens do Modelo Espiral é que as primeiras iterações são as mais baratas do ponto de vista de investimento de tempo e recursos e, também, aquelas que resolvem os maiores problemas do projeto. A escolha dos riscos a serem mitigados é feita em função das necessidades de projeto. O método não preconiza este ou aquele risco, então, as atividades concretas nessas fases iniciais podem variar muito de projeto para projeto.

À medida que os custos aumentam, porém, o risco diminui, o que é altamente desejável em projetos de envergadura. Se o projeto não puder ser concluído por razões técnicas, isso será descoberto cedo. Além disso, o modelo possui fases bem definidas, o que permite o acompanhamento objetivo do desenvolvimento.

O modelo não provê a equipe com indicações claras sobre a quantidade de trabalho esperada a cada ciclo, o que pode tornar o tempo de desenvolvimento nas primeiras fases bastante imprevisível. Além disso, o movimento complexo entre as diferentes fases ao longo das várias iterações da espiral exige uma gerência complexa e eficiente.

Esse ciclo de vida é bastante adequado a projetos complexos, com alto risco e requisitos pouco conhecidos, como projetos de pesquisa e desenvolvimento (P&D). Não se recomenda esse ciclo de vida para projetos de pequeno e médio porte ou com requisitos bem conhecidos.

Segundo Schell (2008), o Modelo Espiral é bastante adequado para a área de jogos eletrônicos, nos quais, em geral, os requisitos não são conhecidos sem que protótipos tenham sido testados e os riscos se apresentam altos, tanto do ponto de vista tecnológico quanto do ponto de vista da usabilidade do sistema.

3.9 Prototipação Evolucionária

Em geral, distinguem-se duas abordagens de prototipação:

- a) *throw-away*¹¹, que consiste na construção de protótipos que são usados unicamente para estudar aspectos do sistema, entender melhor seus requisitos e reduzir riscos. O protótipo, depois de cumprir essas finalidades, é descartado (Crinnion, 1991);
- b) *cornerstone*¹², que consiste na construção de protótipos que também são usados para estudar aspectos do sistema, entender melhor seus requisitos e reduzir riscos. O protótipo será parte do sistema final, ou seja, ele vai evoluindo até se tornar um sistema que possa ser entregue (Budde et al., 1992).

O *Modelo de Prototipação Evolucionária* (*Evolutionary Prototyping*, Brooks, 1975) baseia-se na técnica de prototipação *cornerstone*, que exige um planejamento de protótipos muito mais cuidadoso do que a técnica *throw-away*, porque, se não forem consertados, os defeitos nos protótipos iniciais serão propagados ao sistema final.

O Modelo de Prototipação Evolucionária sugere que a equipe de desenvolvimento trabalhe com o cliente os aspectos mais visíveis do sistema, na forma de protótipos (em geral de interface), até que o produto seja aceitável. A Figura 3.12 apresenta o diagrama de atividades UML simplificado para esse modelo.

Esse modelo pode ser particularmente interessante se for difícil fazer o cliente comunicar os requisitos. Nesse caso, um protótipo do sistema será uma ferramenta mais fácil para o analista se comunicar com o cliente e chegar a um acordo sobre o que deve ser desenvolvido.

Esse modelo também pode ser interessante quando nem a equipe nem o cliente conhecem bem os requisitos do sistema. Pode ser difícil elaborar requisitos quando não se sabe exatamente o que é necessário sem ver o software funcionando e sendo testado.

Entretanto, o modelo não é muito bom em relação à previsão de tempo para desenvolvimento ou em relação à gerência do projeto, já que é difícil avaliar quando cada fase foi efetivamente realizada. O projeto que seguir esse modelo pode até regredir para o Modelo Codificar e Consertar. Para evitar isso, deve-se garantir que o processo

¹¹“Descartável”.

¹²“Pedra fundamental”.

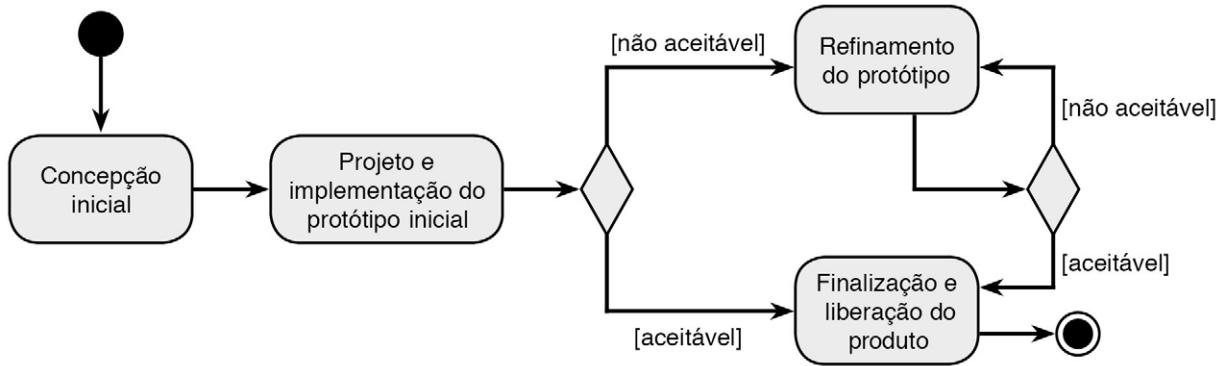


Figura 3.12 Modelo Prototipação Evolucionária.

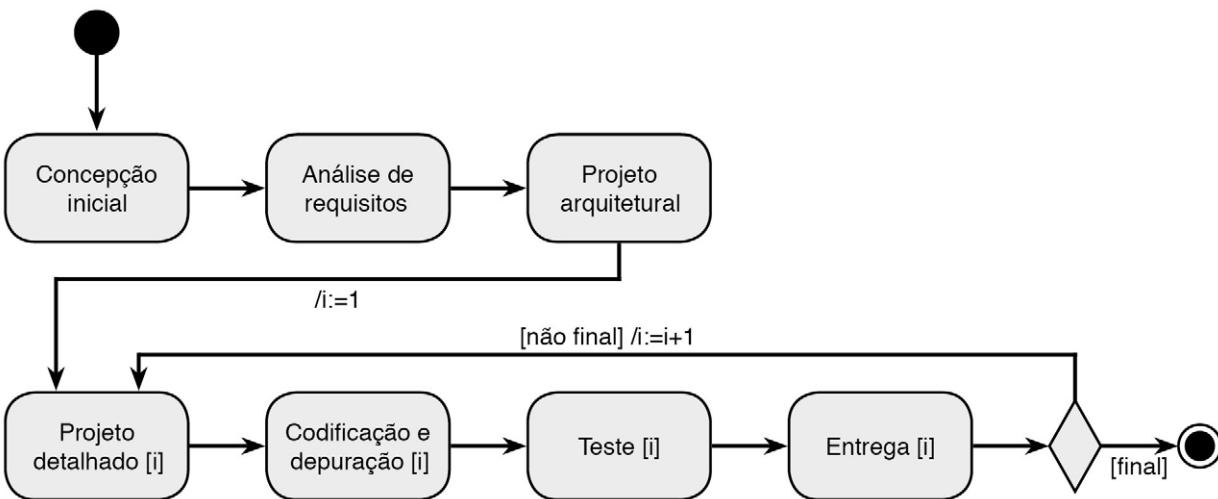


Figura 3.13 Modelo Entregas em Estágios.

efetivamente obtenha uma concepção real do sistema, com requisitos definidos da melhor forma possível e um projeto realista antes de iniciar a codificação propriamente dita.

3.10 Entregas em Estágios

O *Modelo Entregas em Estágios* (*Staged Deliveries* ou *Implementação Incremental*) é uma variação mais bem estruturada do Modelo de Prototipação Evolucionária, embora também seja considerado uma variação do Modelo Cascata. Ao contrário da Prototipação Evolucionária, o Modelo Entregas em Estágios prevê que a cada ciclo a equipe planeje e saiba exatamente o que vai entregar ao cliente.

A abordagem é interessante, porque haverá vários pontos de entrega e o cliente poderá acompanhar mais diretamente a evolução do sistema. Não existe, portanto, o problema do Modelo Cascata, em que o sistema só é entregue quando totalmente acabado.

A Figura 3.13 mostra o diagrama de atividades UML para o modelo. Assim que o *design* arquitetural estiver completo, será possível iniciar a implementação e a entrega de partes do produto final.

Uma das principais vantagens desse modelo (Ellis, 2010) é o fato de colocar funcionalidades úteis nas mãos do cliente antes de completar o projeto. Se os estágios forem planejados cuidadosamente, funcionalidades importantes estarão disponíveis muito mais cedo do que com outros ciclos de vida. Além disso, esse modelo provê entregas mais cedo e de forma contínua, o que pode aliviar um pouco a pressão de cronograma colocada na equipe.

Entretanto, esse modelo não funcionará se as etapas não forem cuidadosamente planejadas nos seguintes níveis:

- a) **Técnico**: as dependências técnicas entre os diferentes módulos entregáveis devem ser cuidadosamente verificadas. Se um módulo tem dependências com outro, o segundo deve ser entregue antes deste.
- b) **Gerencial**: deve-se procurar garantir que os módulos sejam efetivamente significativos para o cliente. Será menos útil entregar funcionalidades parciais que não possam produzir nenhum trabalho consistente.

Para essa técnica funcionar, é necessário que os requisitos sejam bem compreendidos e o planejamento seja efetivo. Ela também pode ser considerada uma precursora dos modelos iterativos, como UP e métodos ágeis.

3.11 Modelo Orientado a Cronograma

O *Modelo Orientado a Cronograma (Design to Schedule)* é similar ao Modelo Entregas em Estágios, exceto pelo fato de que, ao contrário deste último, não se sabe *a priori* quais funcionalidades serão entregues a cada ciclo.

O Modelo Orientado a Cronograma prevê que os ciclos terminarão em determinada data e as funcionalidades implementadas até ali serão entregues. É importante priorizar, portanto, as funcionalidades, de forma que as mais importantes sejam abordadas e entregues primeiro, enquanto as menos importantes ficam para depois. A Figura 3.14 apresenta esse modelo.

Na figura, a cada iteração do ciclo de desenvolvimento, desenvolve-se um conjunto de requisitos, tomando primeiro aqueles que ainda não tiverem sido abordados. Encerra-se o projeto quando o tempo limite for atingido ou quando todos os requisitos tiverem sido atendidos. Espera-se que, mesmo que não tenha sido possível atender a todos os requisitos, pelo menos os que ficaram de fora sejam os menos importantes.

Esse modelo é uma boa estratégia para garantir que haverá algum produto disponível em determinada data, se isso for absolutamente imprescindível, o que faz dele um modelo apropriado para quando existe uma data limite para a entrega, que é intransferível. Porém, se a equipe é altamente confiante na sua capacidade de previsão de esforço (se cumpre prazos constantemente), essa abordagem não é recomendada.

Uma das desvantagens desse modelo é que, caso nem todas as funcionalidades sejam entregues, a equipe terá perdido tempo analisando-as nas etapas iniciais.

Em relação ao Processo Unificado e aos métodos ágeis, esse modelo difere na forma como concebe os ciclos. No Modelo Orientado a Cronograma, a duração dos ciclos não é estabelecida *a priori*. Já nos modelos mais modernos se estabelece uma duração fixa para os ciclos e se tenta implementar um conjunto de funcionalidades dentro dos prazos fixos estabelecidos. Dessa forma, é mais fácil verificar se o projeto está andando bem ou atrasando. Mas, para que isso funcione, é necessário ser capaz de estimar o esforço necessário para desenvolvê-lo (Capítulo 7).

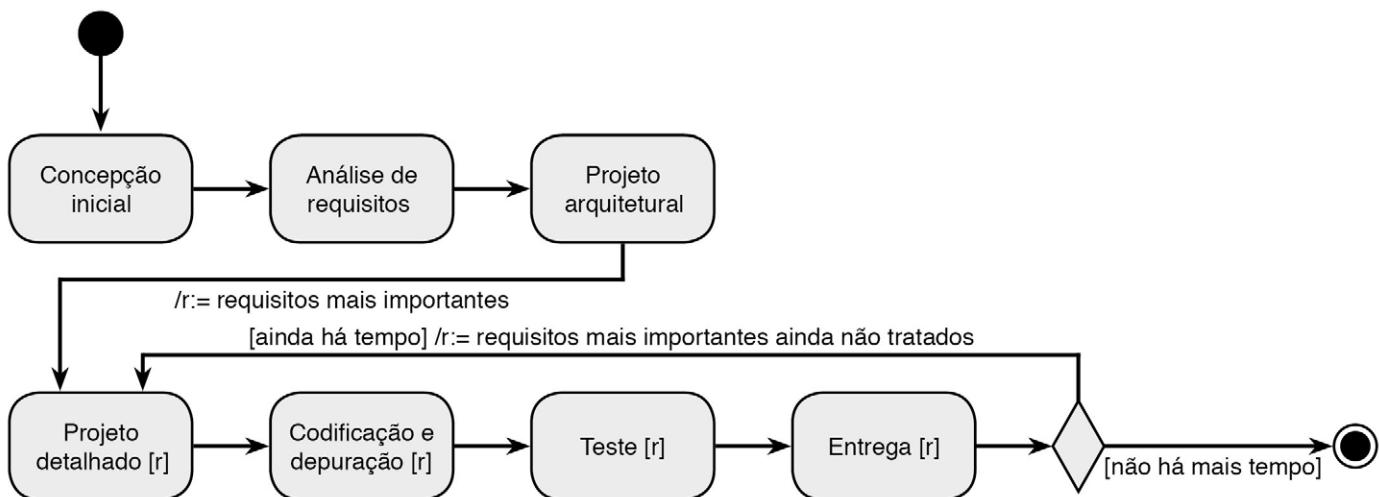


Figura 3.14 Modelo Orientado a Cronograma.

3.12 Entrega Evolucionária

O *Modelo Entrega Evolucionária (Evolutionary Delivery)* é um meio-termo entre a Prototipação Evolucionária (Seção 3.9) e a Entrega em Estágios (Seção 3.10). Nesse modelo, a equipe também desenvolve uma versão do produto, mostra ao cliente e cria novas versões baseadas no *feedback* dado por ele.

O quanto esse modelo se aproxima da Prototipação Evolucionária ou da Entrega em Estágios depende do grau em que se pretende acomodar as modificações solicitadas:

- a) Se a ideia é acomodar todas ou a grande maioria das modificações, então a abordagem tende mais para a Prototipação Evolucionária.
- b) Se, entretanto, as entregas continuarem sendo planejadas de acordo com o previsto e as modificações acomodadas aos poucos nas entregas, então a abordagem se parece mais com a Entrega em Estágios.

Assim, esse modelo permite ajustes que lhe deem um pouco da flexibilidade do Modelo de Prototipação Evolucionária, ao mesmo tempo que se tenha o benefício do planejamento da Entrega em Estágios.

As diferenças entre esse modelo e os anteriores está mais na ênfase do que nas atividades relacionadas. No Modelo de Prototipação Evolucionária, a ênfase está nos aspectos visíveis do sistema. Na Entrega Evolucionária, porém, a ênfase está nas funcionalidades mais críticas do sistema.

De certa maneira, o Processo Unificado implementa essa flexibilidade ao permitir que, ao planejar cada ciclo iterativo, o gerente de projeto escolha se vai abordar um *caso de uso* (equivalente a um conjunto de requisitos), um *risco* ou uma *requisição de modificação*, baseando-se em suas prioridades.

3.13 Modelos Orientados a Ferramentas

Chama-se de *Modelo Orientado a Ferramentas (Design to Tools)* qualquer modelo baseado no uso intensivo de ferramentas de prototipação e geração de código, que permitem a rápida produção de sistemas executáveis a partir de especificações em alto nível, como *jCompany* (Alvim, 2008) ou *WebRatio* (Ceri et al., 2003). É uma abordagem extremamente rápida de desenvolvimento e prototipação, mas é limitada pelas funcionalidades oferecidas pelas ferramentas específicas.

Assim, requisitos só são atendidos se a ferramenta de produção permite atender à funcionalidade requerida. Se as ferramentas forem cuidadosamente escolhidas, entretanto, pode-se conseguir implementar grande parte dos requisitos rapidamente.

Essa abordagem pode ser combinada com outros modelos de processo. A prototipação necessária no Modelo Espiral, por exemplo, pode ser feita com ferramentas de geração de código para produzir protótipos rapidamente.

3.14 Linhas de Produto de Software

Uma Linha de Produto de Software (*Software Product Line – SPL*) consiste, segundo o SEI¹³, de um conjunto de sistemas de software que compartilham características comuns gerenciadas de maneira a satisfazer as necessidades específicas de um segmento de mercado ou missão particular e que foram desenvolvidos a partir de um núcleo comum de forma sistemática.

Segundo Northrop (2008)¹⁴, as SPLs podem ser vistas como uma evolução das estratégias de reutilização na indústria de software. Essas estratégias seriam caracterizadas pela reutilização de *sub-rotinas* nos anos 1960, *módulos* nos anos 1970, *objetos* nos anos 1980, *componentes* nos anos 1990 e *serviços* nos anos 2000. Porém, enquanto as abordagens citadas são meramente técnicas, o reúso obtido com SPL consiste em uma abordagem estratégica para a indústria de software. Dessa forma, o reúso deixa de ser realizado de forma imprevisível e *ad hoc* para ser incorporado sistematicamente aos processos produtivos.

Segundo Weiss e Lay (1999), o uso de uma tecnologia como SPL só se torna financeiramente praticável a partir de certo ponto na escala do número e produtos de uma família. A técnica tem a ver com o desenvolvimento de vários

¹³Disponível em: <www.sei.cmu.edu/productlines/>. Acesso em: 21 jan. 2013.

¹⁴Disponível em: <www.sei.cmu.edu/library/assets/SPL-essentials.pdf>. Acesso em: 21 jan. 2013.

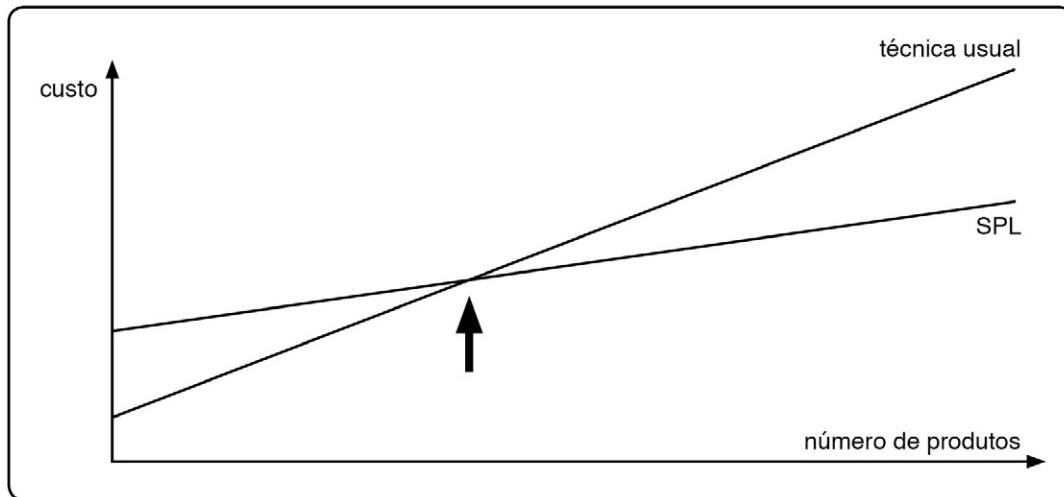


Figura 3.15 Custo/benefício de SPL¹⁵.

produtos diferenciados, mas com características comuns a partir de um núcleo comum. Assim, SPL não se aplica a situações nas quais um único produto é desenvolvido ou vários produtos não relacionados são desenvolvidos.

SPL trata de famílias de produtos como sistemas relacionados a uma mesma área (software para telefones celulares, por exemplo) ou personalizações planejadas. Então, se o número de produtos for pequeno, o investimento para criar e gerenciar uma SPL não se justifica. Mas, para famílias de produtos a partir de certo tamanho, a técnica pode ser a melhor escolha, pois permite o gerenciamento de versões e a otimização do processo produtivo (Figura 3.15).

Segundo Rombach (2005), o investimento em uma SPL começa a compensar a partir do terceiro produto gerado.

A implantação de uma SPL exige mudanças de gerência, forma de desenvolvimento de software, organização estrutural e de pessoal, abordagem de negócio da empresa e, principalmente, na concepção arquitetônica dos produtos. A escolha da arquitetura é fundamental para se obter as funcionalidades corretas com as propriedades desejadas. Uma arquitetura ruim poderá ser causa de fracasso não só em iniciativas de SPL, mas em qualquer projeto de software.

Northrop (2008) destaca, entre todas as novas disciplinas relacionadas a SPL, três atividades essenciais (Figura 3.16):

- a) Desenvolvimento de um núcleo de ativos de produtos (*core asset*).
- b) Desenvolvimento de produtos.
- c) Gerência.

Não há uma ordem predefinida para a execução dessas atividades. Muitas vezes, o produto é produzido a partir do núcleo de ativos; noutras, o núcleo de ativos é gerado a partir de produtos já existentes, ou estes podem ser desenvolvidos em paralelo (Figura 3.16).

Como se vê na figura anterior, não há dependência ou precedência entre as atividades. Elas estão sempre em andamento e mutuamente interligadas.

3.14.1 DESENVOLVIMENTO DO NÚCLEO DE ATIVOS

O objetivo da atividade de *desenvolvimento do núcleo de ativos* é estabelecer um potencial para a produção de produtos de software. O núcleo de ativos é o conjunto de elementos que podem ser reusados na SPL. Esse reúso nem sempre ocorre da mesma forma, por isso talvez precisem ser identificados *pontos de variação*. Pontos de variação são as características dos elementos que podem variar de um reúso para outro. Em geral, consistem num conjunto de restrições às quais suas eventuais instâncias devem se conformar. Além disso, não se apresentam apenas como módulos de software a serem reusados, mas também incluem um conjunto de instruções sobre como proceder ao seu reúso.

A Figura 3.17 apresenta esquematicamente a atividade de desenvolvimento do núcleo de ativos.

¹⁵Weiss e Lai (1999).



Figura 3.16 As três atividades essenciais para SPL¹⁶.

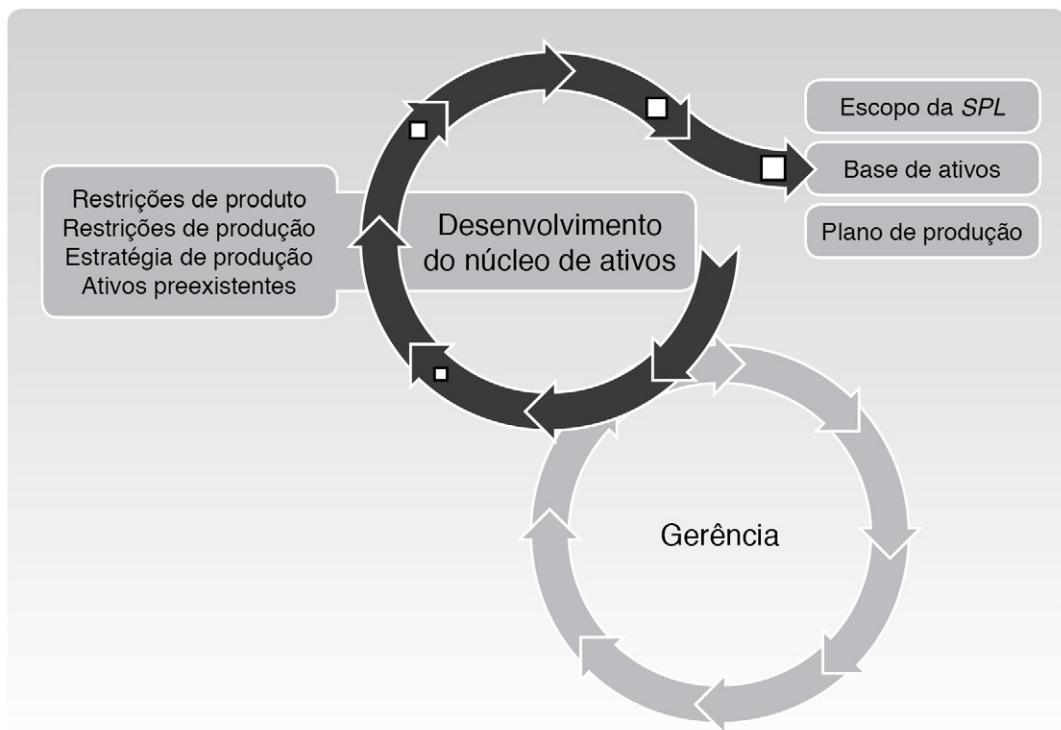


Figura 3.17 A atividade de desenvolvimento do núcleo de ativos¹⁷.

¹⁶Disponível em: <www.sei.cmu.edu/productlines/frame_report/PL.essential.act.htm>. Acesso em: 21 jan. 2013.

¹⁷Disponível em: <www.sei.cmu.edu/productlines/frame_report/coreADA.htm>. Acesso em: 21 jan. 2013.

Essa atividade, assim como as outras duas, é iterativa. Ela ocorre sob a influência de certos fatores ambientais, que são definidos assim¹⁸:

- a) *Restrições de produto*: quais são as coisas em comum e as especificidades dos produtos que constituem a SPL? Quais características de comportamento elas provêm? Quais são as expectativas de mercado e de tecnologia? Quais padrões se aplicam? Quais são suas limitações de performance? Quais limitações físicas (interfaces com sistemas externos, por exemplo) elas devem observar? Quais são os requisitos de qualidade (como segurança e disponibilidade)?
- b) *Restrições de produção*: qual é o prazo esperado para disponibilização de um novo produto? Quais ferramentas de produtividade serão disponibilizadas? Quais padrões de processo de desenvolvimento serão adotados?
- c) *Estratégia de produção*: a SPL será construída de forma proativa, reativa ou como uma combinação das duas? Como será a estratégia de preço? Os componentes-chave serão produzidos ou comprados?
- d) *Ativos preexistentes*: quais ativos da organização podem ser usados na SPL? Existem tanto bibliotecas, frameworks, componentes, web services produzidos internamente quanto obtidos fora?

As saídas da atividade de desenvolvimento de núcleo de ativos podem ser definidas assim:

- a) *Escopo da SPL*: é uma descrição dos produtos que a SPL é capaz de produzir. Essa descrição pode ser uma simples lista ou uma estrutura de similaridades e diferenças. Se o escopo for muito grande, os produtos vão variar demais, haverá poucos pontos em comum e pouca economia no processo produtivo.
- b) *Base de ativos*: é a base para a produção da SPL. Nem todo ativo é necessariamente usado em todos os produtos. Entretanto, todos eles devem ter uma quantidade de usos e reúses que justifiquem seu gerenciamento na base de ativos. Cada ativo deve ter anexado um processo que especifica como ele deve ser usado na produção de novos produtos.
- c) *Plano de produção*: um plano de produção prescreve como os produtos serão produzidos a partir dos ativos. Ele inclui o processo de produção.

3.14.2 DESENVOLVIMENTO DO PRODUTO

O desenvolvimento do produto depende de três insumos: o escopo da SPL, o núcleo de ativos e o plano de produção com a descrição do produto individual (Figura 3.18).

A atividade de desenvolvimento de produto é iterativa e integrada com as outras duas atividades (desenvolvimento do núcleo de ativos e gerência). As entradas para essa atividade são:

- a) A descrição de um produto em particular, frequentemente expressa como um delta ou variação a partir de alguma descrição genérica de produto contida no escopo da SPL.
- b) O escopo da SPL, que indica se é viável incluir um novo produto na linha.
- c) O núcleo de ativos a partir do qual o produto é construído.
- d) O plano de produção, que detalha como o núcleo de ativos pode ser usado para produzir o produto.

Já as saídas previstas para a atividade incluem:

- a) O produto em si, podendo consistir de um ou mais sistemas personalizados.
- b) O *feedback* para o processo produtivo, que permite capitalizar lições aprendidas e melhorar o processo.
- c) Novos ativos, que podem ser gerados ou identificados durante a produção de um produto específico.
- d) Novas restrições de produto, semelhantes às já definidas na Seção 3.14.1.

3.14.3 GERÊNCIA

A atividade de gerência (Capítulo 9), como em qualquer processo de produção de software, desempenha um papel crítico nas SPL. Atividades e recursos devem ser atribuídos e então coordenados e supervisionados. A gerência, tanto em nível de projeto quanto em nível organizacional, deve estar comprometida com a SPL.

¹⁸Disponível em: <www.sei.cmu.edu/productlines/>. Acesso em: 21 jan. 2013.

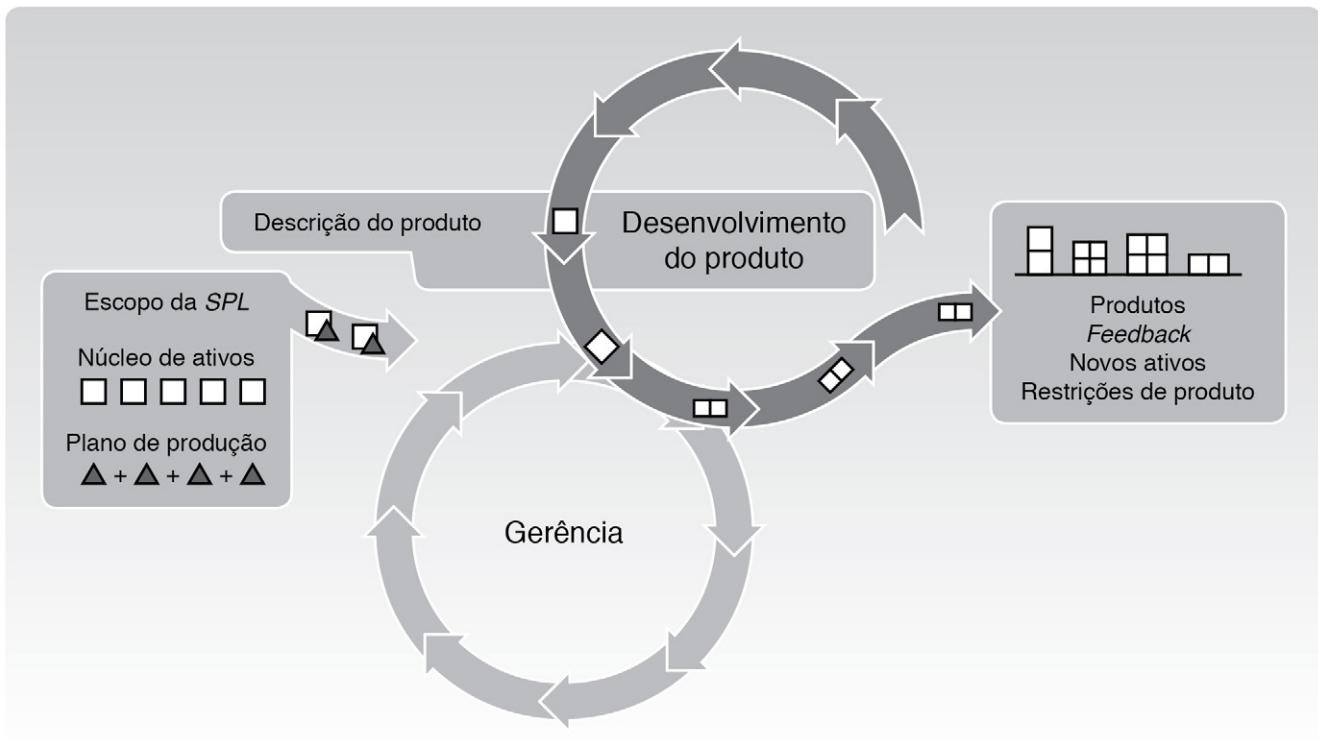


Figura 3.18 Atividade de desenvolvimento do produto em uma SPL¹⁹.

A gerência organizacional identifica a estratégia de negócio e de oportunidades com a SPL. Pode ser considerada a responsável pelo sucesso ou pelo fracasso de um projeto.

Já a gerência técnica deve acompanhar as atividades de desenvolvimento, verificando se os padrões são seguidos, se as atividades são executadas e se o processo pode ser melhorado.

3.14.4 JUNTANDO TUDO

As três disciplinas (desenvolvimento do núcleo e ativos, desenvolvimento do produto e gerência) são organizadas de forma dinâmica, e sua adoção pode se dar de diferentes formas em diferentes empresas. Algumas empresas iniciam pela produção do núcleo de ativos (*abordagem proativa*); outras tomam produtos existentes e identificam as partes em comum para produzir o núcleo (*abordagem reativa*). Northrop (2004)²⁰ apresenta um conjunto de orientações para empresas que desejam adotar SPLs.

As duas abordagens mencionadas podem ser atacadas incrementalmente, isto é, pode-se iniciar com um núcleo de ativos pequeno e ir gradativamente aumentando tanto o núcleo de ativos quanto o número de produtos.

Uma excelente leitura para aprofundar aspectos práticos de SPL está disponível no site do SEI.²¹ Nas descrições das práticas de engenharia, de gerência e organizacionais são destacadas as diferenças fundamentais entre os processos usuais e os processos envolvendo SPLs.

¹⁹Disponível em: <www.sei.cmu.edu/productlines/frame_report/productDA.htm>. Acesso em: 21 jan. 2013.

²⁰Disponível em: <www.sei.cmu.edu/library/abstracts/reports/04tr022.cfm>. Acesso em: 21 jan. 2013.

²¹Disponível em: <www.sei.cmu.edu/productlines/frame_report/productLPAs.htm>. Acesso em: 21 jan. 2013.

Modelos Ágeis

Este capítulo apresenta modelos que têm menos ênfase nas definições de atividades e mais ênfase na pragmática e nos fatores humanos do desenvolvimento. Os métodos ágeis, antigamente conhecidos também como *processos leves*, estão em franco desenvolvimento e várias abordagens podem ser encontradas na literatura. Neste capítulo são apresentados alguns métodos representativos, iniciando com FDD (Seção 4.1), ou *Desenvolvimento Dirigido por Funcionalidade*, que ainda tem certo grau de prescrição e detalhamento de atividades que não é comum a outros métodos considerados ágeis. Outro modelo semelhante é o DSDM (Seção 4.2), ou *Método de Desenvolvimento Dinâmico de Sistemas*. Um método um tanto mais radical em relação a fugir de prescrições e gerenciamento em função de investir na equipe para que ela possa se auto-organizar é o Scrum (Seção 4.3). O modelo XP (Seção 4.4), ou *Programação Extrema*, também é um método pouco baseado em prescrição de atividades, mas, assim como o Scrum, possui regras bem definidas sobre como a equipe deve se organizar, como as iterações e reuniões devem ocorrer e até sobre como o ambiente de trabalho deve ser organizado para que a produtividade e o sucesso pessoal e empresarial sejam maximizados. *Crystal Clear* (Seção 4.5) é o método mais leve da família de métodos de desenvolvimento Crystal, que são personalizados conforme o tamanho e a complexidade do projeto. Finalmente, ASD (Seção 4.6) ou *Desenvolvimento Adaptativo de Sistemas* é um método ágil que aplica ideias de sistemas adaptativos complexos.

Os modelos ágeis de desenvolvimento de software seguem uma filosofia diferente da filosofia dos modelos prescritivos. Em vez de apresentar uma “receita de bolo”, com fases ou tarefas a serem executadas, eles focam valores humanos e sociais.

Apesar de os métodos ágeis serem usualmente mais leves, é errado entendê-los como modelos de processos menos complexos ou simplistas. Não se trata apenas de simplicidade, mas de focar mais nos resultados do que no processo.

Os princípios dos modelos ágeis foram claramente colocados no *Manifesto ágil* (Agile, 2011)¹ e assinados por 17 pesquisadores da área, entre os quais Martin Fowler, Alistair Cockburn e Robert Martin. O manifesto estabelece o seguinte:

Nós estamos descobrindo formas melhores de desenvolver software fazendo e ajudando outros a fazer. Através desse trabalho chegamos aos seguintes valores:

- a) *Indivíduos e interações estão acima de processos e ferramentas.*
- b) *Software funcionando está acima de documentação comprehensível.*

¹Disponível em: <agilemanifesto.org/> . Acesso em: 21 jan. 2013.

- c) Colaboração do cliente está acima de negociação de contrato.
- d) Responder às mudanças está acima de seguir um plano.

Ou seja, enquanto forem valorizados os primeiros, os outros valerão mais.

Isso não significa que os modelos ágeis não valorizem processos, ferramentas, documentação, contratos e planos. Quer dizer apenas que esses elementos terão mais sentido e mais valor depois que indivíduos, interações, software funcionando, colaboração do cliente e resposta às mudanças também forem considerados importantes.

Processos bem estruturados de nada adiantam se as pessoas não os seguem; software bem documentado também não adianta se não satisfaz os requisitos ou não funciona, e assim por diante.

O manifesto ágil é complementado por doze princípios, citados a seguir:

- a) Nossa maior prioridade é satisfazer o cliente através da entrega rápida e contínua de software com valor.
- b) Mudanças nos requisitos são bem-vindas, mesmo nas etapas finais do projeto. Processos ágeis usam a mudança como um diferencial competitivo para o cliente.
- c) Entregar software frequentemente, com intervalos que variam de duas semanas a dois meses, preferindo o intervalo mais curto.
- d) Administradores (*business people*) e desenvolvedores devem trabalhar juntos diariamente durante o desenvolvimento do projeto.
- e) Construa projetos em torno de indivíduos motivados. Dê a eles o ambiente e o suporte e confie que eles farão o trabalho.
- f) O meio mais eficiente e efetivo de tratar a comunicação entre/para a equipe de desenvolvimento é a conversa “cara a cara”.
- g) Software funcionando é medida primordial de progresso.
- h) Processos ágeis promovem desenvolvimento sustentado. Os financiadores, usuários e desenvolvedores devem ser capazes de manter o ritmo indefinidamente.
- i) Atenção contínua à excelência técnica e bom *design* melhoram a agilidade.
- j) Simplicidade – a arte de maximizar a quantidade de um trabalho não feito – é essencial.
- k) As melhores arquiteturas, requisitos e projetos emergem de equipes auto-organizadas.
- l) Em intervalos regulares, a equipe reflete sobre como se tornar mais efetiva e então ajusta seu comportamento de acordo com essa meta.

Um conjunto significativo de modelos atuais é considerado ágil. Alguns são até muito diferentes entre si, mas praticamente todos consideram os princípios mencionados como pontos fundamentais em seu funcionamento. Nas próximas subseções serão apresentados os modelos ágeis mais conhecidos e representativos: FDD, DSDM, *Scrum*, XP, *Crystal Clear* e ASD.

4.1 FDD – Feature-Driven Development

O FDD ou *Feature-Driven Development*² (Desenvolvimento Dirigido por Funcionalidade) é um método ágil que enfatiza o uso de orientação a objetos. Esse modelo foi apresentado, em 1997, por Peter Coad e Jeff de Luca como a evolução de um processo mais antigo (Coad, De Luca & Lefebvre, 1997).

Duas atualizações importantes do modelo foram apresentadas por Palmer e Mac Felsing (2002) e por Anderson (2004).

O FDD possui apenas duas grandes fases:

- a) *Concepção e planejamento*: implica pensar um pouco (em geral de 1 a 2 semanas) antes de começar a construir.
- b) *Construção*: desenvolvimento iterativo do produto em ciclos de 1 a 2 semanas.

²Disponível em: <www.featuredrivendevolution.com/>. Acesso em: 21 jan. 2013.

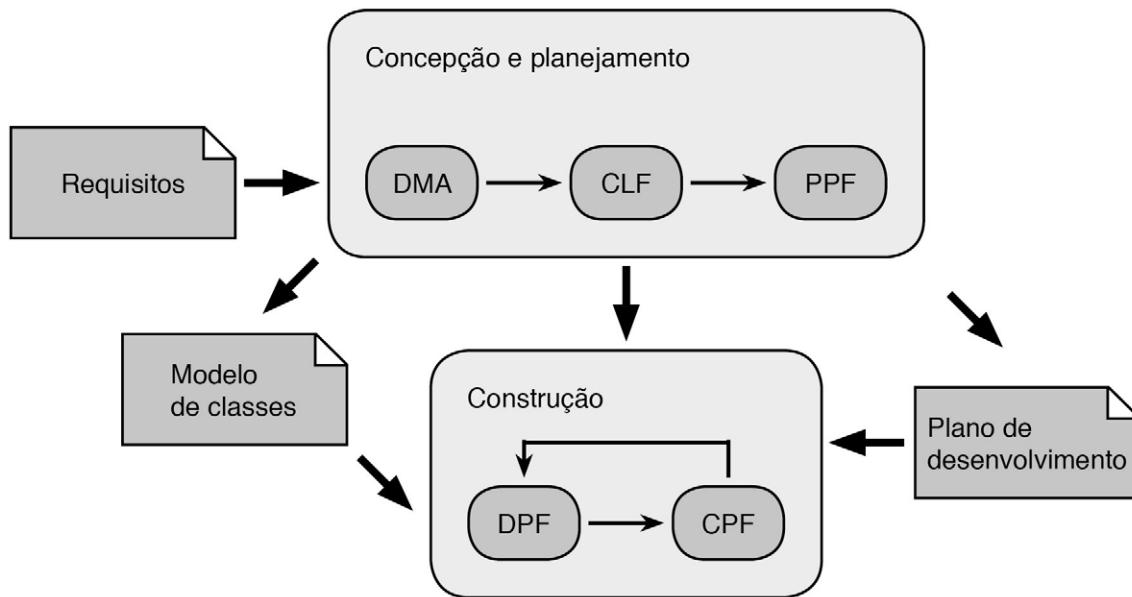


Figura 4.1 Modelo FDD.

A fase de concepção e planejamento possui três disciplinas (chamadas de “processos” em FDD):

- DMA – Desenvolver Modelo Abrangente*, em que se preconiza o uso da modelagem orientada a objetos.
- CLF – Construir Lista de Funcionalidades*, em que se pode aplicar a decomposição funcional para identificar as funcionalidades que o sistema deve disponibilizar.
- PPF – Planejar por Funcionalidade*, em que o planejamento dos ciclos iterativos é feito em função das funcionalidades identificadas.

Já a fase de construção incorpora duas disciplinas (processos):

- DPF – Detalhar por Funcionalidade*, que corresponde a realizar o *design* orientado a objetos do sistema.
- CPF – Construir por Funcionalidade*, que corresponde a construir e testar o software utilizando linguagem e técnica de teste orientadas a objetos.

A estrutura geral do modelo pode ser representada conforme a Figura 4.1.

4.1.1 DMA – DESENVOLVER MODELO ABRANGENTE

A primeira fase se inicia com o desenvolvimento de um modelo de negócio abrangente, seguido do desenvolvimento de modelos mais específicos (de domínio). Estabelecem-se grupos de trabalho formados por desenvolvedores e especialistas de domínio. Os vários modelos específicos assim desenvolvidos são avaliados em um *workshop* e uma combinação deles é formada para ser usada como modelo da aplicação.

Os modelos de negócio e de domínio são representados por diagramas de classes e sua construção deve ser liderada por um modelador com experiência em orientação a objetos. Depois, esse modelo de classes ou modelo conceitual passará a ser refinado na disciplina DPF (Detalhar por Funcionalidade).

Para iniciar o desenvolvimento do modelo abrangente, é necessário que alguns papéis já tenham sido definidos, especialmente os de arquiteto líder, programadores líderes e especialistas de domínio.

As atividades individuais que compõem essa disciplina são as seguintes:

- Formar a equipe de modelagem*: é uma atividade obrigatória sob a responsabilidade do gerente de projeto. As equipes devem ser montadas com especialistas de domínio, clientes e desenvolvedores. Deve haver rodízio entre os membros das equipes, de forma que todos possam ver o processo de modelagem em ação.
- Estudo dirigido sobre o domínio*: é uma atividade obrigatória sob responsabilidade da equipe de modelagem. Um especialista de domínio deve apresentar sua área de domínio para a equipe, especialmente os aspectos conceituais.

- c) *Estudar a documentação*: é uma atividade opcional sob responsabilidade da equipe de modelagem. A equipe estuda a documentação que eventualmente estiver disponível sobre o domínio do problema, inclusive sistemas legados.
- d) *Desenvolver o modelo*: é uma atividade obrigatória sob responsabilidade das equipes de modelagem específicas. Grupos de até três pessoas vão trabalhar para criar modelos candidatos para suas áreas de domínio. O arquiteto líder pode considerar apresentar às equipes um modelo base para facilitar seu trabalho. Ao final, as equipes apresentam seus modelos, que são consolidados em um modelo único.
- e) *Refinar o modelo de objetos abrangente*: é uma atividade obrigatória sob responsabilidade do arquiteto líder e da equipe de modelagem. As decisões tomadas para o desenvolvimento dos modelos específicos de domínio poderá afetar a forma do modelo geral do negócio, que deve então ser refinado.
O resultado dessas atividades deve ser verificado pela própria equipe de modelagem (verificação interna) e também pelo cliente ou pelos especialistas de domínio (verificação externa).
As saídas esperadas desse conjunto de atividades são:
- f) *O modelo conceitual*, apresentado como um diagrama de classes e suas associações.
- g) *Métodos e atributos*, eventualmente identificados para as classes.
- h) *Diagramas de sequência ou máquina de estados* para as situações que exigirem esse tipo de descrição.
- i) *Comentários* sobre o modelo para indicar por que determinadas decisões de *design* foram tomadas em vez de outras.

As técnicas para construir esses diagramas de classes, sequência e máquina de estados podem ser encontradas nos Capítulos 2 e 7 de Wazlawick (2011).

4.1.2 CLF – CONSTRUIR LISTA DE FUNCIONALIDADES

Esta disciplina vai identificar as funcionalidades que satisfazem aos requisitos. A equipe vai decompor o domínio em áreas de negócio, conforme a disciplina DMA. As áreas de negócio serão decompostas em atividades de negócio e estas, por sua vez, em passos de negócio (Figura 4.2). O processo é relativamente parecido com uma análise de casos de uso, em que haveria áreas de negócio (*pacotes*), *casos de uso* como atividades de negócio e *passos* de casos de uso expandidos como passos de negócio.

Para iniciar as atividades é necessário que os especialistas de domínio, programadores líderes e arquiteto líder estejam disponíveis.

A disciplina é composta por uma única atividade: *construir a lista de funcionalidades*, a qual é obrigatória e de responsabilidade da equipe da lista de funcionalidades (formada pelos programadores líderes da disciplina anterior).

Assim, a equipe vai listar as funcionalidades em três níveis:

- a) *Áreas de negócio*, oriundas das atividades de DMA. Por exemplo, “vendas”.
- b) *Atividades de negócio*, que são a decomposição funcional das áreas de negócio. Por exemplo, registrar pedido, faturar pedido, registrar pagamento de venda etc.
- c) *Passos de atividades de negócio*, que são a descrição sequencial das funcionalidades necessárias para realizar as atividades de negócio. Por exemplo, identificar cliente para pedido, registrar produto e quantidade do pedido, aplicar desconto padrão ao pedido etc.

As funcionalidades *não* devem ser ações realizadas sobre tecnologia (como “abrir janela” ou “acessar menu”), mas ações que tenham significado para o cliente, independentemente de tecnologia. Boas funcionalidades devem poder ser nomeadas como uma instância da tríade <ação resultado objeto>. Por exemplo, “apresentar total das vendas no mês” ou “Registrar concordância do contratado”.

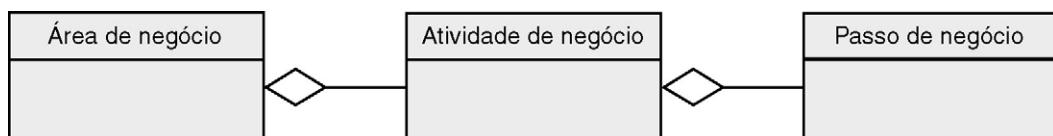


Figura 4.2 Estrutura conceitual da lista de funcionalidades.

Espera-se que o tempo para implementação de uma funcionalidade nunca seja superior a duas semanas, sendo esse um limite absoluto, já que o tempo esperado seria de poucos dias. Funcionalidades com esforço estimado de menos de um dia não precisam ser consideradas na lista de funcionalidades, mas possivelmente serão parte de outras mais abrangentes. De outro lado, quando uma funcionalidade parece levar mais de duas semanas para ser desenvolvida (por uma pessoa), ela deve ser quebrada em funcionalidades menores.

A avaliação dessa atividade também pode ser feita de forma interna pela equipe de lista de funcionalidades ou, externamente, pelos usuários, clientes e especialistas de domínio.

As saídas dessa atividade são as seguintes:

- a) Lista de áreas de negócio.
- b) Para cada área, uma lista de atividades de negócio dentro da área.
- c) Para cada atividade, uma lista de passos de atividade ou funcionalidades que permitem realizar a atividade.

4.1.3 PPF – PLANEJAR POR FUNCIONALIDADE

Esta disciplina, ainda na primeira fase do FDD, visa gerar o plano de desenvolvimento para a fase seguinte, que é formada por ciclos iterativos. O planejamento vai indicar quais atividades de negócio da lista definida em CLF serão implementadas e quando. O planejador deve levar em consideração os seguintes aspectos para agrupar e priorizar as atividades de negócio:

- a) *Complexidade das funcionalidades*: em função do risco, as atividades com funcionalidades de maior complexidade devem ser tratadas primeiro.
- b) *Dependências entre as funcionalidades em termos de classes*: funcionalidades dependentes devem preferencialmente ser abordadas juntas, pois isso evita a fragmentação do trabalho nas classes entre as diferentes equipes.
- c) *Carga de trabalho da equipe*: deve ser usado um método de estimativa (Capítulo 7) para que seja conhecido o esforço para a implementação de cada atividade ou funcionalidade, e o trabalho deve ser atribuído à equipe em função de sua capacidade e dessa estimativa.

Ao contrário de outros métodos ágeis, o FDD propõe que a posse das classes seja atribuída aos programadores líderes, ou seja, cada um se responsabiliza por um subconjunto das classes. Então, ao fazer a divisão da carga de trabalho, em geral se estará definindo também a posse das classes.

A entrada para as atividades dessa disciplina consiste na *lista de funcionalidades* construída em CLF.

São quatro as atividades da disciplina PPF:

- a) *Formar a equipe de planejamento*: é uma atividade obrigatória de responsabilidade do gerente do projeto. Essa equipe deve ser formada pelo gerente de desenvolvimento e pelos programadores líderes.
- b) *Determinar a sequência de desenvolvimento*: é uma atividade obrigatória de responsabilidade da equipe de planejamento. A equipe deve determinar o prazo de conclusão do desenvolvimento de cada uma das atividades de negócio. Esse prazo deve ser determinado em função de mês e ano, ou seja, de prazos mensais. A sequência de desenvolvimento deve ser construída levando-se em consideração estes fatores:
 - Priorizar as atividades com funcionalidades mais complexas ou de alto risco.
 - Alocar juntas atividades ou funcionalidades dependentes umas das outras, se possível.
 - Considerar marcos externos, quando for o caso, para a criação de *releases*.
 - Considerar a distribuição de trabalho entre os proprietários das classes.
- c) *Atribuir atividades de negócio aos programadores líderes*: é uma atividade obrigatória de responsabilidade da equipe de planejamento e vai determinar quais programadores líderes serão proprietários de quais atividades de negócio. Essa atribuição de propriedade deve ser feita considerando-se os seguintes critérios:
 - Dependência entre as funcionalidades e as classes das quais os programadores líderes já são proprietários.
 - A sequência de desenvolvimento.
 - A complexidade das funcionalidades a serem implementadas em função da carga de trabalho alocada aos programadores líderes.

d) Atribuir classes aos desenvolvedores: é uma atividade obrigatória de responsabilidade da equipe de planejamento, que deve atribuir a propriedade das classes aos desenvolvedores. Essa atribuição é baseada em:

- distribuição de carga de trabalho entre os desenvolvedores.
- complexidade das classes (priorizar as mais complexas ou de maior risco).
- intensidade de uso das classes (priorizar as classes altamente usadas).
- sequência de desenvolvimento.

A verificação das atividades é feita unicamente de forma interna. A própria equipe de planejamento faz uma autoavaliação para verificar se as atividades foram desenvolvidas adequadamente.

O resultado ou artefato de saída das atividades é o *plano de desenvolvimento*, que consiste em:

- a) Prazos** (mês e ano) para a conclusão do desenvolvimento referente a cada uma das atividades de negócio.
- b) Atribuição** de programadores líderes a cada uma das atividades de negócio.
- c) Prazos** (mês e ano) para a conclusão do desenvolvimento referente a cada uma das áreas de negócio (isso é derivado da última data de conclusão das atividades de negócio incluídas na respectiva data).
- d) Lista** dos desenvolvedores e das classes das quais eles são proprietários.

4.1.4 DPF – DETALHAR POR FUNCIONALIDADE

A disciplina DPF (Detalhar por Funcionalidade) é a primeira executada na fase iterativa do FDD. Ela consiste basicamente em produzir o *design* de implementação da funcionalidade, que costuma ser realizado por diagramas de sequência ou comunicação (Wazlawick, 2011).

A atividade de detalhamento é realizada para cada funcionalidade identificada dentro das atividades de negócio. A atribuição do trabalho aos desenvolvedores é feita pelo programador líder, que poderá considerar, para tanto, a conexão entre as funcionalidades ou ainda a posse (propriedade) das classes envolvidas.

Como entrada, as atividades dessa disciplina necessitam que o planejamento da etapa anterior tenha sido concluído.

As atividades de DPF são as seguintes:

- a) Formar a equipe de funcionalidades:** é uma atividade obrigatória de responsabilidade do programador líder. O programador líder cria um pacote de funcionalidades a serem trabalhadas e, em função das classes envolvidas com essas funcionalidades, define a equipe de funcionalidades com os proprietários dessas classes. O programador líder também deve atualizar o controle de andamento de projeto, indicando que esse pacote de funcionalidades está sendo trabalhado.
- b) Estudo dirigido de domínio:** é uma atividade opcional de responsabilidade do especialista de domínio. Se a funcionalidade for muito complexa, o especialista de domínio deve apresentar um estudo dirigido sobre a funcionalidade no domínio em que ela se encaixa. Por exemplo, uma funcionalidade como “calcular impostos” pode ser bastante complicada e merecerá uma apresentação detalhada por um especialista de domínio antes que os desenvolvedores começem a projetá-la.
- c) Estudar a documentação de referência:** é uma atividade opcional de responsabilidade da equipe de funcionalidades. Dependendo da complexidade da funcionalidade, pode ser necessário que a equipe estude documentos disponíveis, como relatórios, desenhos de telas, padrões de interface com sistemas externos etc.
- d) Desenvolver os diagramas de sequência:** é uma atividade opcional de responsabilidade da equipe de funcionalidades. Os diagramas necessários para descrever a funcionalidade podem ser desenvolvidos. Assim como outros artefatos, devem ser submetidos a um sistema de controle de versão (Seção 10.2). Decisões de *design* devem ser anotadas (por exemplo, uso do padrão *stateless* ou *statefull* etc.).
- e) Refinar o modelo de objetos:** é uma atividade obrigatória de responsabilidade do programador líder. Com o uso intensivo do sistema de controle de versões, o programador líder cria uma área de trabalho a partir de uma cópia das classes necessárias do modelo, disponibilizando essa cópia para a equipe de funcionalidades. Esta, por sua vez, terá acesso compartilhado à cópia, mas o restante do pessoal não, até que ela seja salva como uma nova versão das classes no sistema de controle de versões. Então, a

equipe de funcionalidades deverá adicionar métodos, atributos, associações e novas classes que forem necessárias.

- f) *Escrever as interfaces (assinatura) das classes e métodos:* é uma atividade obrigatória sob responsabilidade da equipe de funcionalidades. Utilizando a linguagem de programação alvo da aplicação, os proprietários das classes escrevem as interfaces das classes, incluindo os atributos e seus tipos, além da declaração dos métodos, incluindo tipos de parâmetros e retornos, exceções e mensagens.

A verificação do produto dessas atividades é feita por avaliação interna de inspeção do *design*. Essa avaliação deve ser feita pela própria equipe de funcionalidades, mas outros membros do projeto podem participar dela. Após o aceite do produto final, uma lista de atividades é gerada para cada desenvolvedor (lembrando que, até aqui, apenas assinaturas dos métodos foram definidas e eles ainda precisam ser efetivamente implementados), e a nova versão das classes é gravada (*commit*) no sistema de controle de versões.

A saída das atividades é um *pacote de design* inspecionado e aprovado. Esse pacote consiste de:

- a) Uma capa com comentários que descreve o pacote de forma suficientemente clara.
- b) Os requisitos abordados na forma de atividades e/ou funcionalidades.
- c) Os diagramas de sequência.
- d) Os projetos alternativos (se houver).
- e) O modelo de classes atualizado.
- f) As interfaces de classes geradas.
- g) A lista de tarefas para os desenvolvedores, gerada em função dessas atividades.

4.1.5 CPF – CONSTRUIR POR FUNCIONALIDADE

Esta disciplina, também executada dentro da fase iterativa do FDD, tem como objetivo a produção do código para as funcionalidades identificadas. A partir do *design* gerado pela disciplina anterior e de acordo com o cronograma definido pelo programador líder, os desenvolvedores devem construir e testar o código necessário para cada classe atribuída. Após a inspeção do código pelo programador líder, ele é salvo no sistema de controle de versão e passa a ser considerado um *build*.

A entrada para essas atividades é o *pacote de design* gerado na disciplina anterior.

As atividades envolvidas são:

- a) *Implementar classes e métodos:* é uma atividade obrigatória sob responsabilidade da equipe de funcionalidades. A atividade é realizada pelos proprietários de classes em colaboração uns com os outros.
- b) *Inspecionar o código:* é uma atividade obrigatória sob responsabilidade da equipe de funcionalidades. Uma inspeção do código pode ser feita pela própria equipe ou por analistas externos (Seção 11.4.2), mas é sempre coordenada pelo programador líder e pode ser feita antes ou depois dos testes de unidade.
- c) *Teste de unidade:* é uma atividade obrigatória sob responsabilidade da equipe de funcionalidades (Seção 13.2.1). Os proprietários de classes definem e executam os testes de unidade de suas classes para procurar eventuais defeitos ou inadequação a requisitos. O programador líder poderá determinar testes de integração entre as diferentes classes quando julgar necessário.
- d) *Promover à versão atual (build):* é uma atividade obrigatória sob responsabilidade do programador líder. À medida que os desenvolvedores vão reportando sucesso nos testes de unidade, o programador líder poderá promover a versão atual de cada classe individualmente à *build* (Capítulo 10), não sem antes passar pelos testes de integração.

A verificação do produto de trabalho é feita internamente pelo programador líder e pela equipe de funcionalidades.

Os resultados ou saídas dessas atividades são:

- a) Classes que passaram com sucesso em testes de unidade e integração e foram, por isso, promovidas à versão atual (*build*).
- b) Disponibilização de um conjunto de funcionalidades com valor para o cliente.

4.1.6 FERRAMENTAS PARA FDD

Existem ferramentas disponíveis específicas para uso com o método FDD, entre elas:

- a) *CaseSpec*³: ferramenta proprietária, mas com *free trial*, para gerenciamento de requisitos ao longo de todo o ciclo de vida.
- b) *TexExcel DevSuite*⁴: conjunto de ferramentas específicas para aplicar FDD.
- c) *FDD Tools Project*⁵: projeto que visa criar ferramentas gratuitas de código aberto para FDD.

4.2 DSDM – Dynamic Systems Development Method

Dynamic Systems Development Method (DSDM)⁶ também é um modelo ágil baseado em desenvolvimento iterativo e incremental, com participação ativa do usuário. O método é uma evolução do *Rapid Application Development* (RAD) (Martin, 1990), que, por sua vez, é um sucessor de *Prototipação Rápida* (Seção 3.9).

O DSDM possui três fases:

- a) *Pré-projeto*: nesta fase, o projeto é identificado e negociado, seu orçamento é definido e um contrato é assinado.
- b) *Ciclo de vida*: o ciclo de vida se inicia com uma fase de análise de viabilidade e de negócio. Depois entra em ciclos iterativos de desenvolvimento.
- c) *Pós-projeto*: equivale ao período normalmente considerado como operação ou manutenção. Nessa fase, a evolução do software é vista como uma continuação do processo de desenvolvimento, podendo, inclusive, retomar fases anteriores, se necessário.

A fase de ciclo de vida, por sua vez, subdivide-se nos seguintes estágios:

- a) Análise de viabilidade.
- b) Análise de negócio (por vezes aparece junto com a anterior).
- c) Iteração do modelo funcional.
- d) Iteração de elaboração e construção.
- e) Implantação.

DSDM fundamenta-se no *Princípio de Pareto*,⁷ ou Princípio 80/20. Assim, o DSDM procura iniciar pelo estudo e implementação dos 20% dos requisitos que serão mais determinantes para o sistema como um todo. Essa prática é compatível com a prática de abordar inicialmente requisitos mais complexos ou de mais alto risco, presentes em outros modelos, como UP.

Outro aspecto do DSDM é a *objetividade*, que afirma que a gerência de riscos deve se focar nas funcionalidades a serem entregues em detrimento de outros fatores, como documentos ou o processo em si.

A filosofia DSDM pode ser resumida pelos seguintes princípios:

- a) *Envolvimento* do usuário no projeto o tempo todo.
- b) *Autonomia* dos desenvolvedores para tomar decisões sem a necessidade de consultar comitês superiores.
- c) *Entregas frequentes* de releases suficientemente boas são o melhor caminho para conseguir entregar um bom produto no final. Postergar entregas aumenta o risco de o produto final não ser o que o cliente deseja.
- d) *Eficácia* das entregas na solução de problemas de negócio. Como as primeiras entregas vão se concentrar nos requisitos mais importantes, elas serão mais eficazes para o negócio.
- e) *Feedback dos usuários* como forma de realimentar o processo de desenvolvimento iterativo e incremental.
- f) *Reversibilidade* de todas as ações realizadas durante o processo de desenvolvimento.

³Disponível em: <www.casespec.net/>. Acesso em: 21 jan. 2013.

⁴Disponível em: <www.techexcel.com/solutions/alm/fdd.html>. Acesso em: 21 jan. 2013.

⁵Disponível em: <fddtools.sourceforge.net/>. Acesso em: 21 jan. 2013.

⁶Disponível em: <www.dsdm.org/>. Acesso em: 21 jan. 2013.

⁷O princípio foi originalmente formulado pelo economista italiano Alfredo Pareto (século XIX), que observou que, em muitas situações, 80% das consequências são devidas a 20% das causas. Em engenharia de requisitos, pode-se verificar que, em geral, 80% do sistema advém de 20% dos requisitos.

- g) *Previsibilidade* para que o escopo e os objetivos das iterações sejam conhecidos antes do início.
- h) *Ausência de testes no escopo*, já que o método considera a realização de testes como uma atividade fora do ciclo de vida.
- i) *Comunicação* de boa qualidade entre todos os envolvidos é fundamental para o sucesso de qualquer projeto.

O DSDM não é recomendável para projetos nos quais a segurança é um fator crítico, pois a necessidade de testes exaustivos desse tipo de sistema entra em conflito com os objetivos de custo e prazo do DSDM.

O DSDM é bem mais fortemente baseado em documentação do que o *Scrum* ou o *XP*, o que até o deixa mais parecido com o Processo Unificado do que com os métodos ágeis. Uma das características do DSDM que o diferenciam do Processo Unificado, porém, é o fato de que ele não preconiza o uso de nenhuma técnica específica. Ele é, na verdade, um *framework* de processo, no qual os participantes poderão desenvolver suas atividades utilizando suas técnicas preferidas.

A versão atual do método, conhecida como *DSDM Atern*, pode ser visualizada gratuitamente no *site* do DSDM Consortium⁸. Segundo o DSDM Consortium, ela foi projetada para ser compatível com o método de gerência de projetos Prince2 (Seção 9.3), CMMI (Seção 12.3) e ISO9001 (Seção 12.1).

4.2.1 ANÁLISE DE VIABILIDADE

Na etapa de *análise de viabilidade*, será verificado se é viável usar o DSDM para o desenvolvimento do sistema e se existem outros fatores impeditivos eventuais. Nessa etapa também é avaliado se o projeto proposto realmente atenderá aos objetivos de negócios da empresa. Além disso, os primeiros riscos do projeto são levantados. A maioria dessas atividades é realizada por grupos de trabalho. Nessa etapa também é verificado se existem especialistas de domínio disponíveis, usualmente na empresa cliente, para participar dos grupos de trabalho.

Nessa etapa são gerados os seguintes artefatos:

- a) Relatório de viabilidade.
- b) Protótipo de viabilidade.
- c) Plano de desenvolvimento (Seção 6.4).
- d) Controle de risco (Capítulo 8).

4.2.2 ANÁLISE DE NEGÓCIO

Se o projeto for aprovado na análise de viabilidade, então ele passará para a etapa de análise de negócio. Na *análise de negócio* são estudadas as características do negócio e as possíveis tecnologias a serem utilizadas. A análise de negócio é semelhante à análise de requisitos tradicional, mas enfatiza o trabalho em equipe. Os grupos de trabalho devem envolver diversos tipos de especialistas, entre analistas, *designers*, usuários e especialistas no domínio.

Outra característica típica do DSDM nessa fase é o uso da técnica de *timeboxing*, que consiste em fixar previamente o prazo e o orçamento a serem utilizados. Feito isso, a variável que resta é o número de requisitos que poderão ser tratados. Dessa forma, os requisitos menos importantes poderão ficar de fora, caso o tempo ou o orçamento previamente definidos sejam insuficientes. O princípio de Pareto indica que isso não será um problema, já que 20% dos requisitos devem dar origem a 80% do sistema. Assim, se os requisitos menos importantes forem deixados de lado, sua influência final no sistema será muito pequena.

Para que o Princípio de Pareto seja aplicado à análise de requisitos, porém, é necessário que os requisitos sejam priorizados, ou seja, deve-se determinar quais são os mais importantes e os menos importantes. DSDM preconiza o uso da técnica MoSCoW (Clegg & Barker, 2004), que é um acrônimo para:

- a) *Must*: requisitos que devem estar necessariamente de acordo com as regras de negócio.
- b) *Should*: requisitos que devem ser considerados tanto quanto possível, mas que não têm um impacto decisivo no sucesso do projeto.
- c) *Could*: requisitos que podem ser incluídos, desde que não afetem negativamente os objetivos de tempo e orçamento do projeto.
- d) *Would*: requisitos que podem ser incluídos se sobrar tempo.

⁸Disponível em: <www.dsdm.org/dsdm-atern> . Acesso em: 21 jan. 2013.

Nessa etapa também é produzida a *arquitetura inicial de sistema*, indicando seus principais componentes, como classes, pacotes, componentes, nodos de processamento, servidores, meios de comunicação etc.

Finalmente, nessa etapa, a lista de riscos do projeto é atualizada de acordo com as descobertas atuais.

4.2.3 ITERAÇÃO DO MODELO FUNCIONAL

Os requisitos identificados na etapa anterior serão convertidos em um primeiro protótipo funcional na etapa de iteração do modelo funcional. Essa etapa é composta por quatro atividades:

- a) *Identificar o protótipo funcional*, ou seja, devem ser identificadas as funcionalidades que serão implementadas no ciclo atual. A base para a determinação dessas funcionalidades é o resultado da etapa de análise de negócio, ou seja, o modelo funcional.
- b) *Agenda*, que determina quando e como cada uma das funcionalidades será implementada.
- c) *Criação do protótipo funcional*, que consiste na implementação preliminar das funcionalidades definidas de acordo com a agenda.
- d) *Revisão do protótipo funcional*, que é feita tanto a partir de revisões da documentação quanto da avaliação do usuário final (semelhante ao teste de aceitação, visto na Seção 13.2.4). Essa atividade deve produzir o *documento de revisão do protótipo funcional*.

A atividade de *identificação do protótipo funcional* ainda incorpora quatro subatividades:

- a) *Análise de requisitos*: os requisitos do protótipo atual são analisados e revisados de acordo com a lista de prioridades criada na fase de análise de negócio ou na iteração anterior.
- b) *Listar requisitos funcionais da iteração atual*: selecionar na lista de prioridades os requisitos que serão abordados na iteração atual.
- c) *Listar requisitos não funcionais da iteração atual*: selecionar os requisitos não funcionais que serão abordados na iteração atual.
- d) *Criar um modelo funcional*: implica a criação de modelos funcionais do sistema preferencialmente usando notação gráfica.

A atividade de *agenda* também se subdivide em subatividades:

- a) *Determinar tempo*: segundo a técnica de *timeboxing*, o tempo disponível para as atividades da iteração deve ser predeterminado.
- b) *Definir desenvolvimento*: o plano de prototipagem deve incluir todas as atividades necessárias para o desenvolvimento do protótipo no tempo disponível.
- c) *Confirmação da agenda*: todos os participantes devem concordar com o plano de prototipagem.

A atividade de *criação do protótipo funcional* também possui três subatividades:

- a) *Investigar*: devem-se estudar detalhadamente os requisitos, bem como o modelo funcional, para definir um plano de implementação para o protótipo da iteração atual.
- b) *Refinar*: implementar e refinar o protótipo da iteração atual.
- c) *Consolidar*: consolidar ou integrar o protótipo da iteração atual com o protótipo das iterações anteriores.

Finalmente, a atividade de *revisão do protótipo* se divide em duas subatividades:

- a) *Testar protótipo*: o artefato *registro de teste* deve estar sendo atualizado e considerado em todas as etapas do DSDM. Nessa subatividade, ele será útil para que se saiba quais são os testes importantes pelos quais o protótipo deve passar para ser aceito.
- b) *Revisão final*: em função dos testes e do *feedback* dos usuários será criado o *documento de revisão de prototipagem*, que será usado para revisar a lista de requisitos e sua priorização, bem como a lista de riscos.

O modelo funcional e o protótipo funcional são os principais artefatos produzidos nessa etapa. O protótipo funcional é usado especialmente para que o usuário possa avaliar se os requisitos implementados são realmente aqueles de que ele necessita. A partir da avaliação do protótipo funcional, a lista de requisitos é revisada, bem como sua priorização e a lista de riscos.

4.2.4 ITERAÇÃO DE DESIGN E CONSTRUÇÃO

A *iteração de design e construção* é uma etapa que vai tratar da integração das funcionalidades já aprovadas em um sistema que satisfaça aos interessados. Além disso, essa fase vai comportar uma verificação mais detalhada dos requisitos não funcionais, que poderiam ter sido relegados a segundo plano no protótipo funcional. Essa fase também comporta quatro atividades:

- a) *Identificar o modelo de design*: devem-se identificar aqui os requisitos funcionais e não funcionais que precisam estar no sistema final. As *evidências de teste* obtidas a partir do protótipo da etapa anterior serão usadas para criar a *estratégia de implementação*.
- b) *Agenda*: a agenda define quando e como serão implementados os requisitos.
- c) *Criação do protótipo de design*: aqui é criado um protótipo do sistema, que pode ser utilizado pelos usuários finais e também para efeito de teste.
- d) *Revisão do protótipo*: o protótipo assim construído deve ser testado e revisado, gerando dois artefatos: *documentação para usuário* e *evidências de teste*.

A *prototipagem* é um dos pontos fundamentais do DSDM. O primeiro protótipo (funcional) deve ser desenvolvido para mostrar as principais funções do sistema, de forma que o cliente possa ter rapidamente uma visão daquilo que será implementado. O segundo protótipo (de *design*) deve incluir progressivamente todas as características não funcionais que não constam do primeiro protótipo, bem como todas as modificações solicitadas pelo cliente no processo de revisão do protótipo.

4.2.5 IMPLANTAÇÃO

A etapa de *implantação* tem como objetivo entregar o sistema aos usuários finais e é composta pelas seguintes atividades:

- a) *Orientações e aprovação do usuário*: os usuários finais aprovam o protótipo final como sistema definitivo a partir de seu uso e da observação da documentação fornecida.
- b) *Treinamento*: os usuários finais são treinados para o uso do sistema. *Usuários treinados* são considerados o artefato de saída dessa atividade.
- c) *Implantação*: quando o sistema for implantado e liberado para os usuários finais será obtido o artefato *sistema entregue*.
- d) *Revisão de negócio*: nessa atividade, o impacto do sistema sobre os objetivos de negócio é avaliado. Dependendo do resultado, o projeto passa para um ciclo posterior ou então reinicia esse ciclo a fim de refinar e melhorar os resultados.

4.2.6 PAPÉIS NO DSDM

O DSDM tem um conjunto muito peculiar de *papéis* que lhe são próprios. Como em outros processos, uma pessoa pode ter mais de um papel, mas é importante que cada um saiba quais são as responsabilidades que lhe cabem. Os papéis são os seguintes:

- a) *Campeão do projeto*: funciona como um gerente executivo. Deve ser uma pessoa com capacidade de administrar prazos e tomar decisões, já que a ela caberão sempre as decisões finais em caso de conflito.
- b) *Visionário*: tem a missão de saber se os requisitos iniciais estão adequados para que o projeto se inicie. O visionário também funciona como uma espécie de engenheiro do processo, porque tem a responsabilidade de manter as atividades de acordo com o DSDM.
- c) *Intermediador*: faz a interface da equipe de desenvolvimento com os clientes, usuários e especialistas de domínio. Sua responsabilidade é buscar e trazer as informações adequadas e necessárias para a equipe.
- d) *Anunciante*: é qualquer usuário que, por representar um importante ponto de vista, deve trazer informações frequentemente para a equipe, de preferência diariamente.
- e) *Gerente de projeto*: é responsável por manter as atividades nos prazos, com o orçamento definido e a qualidade necessária.

- f) *Coordenador técnico*: é responsável pelo projeto da arquitetura do sistema e seus aspectos técnicos.
- g) *Líder de equipe*: é um desenvolvedor com a função especial de motivar e manter a harmonia de seu grupo.
- h) *Desenvolvedor*: trabalha para transformar requisitos e modelos em código executável.
- i) *Testador*: é o responsável pelos testes do sistema.
- j) *Escrivão*: é responsável por tomar notas para registro de requisitos identificados, bem como de decisões de design tomadas ao longo do processo de desenvolvimento.
- k) *Facilitador*: disponibiliza o ambiente de trabalho e avalia o progresso das diversas equipes.

Outros papéis típicos de processo de desenvolvimento podem ser usados quando necessário, como o *designer de interface*, o gerente de banco de dados, o especialista em segurança etc.

Além disso, como em outros modelos, uma pessoa pode assumir diferentes papéis, sem prejuízo, caso a equipe seja pequena.

4.3 Scrum

Scrum é um modelo ágil para a gestão de projetos de software. No *Scrum* um dos conceitos mais importantes é o *sprint*, que consiste em um ciclo de desenvolvimento que, em geral, vai de duas semanas a um mês.

A concepção inicial do *Scrum* deu-se na indústria automobilística (Takeuchi & Nonaka, 1986), e o modelo pode ser adaptado a várias outras áreas diferentes da produção de software.

Na área de desenvolvimento de software, o *Scrum* deve sua popularidade inicialmente ao trabalho de Schwaber. Uma boa referência para quem deseja adotar o método é o livro de Schwaber e Beedle (2001), que apresenta o método de forma completa e sistemática.

4.3.1 PERFIS

Há três perfis importantes no Modelo *Scrum*:

- a) O *Scrum master*, que não é um gerente no sentido dos modelos prescritivos. Não é um líder, já que as equipes são auto-organizadas, mas um facilitador (pessoa que conhece bem o modelo) e um solucionador de conflitos.
- b) O *product owner*, ou seja, a pessoa responsável pelo projeto em si. Tem, entre outras atribuições, a de indicar quais são os requisitos mais importantes a serem tratados em cada *sprint*. O *product owner* é o responsável pelo *ROI* (*Return Of Investment*) e também por conhecer e avaliar as necessidades do cliente.
- c) O *Scrum team*, que é a equipe de desenvolvimento. Essa equipe não é necessariamente dividida em papéis como analista, *designer* e programador, mas todos interagem para desenvolver o produto em conjunto. Em geral são recomendadas equipes de 6 a 10 pessoas.

No caso de projetos muito grandes, é possível aplicar o conceito de *Scrum of Scrums*,⁹ em que vários *Scrum teams* trabalham em paralelo e cada um contribui com uma pessoa para a formação do *Scrum of Scrums*, quando então as várias equipes são sincronizadas.

4.3.2 PRODUCT BACKLOG

As funcionalidades a serem implementadas em cada projeto (requisitos ou histórias de usuário) são mantidas em uma lista chamada de *product backlog*.

Um dos princípios do manifesto ágil é usado aqui: *adaptação* em vez de planejamento. Então o *product backlog* não precisa ser completo no início do projeto. Pode-se iniciar apenas com as funcionalidades mais evidentes, aplicando o princípio de Pareto, para depois, à medida que o projeto avançar, tratar novas funcionalidades que forem sendo descobertas. Isso, porém, não significa fazer um levantamento inicial excessivamente superficial. Deve-se tentar obter com o cliente o maior número possível de informações sobre suas necessidades. É possível que aquelas que efetivamente surgiem nessa interação tenham maior relevância do que outras que forem descobertas mais adiante (Figura 4.3).

⁹Disponível em: <www.scrumalliance.org/articles/46-advice-on-conducting-the-scrum-of-scrums-meeting>. Acesso em: 21 jan. 2013.

| Product backlog (exemplo) | | | | | |
|---------------------------|-------------|------|----|--|--|
| ID | Nome | Imp. | PH | Como demonstrar | Notas |
| 1 | Depósito | 30 | 5 | Logar, abrir página de depósito, depositar R\$ 10,00, ir para a página de saldo e verificar que ele aumentou em R\$ 10,00. | Precisa de um diagrama de sequência UML. Não há necessidade de se preocupar com criptografia por enquanto. |
| 2 | Ver extrato | 10 | 8 | Logar, clicar em "Transações". Fazer um depósito. Voltar para "Transações", ver que o depósito apareceu. | Usar paginação para evitar consultas grandes ao BD. Design similar para visualizar página de usuário. |

Figura 4.3 Exemplo de *product backlog*¹⁰.

Segundo Kniberg (2007)¹¹, vários outros campos poderiam ser adicionados, mas, na sua experiência, os seis campos representados na Figura 4.3 acabam sempre sendo importantes:

- a) *Id*: identificador numérico (contador), importante para que a equipe não perca a trilha da história de usuário se, eventualmente, ela mudar de nome ou descrição.
- b) *Nome*: nome curto que representa mnemonicamente a história de usuário.
- c) *Imp.*: importância da história de usuário. Números mais altos indicam importância maior. Não há limite. A relação entre os números não deve ser interpretada como uma proporção de importância. Se uma história tem importância 10 e outra tem importância 50, isso só quer dizer que a segunda é mais importante, mas não significa que seja 5 vezes mais importante.
- d) *PH*: estimativa de esforço necessário para transformar a história em software. O valor é dado em *pontos de histórias* (*PH*). Veja mais detalhes na Seção 7.6.
- e) *Como demonstrar*: descrição do que teria que ser possível fazer para que se considerasse a história efetivamente implementada. Esse campo poderia ser considerado o código de alto nível para o módulo de teste da história.
- f) *Notas*: quaisquer outras informações julgadas importantes para a implementação da história de usuário.

4.3.3 SPRINT

O *sprint* é o ciclo de desenvolvimento de poucas semanas de duração sobre o qual se estrutura o *Scrum*.

No início de cada *sprint* é feito um *sprint planning meeting*, no qual a equipe prioriza os elementos do *product backlog* a serem implementados e transfere esses elementos do *product backlog* para o *sprint backlog*, ou seja, a lista de funcionalidades a serem implementadas no ciclo que se inicia.

A equipe se compromete a desenvolver as funcionalidades, e o *product owner* se compromete a não trazer novas funcionalidades durante o mesmo *sprint*. Se novas funcionalidades forem descobertas, serão abordadas em *sprints* posteriores.

Pode-se dizer que os dois *backlogs* têm naturezas diferentes:

- a) O *product backlog* apresenta requisitos de alto nível, bastante voltados às necessidades diretas do cliente.
- b) Já o *sprint backlog* apresenta uma visão desses requisitos de forma mais voltada à maneira como a equipe vai desenvolvê-los.

Durante o *sprint*, cabe ao *product owner* manter o *sprint backlog* atualizado, indicando as tarefas já concluídas e aquelas ainda por concluir, preferencialmente mostradas em um gráfico atualizado diariamente e à vista de todos. Um exemplo de quadro de andamento de atividades é apresentado na Figura 4.4.

Existem ferramentas, como o *Virtual Scrum Board*¹², que permitem editar esses quadros eletronicamente.

O quadro de andamento é inicializado com as histórias de usuário (primeira coluna), retiradas do *product backlog*. A equipe, então, se reúne para determinar quais são as atividades de desenvolvimento necessárias para implementar cada uma das histórias de usuário. Essa lista de atividades constitui o *sprint backlog*, e ela é usada para inicializar a segunda coluna "tarefas a fazer" do quadro de andamento. À medida que as atividades vão sendo feitas, verificadas e terminadas, as fichas correspondentes vão sendo movidas para a coluna da direita.

¹⁰Adaptado de: Kniberg (2007, p. 10).

¹¹Disponível em: <www.metaproj.com/csm/ScrumAndXpFromTheTrenches.pdf> . Acesso em: 21 jan. 2013.

¹²Disponível em: <www.downloadplex.com/Windows/Business/Project-Management/virtual-scrum-board_220058.html> . Acesso em: 21 jan. 2013.

| História | Tarefas a fazer | Em andamento | Para verificar | Terminadas |
|------------------------------|--|---------------------------|---------------------------|---|
| Como usuário, eu... 8 pontos | Codificar... Testar... Testar... | Codificar... Testar... | Codificar... | Codificar... Codificar... Codificar... Testar... |
| Como usuário, eu... 5 pontos | Codificar... Codificar... Testar... Testar... | Testar... | Codificar... Testar... | Codificar... Testar... Testar... |

Figura 4.4 Um típico quadro de andamento de tarefas de um *sprint*.

A cada dia pode-se descobrir que tarefas que não foram inicialmente previstas eram necessárias para implementar as histórias de usuário do *sprint*. Essas novas tarefas devem ser colocadas na coluna de tarefas por fazer. Porém, novas histórias de usuário não podem ser adicionadas durante o *sprint*.

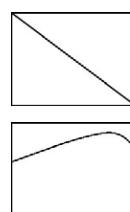
A cada dia também pode-se avaliar o andamento das atividades, contando a quantidade de atividades por fazer e a quantidade de atividades terminadas, o que vai produzir o diagrama *sprint burndown*.

O diagrama *sprint burndown* consiste basicamente de duas linhas: a primeira indica a quantidade de trabalho por fazer e a segunda indica a quantidade de trabalho feito ao longo do tempo. O gráfico é atualizado todo dia, do início até o final do *sprint*, e espera-se, numa situação ideal, que seja semelhante ao da Figura 4.5.

Porém, quando as histórias de usuário não foram bem compreendidas e/ou as tarefas não foram bem planejadas, ou ainda, quando a equipe atribui a si mesma tarefas extras, além das inicialmente previstas, o *sprint burndown* acaba ficando com uma aparência semelhante ao gráfico da Figura 4.6.

Kane Mar (2006)¹³ analisa as linhas de tarefas por fazer, identificando sete tipos de comportamentos de equipes conforme seus *sprint burndown*:

- *Fakey-fakey*: caracteriza-se por uma linha reta e regular que indica que provavelmente a equipe não está sendo muito honesta, porque o mundo real é bem complexo e dificilmente projetos se comportam com tanta regularidade.
- *Late-learner*: indica um acúmulo de tarefas até perto do final do *sprint*. É típico de equipes iniciantes que ainda estão tentando aprender o funcionamento do *Scrum*.

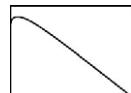


¹³Disponível em: <kanemar.com/2006/11/07/seven-common-sprint-burndown-graph-signatures/> . Acesso em: 21 jan. 2013.

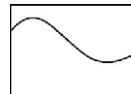
- *Middle-learner*: indica que a equipe pode estar amadurecendo e começando mais cedo as atividades de descoberta e, especialmente, os testes necessários.



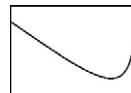
- *Early-learner*: indica uma equipe que procura, logo no início do *sprint*, descobrir todas as necessidades e depois desenvolvê-las gradualmente até o final do ciclo.



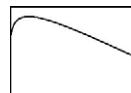
- *Plateau*: indica uma equipe que tenta balancear os aprendizados precoce e tardio, o que acaba levando o ritmo de desenvolvimento a um platô. Inicialmente, fazem bom progresso, mas não conseguem manter o ritmo até o final do *sprint*.



- *Never-never*: indica uma equipe que acaba tendo surpresas desagradáveis no final de um *sprint*.



- *Scope increase*: indica uma equipe que percebe um súbito aumento na carga de trabalho por fazer. Usualmente, a solução nesses casos é tentar renegociar o escopo da *sprint* com o *product owner*, mas não se descarta também uma finalização da *sprint* para que seja feito um replanejamento do *product backlog*.



Ao final de cada *sprint*, a equipe deve realizar um *sprint review meeting* (ou *sprint demo*) para verificar o que foi feito e, então, partir para uma nova *sprint*. O *sprint review meeting* é a demonstração e a avaliação do produto do *sprint*.

Outra reunião que pode ser feita ao final de uma *sprint* é a *sprint retrospective*, cujo objetivo é avaliar a equipe e os processos (impedimentos, problemas, dificuldades, ideias novas etc.).

4.3.4 DAILY SCRUM

O modelo sugere que a equipe realize uma reunião diária, chamada *daily scrum*, na qual o objetivo é fazer que cada membro da equipe fale sobre o que fez no dia anterior, o que vai fazer no dia seguinte e, se for o caso, o que o impede de prosseguir.

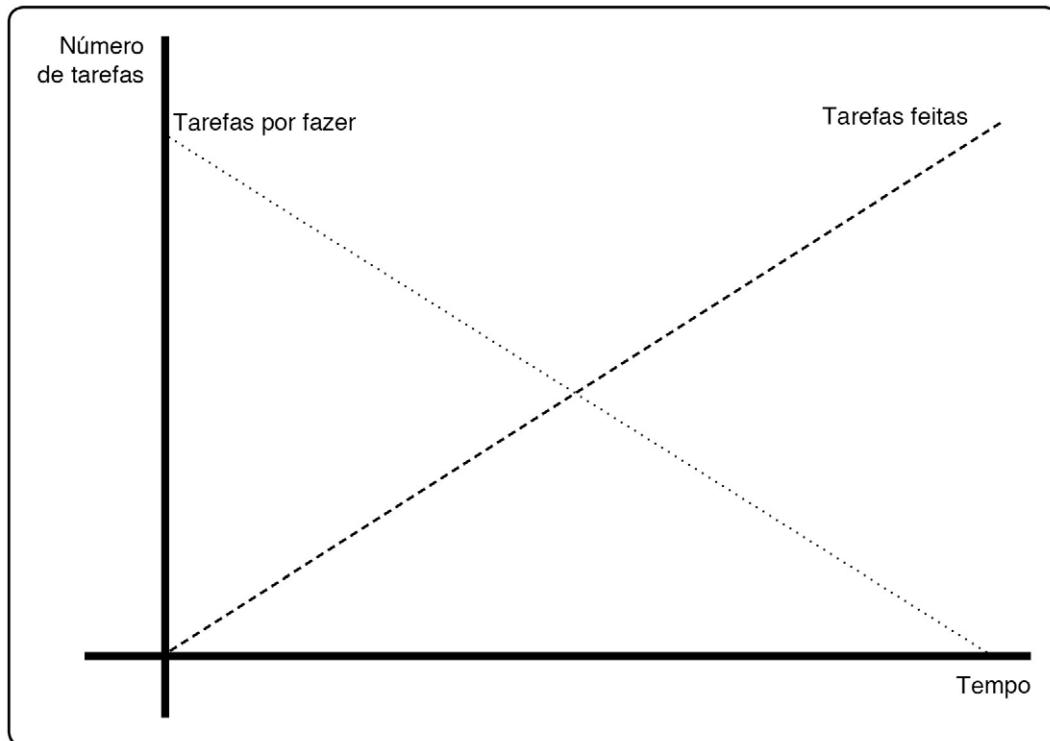


Figura 4.5 Um *sprint burndown* ideal.

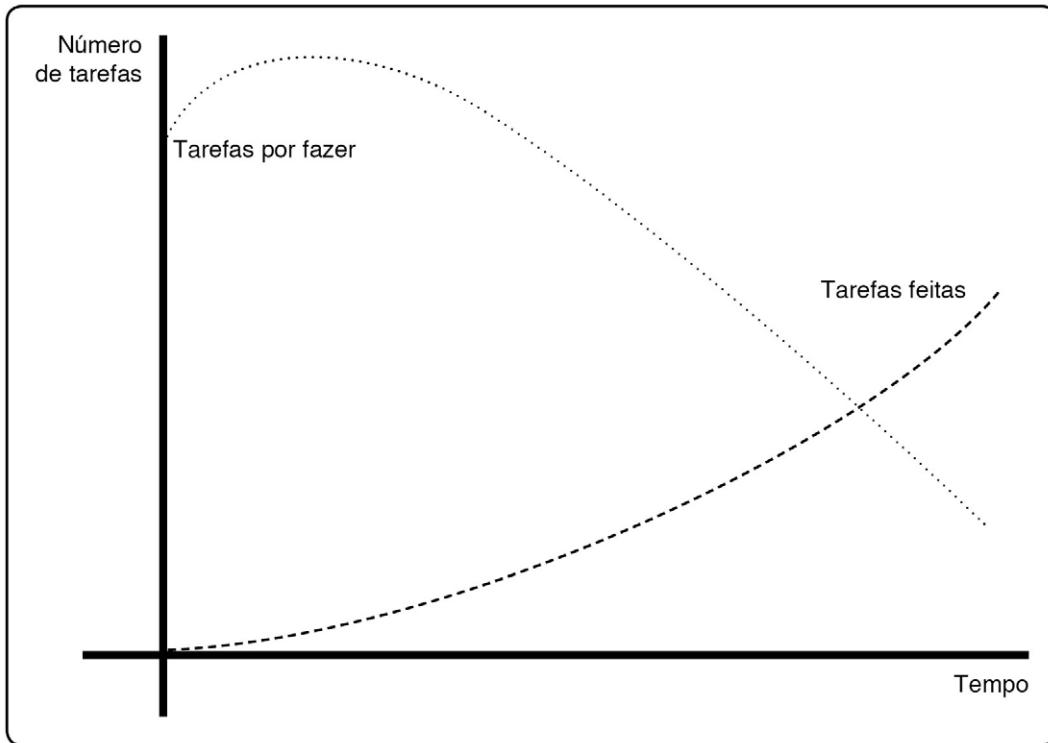


Figura 4.6 Um *sprint burndown* em que as tarefas adicionais são incluídas após o início do *sprint*.

Essas reuniões devem ser rápidas. Por isso, sugere-se que sejam feitas com as pessoas em pé na frente de um quadro de anotações. Além disso, recomenda-se que sejam feitas logo após o almoço, porque nessa hora do dia os participantes estarão mais imersos nas questões do trabalho (longe dos problemas pessoais), além de ser uma boa maneira de dissipar o cansaço que atinge os desenvolvedores no início da tarde.

É desse formato de reunião em pé, semelhante ao que jogadores de alguns esportes fazem nos intervalos, que vem o nome do método (*scrum*, em inglês).

4.3.5 FUNCIONAMENTO GERAL DO SCRUM

O funcionamento geral do Modelo *Scrum* pode ser entendido a partir do resumo apresentado na **Figura 4.7**.

Basicamente, o lado esquerdo da figura mostra o *product backlog*, com as histórias de usuário, que devem ser priorizadas e ter sua complexidade estimada. As histórias mais importantes são, assim, selecionadas durante a *sprint planning meeting*, até que o número de pontos de história se aproxime da capacidade de produção da equipe durante o *sprint*. Cada ponto de história implica um dia de trabalho por pessoa. Assim, um *sprint* de 2 semanas (10 dias de trabalho) com 3 pessoas teria a capacidade de acomodar 30 pontos de história.

Ainda durante a *sprint planning meeting*, as histórias de usuário selecionadas devem ser detalhadas em atividades de desenvolvimento, ou seja, as tarefas do *sprint backlog* devem ser identificadas.

O *sprint* se inicia e, a cada 24 horas, deve acontecer uma *scrum daily meeting*, conforme indicado no círculo menor da figura.

Ao final do *sprint* deverá haver a *sprint review meeting*, para avaliar o produto do trabalho, e, eventualmente, a *sprint retrospective*, para avaliar os processos de trabalho. Assim, se aprovado, o produto (parcial ou final) poderá ser entregue ao cliente. Não sendo esse o *sprint* final, o ciclo é reiniciado.

4.4 XP – eXtreme Programming

Programação Extrema, ou *XP* (*eXtreme Programming*), é um modelo ágil inicialmente adequado a equipes pequenas e médias e baseado em uma série de valores, princípios e regras. O *XP* surgiu nos Estados Unidos no final da década de 1990.

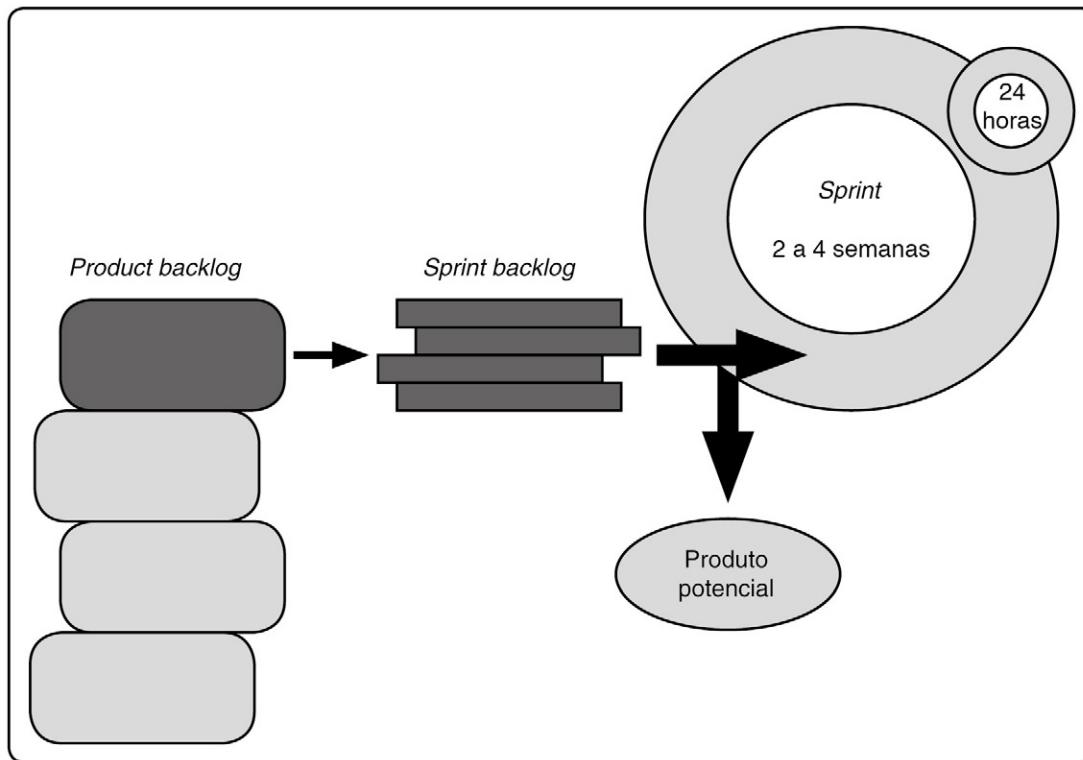


Figura 4.7 Modelo Scrum.

Entre os principais valores de XP podemos citar: simplicidade, respeito, comunicação, *feedback* e coragem:

- Simplicidade*: segundo o *Chaos Report* (Standish Group, 1995),¹⁴ mais da metade das funcionalidades introduzidas em sistemas *nunca* é usada. XP sugere como valor a simplicidade, ou seja, a equipe deve se concentrar nas funcionalidades efetivamente necessárias, e não naquelas que *poderiam* ser necessárias, mas de cuja necessidade real ainda não se tem evidência.
- Respeito*: respeito entre os membros da equipe, assim como entre a equipe e o cliente, é um valor dos mais básicos, que dá sustentação a todos os outros. Se não houver respeito, a comunicação falha e o projeto afunda.
- Comunicação*: em desenvolvimento de software, a comunicação é essencial para que o cliente consiga dizer aquilo de que realmente precisa. O XP preconiza comunicação de boa qualidade, preferindo encontros presenciais em vez de videoconferências, videoconferências em vez de telefonemas, telefonemas em vez de *e-mails*, e assim por diante. Ou seja, quanto mais pessoal e expressiva a forma de comunicação, melhor.
- Feedback*: o projeto de software é reconhecidamente um empreendimento de alto risco. Cientes disso, os desenvolvedores devem buscar obter *feedback* o quanto antes para que eventuais falhas de comunicação sejam corrigidas o mais rapidamente possível, antes que os danos se alastrem e o custo da correção seja alto.
- Coragem*: pode-se dizer que a única coisa constante no projeto de um software é a necessidade de mudança. Para os desenvolvedores XP, é necessário confiar nos mecanismos de gerenciamento da mudança para ter coragem de abraçar as inevitáveis modificações em vez de simplesmente ignorá-las por estarem fora do contrato formal ou por serem muito difíceis de acomodar.

A partir desses valores, uma série de princípios básicos é definida:

- Feedback* rápido.
- Presumir simplicidade.
- Mudanças incrementais.
- Abraçar mudanças.
- Trabalho de alta qualidade.

¹⁴Disponível em: <www.projectsmart.co.uk/docs/chaos-report.pdf> . Acesso em: 21 jan. 2013.

Então, o XP preconiza mudanças incrementais e *feedback* rápido, além de considerar a mudança algo positivo, que deve ser entendido como parte do processo. Além disso, o XP valoriza o aspecto da qualidade, pois considera que pequenos ganhos a curto prazo pelo sacrifício da qualidade não são compensados pelas perdas a médio e a longo prazo.

A esses princípios pode-se adicionar ainda a *priorização de funcionalidades mais importantes*, de forma que, se o trabalho não puder ser todo concluído, pelo menos as partes mais importantes terão sido.

4.4.1 PRÁTICAS XP

Para aplicar XP, é necessário seguir uma série de práticas que dizem respeito ao relacionamento com o cliente, a gerência do projeto, a programação e os testes:

- a) *Jogo de planejamento (planning game)*: semanalmente, a equipe deve se reunir com o cliente para priorizar as funcionalidades a serem desenvolvidas. Cabe ao cliente identificar as principais necessidades e à equipe de desenvolvimento estimar quais podem ser implementadas no ciclo semanal que se inicia. Ao final da semana, essas funcionalidades são entregues ao cliente. Esse tipo de modelo de relacionamento com o cliente é adaptativo, em oposição aos contratos rígidos usualmente estabelecidos.
- b) *Metáfora (metaphor)*: é preciso conhecer a linguagem do cliente e seus significados. A equipe deve aprender a se comunicar com o cliente na linguagem que ele comprehende.
- c) *Equipe coesa (whole team)*: o cliente faz parte da equipe de desenvolvimento e a equipe deve ser estruturada de forma que eventuais barreiras de comunicação sejam eliminadas.
- d) *Reuniões em pé (stand-up meeting)*: como no caso do *Scrum*, reuniões em pé tendem a ser mais objetivas e efetivas.
- e) *Design simples (simple design)*: implica atender a funcionalidade solicitada pelo cliente sem sofisticar desnecessariamente. Deve-se fazer aquilo de que o cliente precisa, não o que o desenvolvedor gostaria que ele precisasse. Por vezes, *design simples* pode ser confundido com *design fácil*. Nem sempre o *design simples* é o mais fácil de se implementar, e o *design fácil* pode não atender às necessidades ou então gerar problemas de arquitetura.
- f) *Versões pequenas (small releases)*: a liberação de pequenas versões do sistema pode ajudar o cliente a testar as funcionalidades de forma contínua. O XP leva esse princípio ao extremo, sugerindo versões ainda menores do que as de outros processos incrementais, como UP e *Scrum*.
- g) *Ritmo sustentável (sustainable pace)*: trabalhar com qualidade um número razoável de horas por dia (não mais de 8). Horas extras só são recomendadas quando efetivamente trouxerem aumento de produtividade, mas não podem ser rotina.
- h) *Posse coletiva (collective ownership)*: o código não tem dono e não é necessário pedir permissão a ninguém para modificá-lo.
- i) *Programação em pares (pair programming)*: a programação é sempre feita por duas pessoas em cada computador, em geral um programador mais experiente e um aprendiz. O aprendiz deve usar a máquina, enquanto o mais experiente deve ajudá-lo a evoluir em suas capacidades. Com isso, o código gerado terá sempre sido verificado por pelo menos duas pessoas, reduzindo drasticamente a possibilidade de erros. Existem sugestões também de que a programação em pares seja feita por desenvolvedores de mesmo nível de conhecimento, os quais devem se alternar na *pilotagem* do computador (Bravo, 2010).
- j) *Padrões de codificação (coding standards)*: a equipe deve estabelecer e seguir padrões de codificação, de forma que o código pareça ter sido todo desenvolvido pela mesma pessoa, mesmo que tenha sido feito por dezenas delas.
- k) *Testes de aceitação (customer tests)*: são testes planejados e conduzidos pela equipe em conjunto com o cliente para verificar se os requisitos foram atendidos.
- l) *Desenvolvimento orientado a testes (test driven development)*: antes de programar uma unidade, devem-se definir e implementar os testes pelos quais ela deverá passar.
- m) *Refatoração (refactoring)*: não se deve fugir da refatoração quando ela for necessária. Ela permite manter a complexidade do código em um nível gerenciável, além de ser um investimento que traz benefícios em médio e longo prazo.
- n) *Integração contínua (continuous integration)*: nunca se deve esperar até o final do ciclo para integrar uma nova funcionalidade. Assim que estiver viável, ela deverá ser integrada ao sistema para evitar surpresas.

As práticas XP, porém, não são consenso entre os desenvolvedores. Keefer (2003)¹⁵ afirma, entre outras coisas, que as práticas nem sempre são aplicáveis, que o estilo de trabalho não é escalável para equipes maiores e que a programação em pares acaba sendo uma atividade altamente cansativa, que só é praticável se sua duração for mantida em períodos de tempo relativamente curtos.

Além disso, Tolfo e Wazlawick (2008) demonstram que a cultura organizacional, em especial seus aspectos mais profundos, que são os valores assumidos e praticados pelas pessoas independentemente do que esteja escrito nos quadros de missão e visão da empresa, é determinante no sentido de oferecer um ambiente fértil ou hostil à implementação de métodos ágeis, em especial o XP.

4.4.2 REGRAS DE PLANEJAMENTO

Wells (2009)¹⁶ vai além das práticas XP, apontando um conjunto sucinto e bastante objetivo de regras para XP. Ele divide as regras em cinco grandes grupos: planejamento, gerência, *design*, codificação e teste. Nessa subseção e nas seguintes, essas regras, que são um detalhamento das práticas XP, serão apresentadas.

O *planejamento* é composto pelas atividades que ocorrem antes do início de um projeto ou um ciclo. Durante o planejamento, a equipe analisa o problema, seus riscos e alternativas, prioriza atividades e planeja como o desenvolvimento e as entregas vão acontecer. As regras de *planejamento* são as seguintes:

- a) *Escrever histórias de usuário*: servem ao mesmo propósito de casos de uso, mas não são a mesma coisa. São usadas no lugar do documento de requisitos. Ao contrário dos casos de uso, que são definidos pelos analistas, as histórias de usuário devem ser escritas pelos usuários considerando-se que elas são os itens de que eles precisam que o sistema faça para eles. Podem ser usadas para definir os testes de aceitação (Seção 13.2.4). A equipe deve estimar se a história pode ser implementada em uma, duas ou três semanas. Tempos maiores do que esses significam que a história deve ser subdividida em duas ou mais histórias. Menos de uma semana significa que a história está em um nível de detalhamento muito alto e precisa ser combinada com outras. Como as histórias são escritas pelo cliente, espera-se que não sejam contaminadas com aspectos técnicos.
- b) *O planejamento de entregas cria o cronograma de entregas*: é feita uma reunião de planejamento de entregas para delinear o projeto como um todo. É importante que os técnicos tomem as decisões técnicas e os administradores tomem as decisões de negócio. Deve-se estimar o tempo de cada história de usuário em termos de semanas de programação ideais¹⁷ e priorizar as histórias mais importantes do ponto de vista do cliente. Essa priorização pode ser feita com histórias impressas em cartões, que devem ser movidos na mesa ou num quadro para indicar as prioridades. As histórias são agrupadas em iterações, que só são planejadas pouco antes de serem iniciadas.
- c) *Faça entregas pequenas frequentes*: algumas equipes entregam software diariamente, o que pode ser um exagero. No pior dos casos, as entregas deveriam acontecer a cada uma ou duas semanas. A decisão de colocar a entrega em operação ou não é do cliente.
- d) *O projeto é dividido em iterações*: prefira iterações de uma a duas semanas. Não planeje as atividades com muita antecedência; deixe para planejá-las pouco antes de elas se iniciarem. Planejamento *just-in-time* é uma forma de estar sempre sintonizado com as mudanças de requisitos e arquitetura. Não tente implementar coisas que virão depois. Leve os prazos a sério. Acompanhe a produtividade. Se perceber que não vai vencer o cronograma, convoque uma nova reunião de planejamento de entregas e repasse algumas entregas para outros ciclos. Concentre-se em completar as tarefas em vez de deixar várias coisas inacabadas.
- e) *O planejamento da iteração inicia cada iteração*: no planejamento que inicia cada iteração, selecionam-se as histórias de usuário mais importantes a serem desenvolvidas e partes de sistema que falharam em testes de aceitação, as quais são quebradas em *tarefas de programação*. As tarefas serão escritas em cartões, assim como as histórias de usuário. Enquanto as histórias de usuário estão na linguagem do cliente, as tarefas de programação estão na linguagem dos desenvolvedores. Cada desenvolvedor que seleciona uma tarefa estima quanto tempo ela demorará a ser concluída. Tarefas devem ser estimadas em um, dois ou três dias ideais de

¹⁵Disponível em: <www.avocallc.com/downloads/ExtremeProgramming.pdf>. Acesso em: 21 jan. 2013.

¹⁶Disponível em: <www.extremeprogramming.org/rules.html>. Acesso em: 21 jan. 2013.

¹⁷Uma semana de programação ideal é aquela em que uma pessoa trabalha todas as horas da semana em um projeto, dedicando-se apenas a ele.

programação¹⁸. Tarefas mais curtas que um dia devem ser combinadas, e tarefas mais longas do que três dias devem ser divididas.

4.4.3 REGRAS DE GERENCIAMENTO

O gerenciamento do projeto ocorre durante sua execução. O gerenciamento busca, basicamente, garantir que as atividades sejam realizadas no prazo, dentro do orçamento e com a qualidade desejada. As regras de *gerenciamento XP* mencionadas são:

- a) *Dê à equipe um espaço de trabalho aberto e dedicado:* é importante eliminar barreiras físicas entre os membros da equipe para melhorar a comunicação. Sugere-se colocar os computadores em um espaço central para a programação em pares e mesas nas laterais da sala para que pessoas que precisam trabalhar a sós não se desconectem do ambiente. Inclua uma área para as reuniões em pé, com um quadro branco e uma mesa de reuniões.
- b) *Defina uma jornada sustentável:* trabalhar além da jornada normal é desgastante e desmoralizante. Se o projeto atrasar, é melhor reprogramar as tarefas em uma *release planning meeting*. Descubra a velocidade ideal para sua equipe e atenha-se a ela. Não tente fazer uma equipe trabalhar na velocidade de outra. Faça planos realistas.
- c) *Inicie cada dia com uma reunião em pé:* em uma reunião típica, nem sempre todos contribuem, mas pelo menos ouvem. Mantenha o mínimo de pessoas o mínimo de tempo em reuniões. As reuniões em pé não são perda de tempo, mas uma forma rápida de manter a equipe sincronizada, pois cada um dirá o que fez ontem, o que vai fazer hoje e o que o impede de prosseguir.
- d) *A velocidade do projeto é medida:* conta-se ou estima-se quantos pontos de histórias de usuário e/ou tarefas de programação são desenvolvidos em cada iteração. A cada encontro de planejamento de iteração, os clientes podem selecionar um conjunto de histórias de usuário cujo número total de pontos seja aproximadamente igual à estimativa de velocidade do projeto. O mesmo vale para os programadores em relação às tarefas de programação. Isso permite que a equipe se recupere de eventuais iterações difíceis. Aumentos e diminuições de velocidade são esperados.
- e) *Mova as pessoas:* a mobilidade de pessoas em projetos é importante para evitar a perda de conhecimento e gargalos de programação. Ficar na dependência de um único funcionário é perigoso. Deve-se evitar criar ilhas de conhecimento, porque elas são suscetíveis a perda. Se precisar de conhecimento especializado, contrate um consultor por prazo determinado.
- f) *Conserte XP quando for inadequado:* não hesite em mudar aquilo que não funciona em XP. Isso não significa que se pode fazer qualquer coisa. As regras devem ser seguidas até que se perceba que elas precisam ser mudadas. Todos os desenvolvedores devem saber o que se espera deles e o que eles podem esperar dos outros. A existência de regras é a melhor forma de garantir isso.

4.4.4 REGRAS DE DESIGN

As regras de *design* são as seguintes:

- a) *Adote a simplicidade:* um *design* simples sempre é executado mais rapidamente do que um *design* complexo. Porém, a simplicidade é subjetiva e difícil de ser medida. Então, é a equipe que deve decidir o que é simples. Uma das melhores formas de obter simplicidade em um *design* é levar a sério o *Design Pattern* “Coesão Alta” (Wazlawick, 2011), porque ele vai levar a elementos de sistema (classes, módulos, métodos, componentes etc.) mais fáceis de se compreender, modificar e estender. Recomendam-se algumas qualidades subjetivas para determinar a simplicidade de um *design*:
 - *Testabilidade:* podem-se escrever testes de unidade para verificar se o código está correto. O sistema deve poder ser quebrado em unidades testáveis, como casos de uso, fluxos, operações de sistema, métodos delegados e métodos básicos.
 - *Browseabilidade:* podem-se encontrar os elementos quando se precisa deles. Bons nomes e uso de boas disciplinas, como polimorfismo, herança e delegação, ajudam nisso.

¹⁸Um dia ideal de programação é uma jornada de trabalho normal em que uma pessoa dedique-se unicamente ao projeto.

- *Compreensibilidade e explicabilidade:* a compreensibilidade é uma qualidade subjetiva, porque um sistema pode ser bastante compreensível para quem está trabalhando nele, mas difícil para quem está de fora. Então, essa propriedade pode ser definida em termos de quanto fácil é explicar o sistema para quem não participou de seu desenvolvimento.
- b) *Escolha uma metáfora de sistema:* uma boa metáfora de sistema ajuda a explicar seu funcionamento a alguém que está fora do projeto. Deve-se evitar que a compreensão sobre o sistema resida em pilhas de documentos. Nomes significativos e padrões de nomeação de elementos de programa devem ser cuidadosamente escolhidos e seguidos para que fragmentos de código sejam efetivamente reusáveis.
- c) *Use cartões CRC durante reuniões de projeto:* trata-se de uma técnica para encontrar responsabilidades e colaborações entre objetos. A equipe se reúne em torno de uma mesa e cada membro recebe um ou mais cartões representando instâncias de diferentes classes. Uma atividade (operação ou consulta) é simulada e, à medida que ela ocorre, os detentores dos cartões anotam responsabilidades do objeto (no lado esquerdo do cartão) e colaborações do objeto (no lado direito do cartão). A documentação dos processos pode ser feita com diagramas de sequência ou de comunicação da UML.
- d) *Crie soluções afiadas (spikes) para reduzir riscos:* riscos de projeto importantes devem ser explorados de forma definitiva e exclusiva, ou seja, deve ser buscada uma *solução afiada* ou *spike* para o problema identificado. Um *spike* é, então, um desenvolvimento ou teste projetado especificamente para analisar e talvez resolver um risco. Caso o risco se mantenha, deve-se colocar um par de programadores durante uma ou duas semanas trabalhando exclusivamente para examiná-lo e mitigá-lo. A maioria dos *spikes* não será aproveitada no projeto, podendo ser classificada como uma das formas de prototipação *throw-away*.
- e) *Nenhuma funcionalidade é adicionada antes da hora:* deve-se evitar a tentação de adicionar uma funcionalidade desnecessária só porque seria fácil fazer isso no momento e deixaria o sistema melhor. Apenas o *necessário* deve ser feito no sistema. Desenvolver o que não é necessário é jogar tempo fora. Manter o código aberto a possíveis mudanças futuras tem a ver com simplicidade de *design*, mas adicionar funcionalidade ou flexibilidade desnecessária sempre deixa o *design* mais complexo e tem o efeito de uma bola de neve. Requisitos futuros só devem ser considerados quando estritamente exigidos pelo cliente. Flexibilidade em *design* é bom, mas toma tempo de desenvolvimento. Deve-se decidir quais requisitos efetivamente merecem ter uma implementação flexível.
- f) *Use refatoração sempre e onde for possível:* refatore sem pena. O XP não recomenda que se continue usando *design* antigo e ruim só porque ele funciona. Devem-se remover redundâncias, eliminar funcionalidades desnecessárias e rejuvenescer *designs* antiquados.

4.4.5 REGRAS DE CODIFICAÇÃO

As regras relacionadas à *codificação* de programas são as seguintes:

- a) *O cliente está sempre disponível:* o XP necessita que o cliente esteja disponível, de preferência pessoalmente, ao longo de todo o processo de desenvolvimento. Entretanto, em razão do longo tempo de duração de um projeto, a empresa cliente pode ser tentada a associar a ele um funcionário pouco experiente ou um estagiário. Contudo, ele não serve. Precisa ser um especialista, que deverá escrever as histórias de usuário, bem como priorizá-las e negociar sua inclusão em iterações. Pode parecer um investimento alto no tempo dos funcionários, mas isso é compensado pela ausência de necessidade de um levantamento de requisitos detalhado no início, bem como pelo fato de que não será entregue um sistema inadequado.
- b) *O código deve ser escrito de acordo com padrões aceitos:* os padrões de codificação mantêm o código compreensível e passível de refatoração por toda a equipe. Além disso, um código padronizado e familiar encoraja a sua posse coletiva.
- c) *Escreva o teste de unidade primeiro:* em geral, escrever o teste antes do código ajuda a escrever o código melhor. O tempo para escrever o teste e o código passa a ser o mesmo que se gastaria para escrever apenas o código, mas assim se obtém um código de melhor qualidade e é uma forma de garantir que ele continue correto mesmo após mudanças posteriores.
- d) *Todo código é produzido por pares:* embora pareça contrassenso, duas pessoas trabalhando em um computador podem produzir tanto código quanto duas pessoas trabalhando separadamente, porém com mais qualidade. Embora seja recomendado que haja um programador mais experiente, a relação não deve ser de professor-aluno, mas de iguais.

- e) *Só um faz integração de código de cada vez:* a integração em paralelo pode trazer problemas de compatibilidade imprevistos, pois duas partes do sistema que nunca foram testadas juntas acabam sendo integradas sem serem testadas. Deve haver versões claramente definidas do produto. Então, para que equipes trabalhando em paralelo não tenham problemas na hora de integrar seu código ao produto, elas devem esperar sua vez. Devem-se estabelecer turnos de integração que sejam obedecidos.
- f) *Integração deve ser frequente:* os desenvolvedores devem integrar o código ao repositório em curtos períodos de tempo. Postergar a integração pode agravar o problema de todos estarem trabalhando em versões desatualizadas do sistema.
- g) *Defina um computador exclusivo para integração:* esse computador funciona como uma ficha de exclusividade (*token*) para a integração. A existência dele no ambiente de trabalho permite que toda a equipe veja quem está integrando uma funcionalidade e que funcionalidade é essa. O resultado da integração deve passar nos testes de unidade de forma que se obtenha estabilidade em cada versão, além de localidade nas mudanças e possíveis erros. Se os testes de unidade falharem, essa unidade deverá ser depurada. Se a atividade de integração levar mais de dez minutos, isso significa que a unidade ainda precisa de alguma depuração adicional antes de ser integrada. Nesse caso, a integração deve ser abortada, retornando o sistema à última versão estável, e a depuração da unidade deve continuar sendo feita pelo par.
- h) *A posse do código deve ser coletiva:* não devem ser criados gargalos pela existência de donos de código. Todos devem ter autorização para modificar, consertar ou refatorar partes do sistema. Para isso funcionar, os desenvolvedores devem sempre desenvolver os testes de unidade com o código, seja novo, seja modificado. Dessa forma, existe sempre uma garantia de que o software satisfaça as condições de funcionamento. Não ter um dono de partes do sistema também diminui o impacto da perda de membros da equipe.

4.4.6 REGRAS DE TESTE

Por fim, as regras referentes ao teste do software são as seguintes:

- a) *Todo código deve ter testes de unidade* (Seção 13.2.1): esse é um dos pilares do XP. Inicialmente, o desenvolvedor XP deve criar ou obter um *framework de teste de unidade*.¹⁹ Depois, deve testar todas as classes do sistema, ignorando métodos básicos triviais, como *getters* e *setters*, especialmente se forem gerados automaticamente. Testes de unidade favorecem a posse coletiva do código, porque o protegem de ser acidentalmente danificado.
- b) *Todo código deve passar pelos testes de unidade antes de ser entregue:* exigir isso ajuda a garantir que sua funcionalidade seja corretamente implementada. Os testes de unidade também favorecem a refatoração, porque protegem o código de mudanças de funcionalidade indesejadas. A introdução de nova funcionalidade deverá levar à adição de outros testes ao teste de unidade específico.
- c) *Quando um erro de funcionalidade é encontrado, testes são criados:* um erro de funcionalidade identificado exige que testes de aceitação sejam criados para proteger o sistema. Assim, os clientes explicam claramente aos desenvolvedores o que eles esperam que seja modificado.
- d) *Testes de aceitação são executados com frequência e os resultados são publicados:* testes de aceitação são criados a partir de histórias de usuário. Durante uma iteração, as histórias de usuário selecionadas serão traduzidas em testes de aceitação. Esses testes são do tipo *funcional* (Seção 13.5) e representam uma ou mais funcionalidades desejadas. Testes de aceitação devem ser automatizados de forma que possam ser executados com frequência.

4.5 Crystal Clear

Crystal Clear é um método ágil criado por Alistair Cockburn em 1997. Ele pertence à família de métodos *Crystal*, mais ampla, iniciada em 1992 (Cockburn, 2004). Os outros métodos são conhecidos como *Yellow*, *Orange* e *Red*. *Clear* é o primeiro método da família. Cada método é indicado para uma equipe cada vez maior (de até 8, 20, 40 e

¹⁹Um bom site para baixar *frameworks* para testes de unidade envolvendo várias linguagens é <www.xprogramming.com/software> . Além disso, em <www.junit.org/> pode-se encontrar o *JUnit*, que vem se tornando um padrão para desenvolvimento dirigido por testes em Java. Acessos em: 21 jan. 2013.

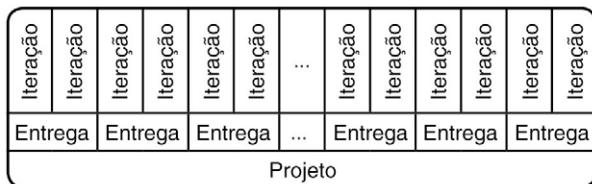


Figura 4.8 Estrutura do ciclo de vida do *Crystal Clear*.

100 desenvolvedores, respectivamente) e de maior risco (desconforto, pequenas perdas financeiras, grandes perdas financeiras e morte, respectivamente). À medida que cresce o tamanho da equipe e do risco do projeto, os métodos vão ficando cada vez mais formais. Assim, *Crystal Clear* é o mais ágil de todos.

Crystal Clear é, portanto, uma abordagem ágil adequada a equipes pequenas (de no máximo 8 pessoas) que trabalham juntas (na mesma sala ou em salas contíguas). Em geral, a equipe é composta por um *designer* líder e por mais dois a sete programadores. O método propõe, entre outras coisas, o uso de radiadores de informação, como quadros e murais à vista de todos, acesso fácil a especialistas de domínio, eliminação de distrações, cronograma de desenvolvimento baseado na técnica de *timeboxing* e ajuste do método quando necessário.

O ciclo de vida de *Crystal Clear* é organizado em três níveis (Figura 4.8):

- a) A *iteração*, composta por estimação, desenvolvimento e celebração, que costuma durar poucas semanas.
- b) A *entrega*, formada por várias iterações, que no espaço máximo de dois meses vai entregar funcionalidades úteis ao cliente.
- c) O *projeto*, formado pelo conjunto de todas as entregas.

Segundo Cockburn (2004)²⁰, a família *Crystal* é centrada em pessoas (*human powered*), ultraleve e na medida (*stretch to fit*):

- a) *Centrada em pessoas*: significa que o foco para o sucesso de um projeto está em melhorar o trabalho das pessoas envolvidas. Enquanto outros métodos podem ser centrados em processo, em arquitetura ou em ferramenta, *Crystal* é centrado em pessoas.
- b) *Ultraleve*: significa que independentemente do tamanho do projeto, a família *Crystal* fará o possível para reduzir a burocracia, a papelada e o *overhead*, que existirão na medida suficiente para as necessidades do projeto.
- c) *Na medida*: significa que o *design* começa com algo menor do que se pensa que seja preciso e, depois, é aumentado apenas o suficiente para suprir as necessidades. Parte-se do princípio de que é mais fácil e barato aumentar um sistema do que cortar coisas que já foram feitas mas são desnecessárias.

Os sete pilares do método são listados a seguir. Os três primeiros são condições *sine qua non* do método; os outros quatro são recomendados para levar a equipe à zona de conforto em relação a sua capacidade de desenvolver software de forma adequada:

- a) *Entregas frequentes*: as entregas ao cliente devem acontecer no máximo a cada dois meses, com versões intermediárias.
- b) *Melhoria reflexiva*: os membros da equipe devem discutir frequentemente se o projeto está no rumo certo e comunicar descobertas que possam impactar o projeto.
- c) *Comunicação osmótica*: a equipe deve trabalhar em uma única sala para que uns possam ouvir a conversa dos outros e participar dela quando julgarem conveniente. Considera-se uma boa prática interferir no trabalho dos outros. O método propõe que os programadores trabalhem individualmente, mas bem próximos uns dos outros. Isso pode ser considerado um meio-termo entre a programação individual e a programação em pares, pois cada um tem a sua atribuição, mas todos podem se auxiliar mutuamente com frequência.
- d) *Segurança pessoal*: os desenvolvedores devem ter certeza de que poderão falar sem medo de repreensões. Quando as pessoas não falam, suas fraquezas viram fraquezas da equipe.
- e) *Foco*: espera-se que os membros da equipe tenham dois ou três tópicos de alta prioridade nos quais possam trabalhar tranquilamente, sem receber novas atribuições.

²⁰Disponível em: <alistair.cockburn.us/Crystal+methodologies>. Acesso em: 21 jan. 2013.

- f) *Acesso fácil a especialistas:* especialistas de domínio, usuários e cliente devem estar disponíveis para colaborar com a equipe de desenvolvimento.
- g) *Ambiente tecnologicamente rico:* o ambiente de desenvolvimento deve permitir testes automáticos, gerenciamento de configuração e integração frequente.

Considera-se que aplicar *Crystal Clear* tem mais a ver com adquirir as propriedades mencionadas do que seguir procedimentos.

4.5.1 ENTREGAS FREQUENTES

As vantagens de *entregas frequentes* como forma de redução de risco em um projeto de software são indiscutíveis, e os ciclos de vida modernos aderem a esse princípio.

Na maioria das vezes em que se fala em entregas frequentes, imaginam-se sistemas feitos sob medida para um cliente conhecido e interessado em dar *feedback* para o processo de desenvolvimento, mas nem sempre o software é feito para esse tipo de cliente. Muitos sistemas desenvolvidos hoje são distribuídos pela internet; então a comunidade de usuários pode não ser totalmente conhecida. Nesses casos, a tática de efetuar entregas (disponibilização de versões) com muita frequência (por exemplo, semanal) pode ser irritante para alguns usuários. De outro lado, diminuir a frequência das entregas pode fazer que a equipe de desenvolvimento perca um importante *feedback*. A solução, nesse caso, é definir um conjunto de usuários amigáveis que não se importem em receber versões com frequência maior do que os usuários normais.

Se usuários amigáveis não forem encontrados, a sugestão é que o ciclo de desenvolvimento seja finalizado como se a entrega fosse ser feita, ou seja, uma falsa entrega deve ser criada, com toda a formalidade, como se fosse uma entrega verdadeira, mas não deve ser disponibilizada aos usuários.

Uma *iteração*, que possivelmente produz uma entrega, não deve ser confundida com uma *integração*. A integração pode ocorrer de hora em hora, sempre que algum programador tiver criado uma nova versão de um componente que possa ser integrado ao sistema. Já a iteração pressupõe o fim de um ciclo de atividades predefinidas e controladas, incluindo várias integrações ao longo do ciclo. Para *Crystal Clear*, é importante que o fim de um ciclo seja marcado com uma celebração, pois, assim, a equipe ganhará ritmo emocional com a sensação de etapa concluída. Afinal, ela é formada por pessoas, e não máquinas.

Crystal Clear assume que uma iteração possa durar de uma hora a três meses. Contudo, o mais comum é que as iterações durem de duas semanas a dois meses, como no Processo Unificado. O importante é que seja usada a técnica de *timeboxing* e que o prazo final das iterações não seja mudado, pois um atraso levará a outros e o ritmo emocional da equipe poderá baixar.

Uma estratégia melhor é manter o prazo, deixar a equipe disponibilizar aquilo que for possível naquele intervalo de tempo e, depois, se necessário, replanejar as iterações seguintes.

Algumas equipes poderão tentar usar a técnica de requisitos fixos (*requirements locking*), ou seja, assumir que durante uma iteração os requisitos ou suas prioridades não poderão mudar. Isso permitirá à equipe saber que poderá completar suas atribuições sem mudanças de rumo no meio do processo. Normalmente, porém, em ambientes não hostis e bem comportados, não é necessário estabelecer isso como regra, pois acaba acontecendo naturalmente.

É importante frisar que entregas frequentes têm a ver com entregar software ao cliente, e não simplesmente completar iterações. Algumas equipes poderão fazer que cada iteração corresponda a uma entrega; outras entregarão software a cada 2, 3 ou 4 iterações; outras ainda definirão no calendário datas específicas para iterações que produzirão entregas. Em todos os casos, não basta a equipe fazer iterações rápidas; é necessário entregar software com frequência. Não adianta fazer 24 iterações ao longo de um ano, mas nenhuma entrega, pois nesse meio-tempo o cliente não terá dado nenhum *feedback* sobre o que foi desenvolvido.

4.5.2 MELHORIA REFLEXIVA

Uma das coisas que podem fazer um projeto que está falhando dar a volta por cima é a *melhoria reflexiva*. Essa prática indica que a equipe deve se reunir, discutir o que está e o que não está funcionando, avaliar formas de melhorar o que não está funcionando e, o que é mais importante, colocar mudanças em prática. Não é necessário gastar muito tempo com essa atividade. Poucas horas por mês normalmente são suficientes.

É interessante observar que muitos projetos enfrentam grandes dificuldades já nas primeiras iterações. Entretanto, o que poderia levar a uma catástrofe logo de início deve ser considerado um ponto de partida para reflexão e aprimoramento das práticas (refletir e melhorar). De outro lado, caso esses problemas não sejam seriamente abordados logo no início, poderão minar o projeto rapidamente e desmoralizar a equipe de forma que se tornará impossível retomar o ritmo, o que fatalmente levará ao cancelamento do projeto.

As mudanças que precisam ser feitas às vezes envolvem pessoas, outras vezes, tecnologia, e, em outras, as práticas de projeto da equipe. A sugestão da *Crystal Clear* é que, a cada semana, mensalmente ou uma ou duas vezes por ciclo de desenvolvimento, a equipe se reúna em um *workshop* de reflexão ou retrospectiva de iteração para discutir as coisas que estão funcionando e aquelas que não estão funcionando. É preciso ser feita uma lista das coisas que serão mantidas e daquelas que devem mudar. Essas listas devem ser colocadas à vista de todos para que sejam gravadas e efetivamente mudadas nas iterações seguintes.

4.5.3 COMUNICAÇÃO OSMÓTICA

Comunicação osmótica é aquela em que a informação deve fluir pelo ambiente, ou seja, as pessoas devem ser capazes de ouvir a conversa das outras e intervir, se desejarem, ou continuar seu trabalho. Isso costuma ser obtido quando se colocam todos os desenvolvedores em uma mesma sala. Além disso, é importante que as telas dos computadores sejam acessíveis, pois em alguns casos é interessante que um pequeno grupo possa se reunir em frente a um computador para visualizar problemas e dar sugestões. Assim, devem ser evitados *designs* de sala que impeçam esse tipo de visualização.

Segundo Cockburn (2004), quando a comunicação osmótica ocorre, as questões e respostas fluem naturalmente pelo ambiente e, surpreendentemente, com pouca perturbação para a equipe. Ele coloca a seguinte questão: “Leva mais de 30 segundos para a sua pergunta chegar aos olhos e ouvidos de alguém que possa respondê-la?”. Se a resposta for sim, o projeto pode enfrentar dificuldades.

A comunicação osmótica tem custo baixo, mas é altamente eficiente em termos de *feedback*, pois os erros são corrigidos antes de se tornarem problemas mais sérios e a informação é disseminada rapidamente.

Embora a comunicação osmótica seja valiosa também para projetos e equipes de grande porte, fica cada vez mais difícil obtê-la nessas condições. Pode-se tentar deixar as equipes em salas próximas, mas, ainda assim, a comunicação osmótica só vai ocorrer entre pessoas da mesma sala. Outra possibilidade, no caso de equipes grandes ou distribuídas, seria utilizar ferramentas de comunicação, como videoconferência e *chat on-line*, de forma que as questões sejam colocadas de uma pessoa para a outra, mas vistas por toda a equipe.

Um problema que pode surgir com a comunicação osmótica é o excesso de ruído na sala ou um fluxo de informação muito grande dirigido ao desenvolvedor mais experiente. Porém, equipes conscientes acabam se autorregulando e autodisciplinando para evitar tais problemas. Isolar o programador líder em outra sala acaba não sendo uma boa solução, pois, se ele é o mais experiente, é natural que acabe sendo muito requisitado, e esse é exatamente o seu papel: ajudar os demais programadores a crescer. Ter o programador líder na mesma sala onde trabalha a equipe é uma estratégia denominada *Expert in the Earshot* (especialista ao alcance do ouvido). Porém, sempre existem situações extremas. Se o programador líder for tão requisitado pela equipe que não consegue mais fazer avanços em seu próprio trabalho, deverá reservar para si horários em que não estará disponível para a equipe. Essa técnica é denominada “cone de silêncio”. O horário deve ser estabelecido de acordo com a necessidade e respeitado por todos.

4.5.4 SEGURANÇA PESSOAL

Segurança pessoal tem relação com o fato de que as pessoas podem falar sobre coisas que estão incomodando sem temer represálias ou reprimendas. Segundo Cockburn (2004), isso envolve, entre outras coisas, dizer ao gerente que o cronograma não é realístico, que o *design* de um colega precisa melhorar ou até mesmo que ele precisa tomar banho com mais frequência. A segurança pessoal é muito importante, porque com ela a equipe consegue descobrir quais são suas fraquezas e repará-las. Sem ela, as pessoas não vão falar e as fraquezas vão continuar minando a equipe.

A segurança pessoal é um passo na direção da *confiança*, que consiste em dar ao outro poder sobre si mesmo. Algumas pessoas confiam no outro até que uma prova em contrário as faça rever essa confiança; outras evitam confiar até que tenham segurança de que o outro não vai prejudicá-las.

Existem várias formas pelas quais uma pessoa pode prejudicar outras no ambiente de trabalho ou até mesmo prejudicar o trabalho. Há pessoas que mentem, pessoas que são incompetentes, pessoas que sabotam o trabalho dos outros tantoativamente quanto por não lhes fornecer informações ou orientações importantes quando necessário. Considerando essas formas de prejuízo, pode ser pedir demais às pessoas de uma equipe que simplesmente confiem umas nas outras. Assim, é mais fácil iniciar com comunicação franca, em que cada uma dirá o que a incomoda e a equipe vai regular ações e comportamentos a partir disso.

Segundo Cockburn (2004), estabelecer confiança envolve expor as pessoas a situações em que outros poderiam prejudicá-las e mostrar que isso não acontece. Por exemplo, um chefe pode expor um erro no *design* de um desenvolvedor e, em vez de puni-lo, dar-lhe suporte para que corrija o erro, mostrando que isso é parte do processo de autodesenvolvimento.

É importante mostrar que as pessoas não serão prejudicadas mesmo se demonstrarem ignorância sobre algum assunto em relação a sua área de conhecimento, ressaltando que lacunas de conhecimento sempre são oportunidades para aprender mais.

Além disso, é importante conscientizar as pessoas a interpretarem a forma de os outros se comunicarem como não agressivas, mesmo durante uma discussão. Uma discussão pode ser motivo para uma briga, mas em um ambiente saudável deve ser uma forma de confrontar diferentes pontos de vista. Mesmo que não haja consenso, o respeito pela opinião do outro deve prevalecer mesmo se for contra as próprias convicções. As pessoas devem ouvir umas às outras com boa vontade, e opiniões diversas sempre devem ser interpretadas como possibilidades ou oportunidades de aprender algo.

Isso tudo é importante para que as pessoas percebam que, com a ajuda dos outros, poderão resolver melhor problemas complexos do que se tentassem sozinhas.

A confiança é reforçada pelo princípio de entregas frequentes, porque, no momento de uma entrega, será possível ver quem realmente fez seu trabalho e quem falhou. Com segurança pessoal, todos poderão falar de seus problemas e limitações nos *workshops* de melhoria reflexiva, para que as falhas sejam minimizadas ou eliminadas no futuro.

4.5.5 Foco

Foco implica primeiramente saber em que se vai trabalhar e, depois, contar com tempo, espaço e paz de espírito para fazer o trabalho. Saber quais são as prioridades é algo usualmente determinado pela gerência. O tempo e a paz de espírito vêm de um ambiente de trabalho onde as pessoas não são arrancadas de suas atividades para realizar outras, muitas vezes sem relação com o que se faz originalmente.

Apenas definir prioridades para os desenvolvedores não é suficiente. Deve-se permitir a eles efetivamente concentrar-se nessas atividades. Um desenvolvedor interrompido de sua linha de raciocínio para apresentar relatórios ou demos de última hora, participar de reuniões ou consertar defeitos recém-descobertos gastará vários minutos para retomar sua linha de raciocínio após a interrupção. Se essas interrupções acontecerem várias vezes ao dia, não é incomum que o desenvolvedor passe a ficar ocioso nos intervalos, apenas esperando a próxima interrupção – se ele for interrompido sempre que estiver retomando o ritmo de trabalho, logo vai perceber a futilidade de tentar se concentrar.

Note-se, porém, que esse princípio não contradiz o da comunicação osmótica. No caso da comunicação osmótica, cada um decide se deseja parar o que está fazendo para responder a alguma pergunta ou auxiliar alguém. Isso costuma tomar poucos segundos ou, no máximo, alguns minutos. Contudo, uma tarefa de última hora, trazida de forma coercitiva ao desenvolvedor que tenta se concentrar em seu trabalho, é diferente: é uma interrupção que poderá afastá-lo do trabalho por muitos minutos ou até mesmo horas.

Sem dúvida, podem ocorrer interrupções de alta prioridade, mas são poucas as tarefas que não podem esperar algumas horas ou até mesmo alguns dias para serem realizadas. Ou seja, o importante é saber qual é a real urgência da tarefa e colocá-la em seu devido lugar na lista de prioridades. E, a não ser que seja algo realmente muito importante, a tarefa atual não deve ser interrompida. A nova tarefa deverá ficar na pilha de prioridades aguardando a finalização da tarefa atual para então ser iniciada.

Pessoas que trabalham em vários projetos ao mesmo tempo dificilmente farão algum progresso em qualquer um deles. Segundo Cockburn (2004), pode-se gastar até uma hora e meia para retomar a linha de pensamento quando se passa de um projeto a outro.

Gerentes de projeto experientes concordam que uma pessoa consegue ser efetiva em um ou dois projetos simultaneamente, mas, quando assume um terceiro projeto, ela passa a não ser efetiva nos três.

Quando um desenvolvedor está atulhado com vários projetos e atividades simultâneas, a solução gerencial é definir a lista de prioridades e qual é a atividade (ou atividades) que deve ser terminada o quanto antes. Enquanto ele estiver focado nessa atividade, as outras vão aguardar sua vez.

Quando a empresa faz rodízio de funcionários entre projetos (o que é saudável), uma das formas de manter o foco nas atividades é garantir que cada um deles fique um tempo mínimo (por exemplo, dois dias) num projeto antes de ser realocado para outro. Isso dá ao funcionário a tranquilidade de saber que seu esforço inicial para entrar no ritmo do projeto não será bruscamente interrompido antes que ele tenha oportunidade de produzir algo de valor.

4.5.6 ACESSO FÁCIL A ESPECIALISTAS

Não há dúvida de que o acesso fácil a especialistas ajuda muito o desenvolvimento de um projeto de software, uma vez que, embora os desenvolvedores sejam especializados em sistemas, não o são *naquele* que estão desenvolvendo. Infelizmente, essa característica não é das mais fáceis de se obter em um projeto.

Os usuários e clientes serão necessários antes, durante e depois do desenvolvimento. *Antes* para apresentar os requisitos e os objetivos de negócio, *durante* para esclarecer dúvidas que invariavelmente surgem ao longo do desenvolvimento, *depois* para validar o que foi desenvolvido.

Cockburn (2004) afirma que, no mínimo uma hora por semana seria essencial para que um especialista no domínio, usuário ou cliente estivesse disponível para responder às dúvidas da equipe de desenvolvimento. Mais tempo do que isso seria certamente salutar; menos, poderia levar o projeto a ter sérios problemas.

Outro problema relacionado a isso é o tempo que uma dúvida dos desenvolvedores leva para ser resolvida. Se uma dúvida demorar para ser respondida, os desenvolvedores poderão incorporar ao código sua melhor estimativa e depois se esquecer disso. Assim, o sistema só vai mostrar que tem uma não conformidade bem mais adiante, quando for liberado para uso pelo cliente.

Para evitar que dúvidas importantes demorem muito a serem respondidas é fundamental que, se o especialista não puder estar fisicamente presente no ambiente de desenvolvimento todas as horas da semana, exista uma forma de comunicação imediata com ele, como o telefone.

Cockburn (2004) também indica três principais estratégias para ter acesso fácil a especialistas:

- a) *Reuniões uma ou duas vezes por semana com usuário e telefonemas adicionais*: o usuário dará muitas informações importantes à equipe nas primeiras semanas. Depois, a necessidade que a equipe terá dele vai diminuir gradativamente. Um ritmo natural vai se constituindo com o usuário informando requisitos e avaliando o software desenvolvido a cada ciclo iterativo. Alguns poucos telefonemas adicionais durante a semana ajudarão a equipe a não investir tempo e esforços na direção errada.
- b) *Um ou mais especialistas permanentemente na equipe de desenvolvimento*: essa é uma situação mais difícil de ser conseguida. As opções são colocar a equipe para trabalhar dentro da empresa-cliente ou coalocada com algum usuário.
- c) *Enviar desenvolvedores para trabalhar como trainees com o cliente por algum tempo*: por mais estranha que essa opção possa parecer, ela é bastante válida, pois os desenvolvedores terão uma visão muito clara do negócio do cliente e entenderão como o sistema que vão desenvolver poderá ajudar a melhorar seu modo de trabalho.

Uma coisa importante, nesse aspecto, é entender que não existe um único tipo de usuário. Há os clientes, que usualmente são as pessoas que pagam pelo sistema; os gerentes de alto e baixo escalão; os especialistas de domínio (aqueles que conhecem ou definem as políticas); e os usuários finais (os que efetivamente usam o sistema). É importante que a equipe de desenvolvimento tenha acesso a cada um deles no momento certo e entenda suas necessidades.

4.5.7 AMBIENTE TECNOLOGICAMENTE RICO

Uma equipe, para estar na zona de conforto de desenvolvimento, deve ter um *ambiente tecnologicamente rico*, não apenas linguagens de programação e ferramentas para desenhar diagramas, mas os três pilares de um bom ambiente de desenvolvimento de software: *teste automatizado*, *sistema de gerenciamento de configuração* e *integração frequente*.

Uma equipe será produtiva de fato quando conseguir fazer integrações frequentes de versões automaticamente controladas e testadas. Dessa forma, o processo de geração de novas versões de um sistema poderá levar poucos minutos e a confiabilidade nessa integração será bastante alta.

4.5.7.1 Teste Automatizado

O teste automatizado não pode ser considerado propriedade essencial de um processo de desenvolvimento, porque as equipes que fazem testes manuais também conseguem produzir com qualidade. Mas a automatização do teste poupa tanto tempo e dá tanta tranquilidade aos desenvolvedores que é um movimento muito importante em direção à zona de conforto.

O teste automatizado implica que a pessoa responsável pelo teste possa, a qualquer momento, iniciá-lo e sair para fazer outra coisa enquanto ele é realizado. Não deve haver necessidade de intervenção humana no teste. Os resultados poderão até mesmo ser publicados na web ou na intranet, de forma que todos os desenvolvedores possam acompanhar em tempo real o que está sendo testado e os resultados disso.

Além disso, deve ser possível combinar sequências de testes individuais para gerar um conjunto de teste completo para o sistema, que poderá, eventualmente, ser rodado nos fins de semana para garantir que novos defeitos não sejam inadvertidamente introduzidos.

É conveniente, porém, lembrar que esse tipo de teste é feito unicamente para detectar defeitos no *design*. Para validar o sistema quanto aos requisitos de usuário é necessário que uma avaliação seja feita por um ser humano. O mesmo também é necessário para avaliar a usabilidade de uma interface.

A automatização de testes de sistema a partir do uso de uma interface gráfica é, em geral, uma atividade cara e que consome tempo. Além disso, qualquer modificação na interface pode fazer que a suíte de testes tenha que ser refeita. Assim, normalmente, tais testes não estão entre os mais recomendados. Para que testes de sistema sejam automatizados, é necessário que a equipe invista em uma arquitetura do tipo *MVC* (*Model/View/Controller*) (Reenskaug, 2003),²¹ na qual os testes de sistema possam ser feitos pelo acesso sequencial de funções da controladora sem passar pela interface gráfica (*view*).

4.5.7.2 Sistema de Gerenciamento de Configuração

O sistema de gerenciamento de configuração também ajuda a equipe a trabalhar com mais tranquilidade, pois caminhos que eventualmente são seguidos, mas não se mostram tão promissores quanto pareciam no início, podem ser desfeitos sem maiores consequências (Capítulo 10).

4.5.7.3 Integrações Frequentes

Quanto mais frequentemente a equipe fizer a integração de partes do sistema, mais rapidamente os erros serão detectados. Deixar de realizar integrações frequentes faz que erros se acumulem e, dessa forma, comecem a se multiplicar, e um conjunto de erros é muito mais difícil de consertar do que cada um deles.

Não há uma resposta definitiva sobre qual deve ser a frequência de integração, assim como não há uma resposta única sobre a duração dos ciclos iterativos. Caberá à equipe avaliar a forma mais orgânica de fazer as integrações, mas sempre lembrando que postergá-las muito tempo poderá gerar problemas.

4.6 ASD – Adaptive Software Development

O *Adaptive Software Development* (ASD)²² é um método ágil, criado por Jim Highsmith (2000), que aplica ideias oriundas da área de sistemas adaptativos complexos (*teoria do caos*²³). Ele vê o processo de desenvolvimento de software como um sistema complexo com *agentes* (desenvolvedores, clientes e outros), *ambientes* (organizacional, tecnológico e de processo) e *saídas emergentes* (produtos sendo desenvolvidos).

O modelo fundamenta-se em desenvolvimento cíclico iterativo baseado em três grandes fases: *especular, colaborar e aprender*.

²¹Disponível em: <heim.ifi.uio.no/~trygver/2003/javazone-jao0/HM1A93.html>. Acesso em: 21 jan. 2013.

²²Disponível em: <www.adaptivesd.com/>. Acesso em: 21 jan. 2013.

²³Disponível em: <www.santafe.edu/>. Acesso em: 21 jan. 2013.

4.6.1 ESPECULAR

“Especular” corresponde ao planejamento adaptativo de ciclo. Em vez de planejar, o modelo assume que o mais provável, nos momentos iniciais, é que os interessados ainda não saibam exatamente o que vão fazer e, por isso, especulem.

Nessa fase, porém, objetivos e prazos devem ser estabelecidos. O plano de desenvolvimento será baseado em componentes. A especulação implica a realização das seguintes atividades:

- a) Determinar o tempo de duração do projeto, o número de ciclos e sua duração ideal (Seção 6.4).
- b) Descrever um grande objetivo para cada ciclo.
- c) Definir componentes de software para serem desenvolvidos a cada ciclo.
- d) Definir a tecnologia e o suporte para os ciclos.
- e) Desenvolver a lista de tarefas do projeto.

4.6.2 COLABORAR

“Colaborar” corresponde à engenharia concorrente de componentes. A equipe deve, então, tentar equilibrar seus esforços para realizar as atividades que podem ser mais previsíveis e aquelas que são naturalmente mais incertas. A partir dessa colaboração, vários componentes serão desenvolvidos de forma concorrente.

4.6.3 APRENDER

“Aprender” corresponde à revisão de qualidade. Os ciclos de aprendizagem são baseados em pequenas interações de projeto, codificação e teste, em que os desenvolvedores, ao cometerem pequenos erros baseados em hipóteses incorretas, aprimoram seu conhecimento sobre o problema até dominá-lo.

Essa fase exige repetidas revisões de qualidade e testes de aceitação com a presença do cliente e de especialistas do domínio. Três atividades de aprendizagem são recomendadas:

- a) *Grupos de revisão com foco de usuário*: um workshop, em que os desenvolvedores apresentam o produto e os usuários tentam usá-lo, apresentando suas observações.
- b) *Inspeções de software*: inspeções (Seção 11.4.2) e testes (Capítulo 13) que têm como objetivo detectar defeitos do software.
- c) *Post-mortems*: neles a equipe avalia o que fez no ciclo e, se necessário, muda o modo de trabalhar.

Essas três fases não são necessariamente sequenciais, mas podem ocorrer de forma simultânea e não linear.

4.6.4 INICIALIZAÇÃO E ENCERRAMENTO

Além dessas três fases iterativas, o modelo prevê uma fase inicial e uma fase final, que ocorrem uma única vez cada uma:

- a) *Inicialização de projeto*: preparação para iniciar os ciclos iterativos. A inicialização deve ocorrer com um workshop de poucos dias, dependendo do tamanho do projeto, em que a equipe vai estabelecer as missões, ou objetivos, do projeto. Nessa fase também serão levantados requisitos iniciais, restrições e riscos, bem como feitas as estimativas iniciais de esforço.
- b) *Garantia final de qualidade e disponibilização*: inclui os testes finais e a implementação do produto.

4.6.5 JUNTANDO TUDO

O modelo ASD pode ser resumido pelo diagrama da Figura 4.9.

O modelo ASD é semelhante aos outros modelos ágeis já vistos nos seguintes aspectos:

- a) É baseado em ciclos iterativos de 4 a 8 semanas.
- b) Os prazos são pré-fixados (*timeboxing*).
- c) É tolerante à mudança e à adaptação.
- d) É orientado a desenvolver primeiramente os elementos de maior risco.

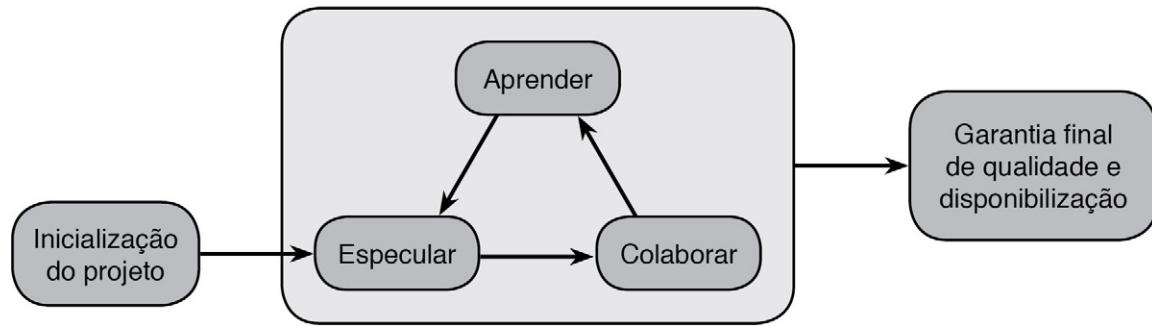


Figura 4.9 Modelo ASD.

Talvez o modelo ASD não seja tão popular nem tão detalhado quanto o XP e o *Scrum*, mas tem o mérito de enfatizar a necessidade do desenvolvimento adaptativo. Nos modelos prescritivos, normalmente a adaptação é entendida como um desvio, causado por um erro. Mas em ASD a adaptação é o caminho que leva os desenvolvedores na direção correta, a qual eles seriam incapazes de conhecer de antemão.

UP – Processo Unificado

Este capítulo apresenta o Processo Unificado, um dos mais importantes padrões da indústria de software atual. Inicialmente, são explicadas suas *características* principais (Seção 5.1) e, em seguida, suas *quatro fases* originais (Seção 5.2). Na sequência, algumas implementações específicas do Processo Unificado são apresentadas, iniciando pelo *Processo Unificado Rational* ou RUP (Seção 5.3), a mais conhecida de todas, seguida da apresentação de AUP (Seção 5.4) e OpenUP (Seção 5.5), que são implementações ágeis do Processo Unificado. EUP (Seção 5.6) é uma implementação que acrescenta aspectos empresariais que não constam na definição original do processo. OUM (Seção 5.7) é uma implementação para uso específico com ferramentas proprietárias. E, por fim, o RUP-SE (Seção 5.8), que é uma implementação orientada a sistemas complexos de grande porte.

O Processo Unificado (UP ou *Unified Process*) foi criado por três importantes pioneiros da orientação a objetos nos anos 1990 (Jacobson, Booch & Rumbaugh, 1999), sendo o resultado de mais de 30 anos de experiência acumulada em projetos, notações e processos.

O UP é o primeiro modelo de processo inteiramente adaptado ao uso com a UML (*Unified Modeling Language*), desenvolvida pelo mesmo grupo. Sua concepção foi baseada nas práticas de maior retorno de investimento (ROI) do mercado.

As atividades do UP são bem definidas no seguinte sentido:

- a) Elas têm uma descrição clara e precisa.
- b) Apresentam responsáveis.
- c) Naselias são determinados artefatos de entrada e saída.
- d) Naselias são determinadas dependências entre as atividades.
- e) Seguem um modelo de ciclo de vida bem definido.
- f) São acompanhadas de uma descrição sistemática de como podem ser executadas com as ferramentas disponibilizadas (procedimentos).
- g) Preconizam o uso da linguagem UML.

As atividades incluem *workflows*, que são grafos que descrevem as dependências entre diferentes atividades. *Workflows* estão associados às disciplinas do Processo Unificado, que variam de implementação para implementação.

O RUP é considerado a principal implementação do Processo Unificado, pois foi definido pelos próprios criadores do UP. Mas existem outras implementações importantes, a maioria das quais varia na maneira como as diferentes disciplinas são definidas e organizadas. As implementações também podem variar na importância que dão a diferentes artefatos. As implementações ágeis do UP, por exemplo, simplificam as disciplinas e reduzem o número de artefatos esperados.

O número de implementações do UP é bastante grande, já que cada empresa pode implementar o modelo de acordo com suas necessidades. A Wikipédia¹ enumera várias implementações importantes, algumas das quais são citadas mais adiante neste capítulo.

5.1 Caracterização do Processo Unificado

O UP é mais do que um processo: é um *framework* extensível para a concepção de processos, podendo ser adaptado às características específicas de diferentes empresas e projetos. As principais características do UP são as seguintes:

- a) É dirigido por casos de uso.
- b) É centrado na arquitetura.
- c) É iterativo e incremental.
- d) É focado em riscos.

Essas características serão discutidas nas próximas subseções.

5.1.1 DIRIGIDO POR CASOS DE USO

O *caso de uso* é um processo compreendido do ponto de vista do usuário. Para o UP, o conjunto de casos de uso deve definir e esgotar *toda a funcionalidade possível* do sistema. Wazlawick (2011) apresenta várias técnicas para identificar casos de uso de boa granularidade.

Os casos de uso são úteis para várias atividades relacionadas ao desenvolvimento de um sistema, entre elas:

- a) *Definição e validação da arquitetura do sistema*: em geral, classes e atributos são obtidos a partir dos textos dos casos de uso expandidos.
- b) *Criação dos casos de teste*: os casos de uso podem ser vistos como um roteiro para o teste de sistema e de aceitação (Seção 13.2), em que as funcionalidades são testadas do ponto de vista do cliente.
- c) *Planejamento das iterações*: os casos de uso são priorizados e o esforço para desenvolvê-los é estimado de forma que cada iteração desenvolva certo número deles (Seção 6.4).
- d) *Base para a documentação do usuário*: os casos de uso são descrições de fluxos normais de operação de um sistema, bem como de fluxos alternativos representando o tratamento de possíveis exceções nos fluxos normais. Essas descrições são uma excelente base para iniciar o manual de operação do sistema, pois todas as funcionalidades possíveis estarão descritas aí de forma estruturada e completa.

Porém, a aplicação mais fundamental do caso de uso no desenvolvimento de sistemas é a incorporação dos requisitos funcionais de forma organizada. Cada passo dos fluxos principal e alternativos de um caso de uso corresponde a uma função do sistema. Requisitos não funcionais podem ser anotados juntamente com os casos de uso ou seus passos, e requisitos suplementares são anotados em um documento à parte.

5.1.2 CENTRADO NA ARQUITETURA

O UP preconiza que deve ser desenvolvida uma sólida arquitetura de sistema. As funcionalidades aprendidas com a elaboração dos diversos casos de uso devem ser integradas a essa arquitetura de forma incremental.

A arquitetura, inicialmente, pode ser compreendida como o conjunto de classes, possivelmente agrupadas em componentes, que realizam as operações definidas pelo sistema. A arquitetura é uma estrutura que provém funcionalidades. Os casos de uso são a descrição dos processos que realizam ou usam essas funcionalidades. Assim, a arquitetura existe para que as funcionalidades sejam possíveis.

A arquitetura é basicamente um modelo que define a estrutura da informação, suas possíveis operações e sua organização em componentes ou até mesmo em camadas.

Segundo o UP, a cada ciclo iterativo devem-se incorporar à arquitetura existente as funcionalidades aprendidas com a análise de cada um dos casos de uso abordados no ciclo. Assim, fazendo-se a priorização dos casos de uso

¹Disponível em: <en.wikipedia.org/wiki/IBM_Rational_Unified_Process>. Acesso em: 21 jan. 2013.

a partir dos mais críticos ou complexos para os mais triviais e simples, desenvolve-se, em um primeiro momento, todos os elementos de maior risco para a arquitetura, não deixando muitas surpresas para depois.

5.1.3 ITERATIVO E INCREMENTAL

Assim como nos métodos ágeis, o UP preconiza o desenvolvimento baseado em ciclos iterativos de duração fixa, em que, a cada iteração, a equipe incorpora à arquitetura as funcionalidades necessárias para realizar os casos de uso abordados no ciclo.

Cada ciclo iterativo produz um incremento no *design* do sistema, seja produzindo mais conhecimento sobre seus requisitos e arquitetura, seja produzindo um código executável. Espera-se que, em cada iteração, todas as disciplinas previstas sejam executadas com maior ou menor intensidade (ver. Figura 5.1). Então, assim como nos métodos ágeis, cada ciclo iterativo vai implicar executar todas as atividades usuais de desenvolvimento de software.

A integração contínua reduz riscos, facilita os testes e melhora o aprendizado da equipe sobre o sistema, especialmente nos primeiros momentos, quando decisões críticas precisam ser tomadas com pouco conhecimento sobre o sistema em si.

Em geral, a fase de concepção, por ser curta, é executada em um único ciclo. Já as fases de elaboração, construção e transição podem ser executadas a partir de uma série de ciclos iterativos. Porém, no caso de projetos muito grandes, a fase de concepção pode ser subdividida em ciclos nos quais se explorem diferentes características de um sistema.

5.1.4 FOCADO EM RISCOS

Em função das priorizações dos casos de uso mais críticos nos primeiros ciclos iterativos, pode-se dizer que o UP é focado em riscos. Se esses casos de uso são os que apresentam maior risco de desenvolvimento, então devem ser tratados o quanto antes para que esse risco seja resolvido enquanto o custo para tratá-lo ainda é baixo e o tempo disponível para lidar com as surpresas é relativamente grande.

Esse tipo de abordagem (tratar primeiro os problemas mais difíceis) tem sido um valor incorporado a vários modelos de desenvolvimento modernos. Os requisitos ou casos de uso de maior risco são os mais imprevisíveis. Assim, estudá-los primeiramente, além de garantir maior aprendizado sobre o sistema e decisões arquiteturais mais importantes, vai fazer que riscos positivos ou negativos sejam dominados o mais cedo possível (um risco positivo é, por exemplo, o sistema ser mais simples do que inicialmente imaginado).

5.2 Fases do Processo Unificado

O Processo Unificado divide-se em quatro grandes fases:

- a) *Concepção (inception)*: trata-se da elaboração de uma visão em abrangência do sistema. Nessa fase são levantados os principais requisitos, um modelo conceitual preliminar é construído, bem como são identificados os casos de uso de alto nível (Wazlawick, 2011), que implementam a funcionalidade requerida pelo cliente. Na fase de concepção calcula-se o esforço de desenvolvimento dos casos de uso e constrói-se o plano de desenvolvimento, composto por um conjunto de ciclos iterativos nos quais são acomodados os casos de uso. Pode haver alguma implementação e teste, caso seja necessário elaborar protótipos para redução de risco.
- b) *Elaboração (elaboration)*: nessa fase, as iterações têm como objetivo, predominantemente, detalhar a análise, expandindo os casos de uso, para obter sua descrição detalhada e situações excepcionais (fluxos alternativos). O modelo conceitual preliminar será transformado em um modelo definitivo, cada vez mais refinado, sobre o qual serão aplicados padrões de análise e uma descrição funcional poderá ser feita, bem como o *design* lógico e físico do sistema. Um código será gerado e testado. Contudo, essas atividades não são as que vão ocupar a maior parte do ciclo iterativo, pois haverá proporcionalmente mais carga de trabalho em análise e *design* do que em codificação e teste.
- c) *Construção (construction)*: a fase de construção possui iterações nas quais os casos de uso mais complexos já foram tratados e a arquitetura já foi estabilizada. Assim, as atividades de suas iterações consistem predominantemente na geração de código e testes do sistema. Com a automatização da geração de código e a introdução de modelos de desenvolvimento dirigidos a teste, pressupõe-se que um bom *design* possa dar origem rapidamente a um código de alta qualidade.

- d) *Transição (deployment)*: a fase de transição consiste na implementação do sistema no ambiente final, com a realização de testes de operação. Também é feita a transferência de dados de possíveis sistemas antigos para o novo sistema e o treinamento de usuários, bem como outras adaptações, que variam de caso para caso. Nessa fase pode haver ainda alguma revisão de requisitos e geração de código, mas não de forma significativa.

Apesar de as fases do Processo Unificado terem diferentes ênfases, espera-se que cada ciclo iterativo tome um conjunto de casos de uso e os desenvolva desde os requisitos até a implementação e a integração de código final. Na fase de elaboração a equipe necessariamente trabalhará mais tempo em questões de análise e projeto do que de implementação e teste. Na fase de construção, porém, os requisitos já terão sido em grande parte desvendados e o esforço recairá mais nas atividades de programação.

Uma das características do UP é o fato de que, a cada fase, um macro-objetivo (*milestone*) é atingido. Ao final da fase de concepção, o objetivo é ter entendido o escopo do projeto e planejado seu desenvolvimento. Ao final da fase de elaboração, os requisitos devem ter sido extensivamente compreendidos e uma arquitetura estável deve ter sido definida. Ao final da fase de construção, o sistema deve estar programado e testado. Ao final da fase de transição, o software deve estar instalado e sendo usado pelos usuários finais (West, 2002).

5.2.1 CONCEPÇÃO

No Processo Unificado, a fase de concepção não deve ser muito longa. Recomenda-se um período de duas semanas a dois meses, dependendo da dimensão relativa do projeto.

Nessa etapa os requisitos de projeto são analisados da melhor forma possível, em abrangência, e não em profundidade. É importante que o analista perceba claramente a diferença entre as necessidades lógicas e tecnológicas do cliente e os projetos de implementação que ele poderia fazer. A ideia é que o analista não polua a descrição dos requisitos com possibilidades tecnológicas de implementação que não foram expressamente requisitadas pelo cliente.

O UP espera que, nessa fase, sejam estabelecidos também os casos de uso. Há basicamente três formas de agir em relação a isso:

- a) Obter casos de uso a partir de uma organização dos requisitos funcionais, em que cada grupo de requisitos poderá dar origem a um ou mais casos.
- b) Inicialmente, proceder a uma análise de cenários e, posteriormente, extrair deles os casos de uso, ou seja, os requisitos.
- c) Trabalhar apenas com casos de uso, sendo eles a única expressão dos requisitos, não havendo outro documento de requisitos, exceto para os requisitos suplementares (Wazlawick, 2011).

De posse de um conjunto significativo de casos de uso, a equipe deve proceder como nas fases iniciais de planejamento do *Scrum* ou do *XP*, priorizando os casos de uso mais complexos e críticos em detrimento dos mais simples e triviais. Isso pode ser feito, inicialmente, por uma análise bem simples:

- a) Descubra os casos de uso que são meros relatórios, ou seja, que não alteram a informação armazenada, e classifique-os como os mais simples de todos.
- b) Verifique se existem casos de uso que seguem algum padrão conhecido, como CRUD (Wazlawick, 2011), mestre-detalhe etc. Esses casos serão de média complexidade, pois, embora possuam um comportamento bem conhecido e padronizado, podem esconder algumas surpresas nas suas regras de negócio, ou seja, inclusões, alterações e exclusões que só poderão ser feitas mediante determinadas condições (ver também Seção 7.5.9).
- c) Os demais casos de uso serão considerados os mais complexos. É recomendável que eles sejam organizados do mais importante para o menos importante em relação ao negócio da empresa. O analista deve se perguntar qual dos processos é o mais crítico para o sucesso da empresa e, em função disso, fazer a ordenação. Por exemplo, uma venda possivelmente será mais importante do que uma compra, uma compra mais importante do que uma reserva, uma reserva mais importante do que um cancelamento de reserva etc.

Na fase de concepção, a equipe deve produzir estimativas de esforço para casos de uso, que serão explicadas melhor na Seção 7.5. A partir desse cálculo, deve-se fazer um planejamento de longo prazo, procurando acomodar os casos de uso de acordo com sua prioridade nos diferentes ciclos durante o processo de desenvolvimento. Contudo, esse planejamento não deve ser muito detalhado.

Apenas o primeiro ciclo deve ter um planejamento mais detalhado, com atividades determinadas de acordo com o processo em uso e os tempos, responsáveis e recursos para a execução de cada atividade bem definidos. Os demais ciclos só deverão ter seu planejamento detalhado pouco antes de serem iniciados.

A fase de concepção envolve também o estudo de viabilidade, pois, ao final dela, analisadas as questões tecnológicas, de orçamento e de cronograma, a equipe deve decidir se é viável prosseguir com o projeto.

O marco final (*milestone*) da fase de concepção é conhecido como LCO, ou *Lifecycle Objective Milestone* (marco do ciclo de vida).

5.2.2 ELABORAÇÃO

A fase de elaboração consiste no detalhamento da análise e da realização do projeto para o sistema como um todo. A elaboração ocorre em ciclos, com partes de *design* sendo desenvolvidas e integradas ao longo de cada ciclo. Os principais objetivos dessa fase, segundo Ambler e Constantine (2000), são:

- a) Produzir uma arquitetura executável confiável para o sistema.
- b) Desenvolver o modelo de requisitos até completar pelo menos 80% dele.
- c) Desenvolver um projeto geral para a fase de construção.
- d) Garantir que as ferramentas críticas, processos, padrões e regras estejam disponíveis para a fase de construção.
- e) Entender e eliminar os riscos de alta prioridade do projeto.

Essa fase permite analisar o domínio do problema de forma mais refinada e definir uma arquitetura mais adequada e sólida. Além disso, a priorização dos casos de uso mais complexos permitirá eliminar ou mitigar os elementos do projeto que apresentam maior risco.

Assim, embora o Processo Unificado também trabalhe com a perspectiva de acomodação de mudanças, procura minimizar seu impacto mitigando riscos e elaborando uma arquitetura o mais próximo possível do necessário para que as funcionalidades requeridas possam ser desenvolvidas.

A fase de elaboração é caracterizada pela exploração dos casos de uso mais complexos, que vão precisar de mais trabalho de análise do que de implementação, já que será necessário entender e modelar seu funcionamento. À medida que esses casos de uso são estudados e desenvolvidos, a arquitetura do sistema vai se formando. Espera-se que a arquitetura esteja estável no momento em que se passar a considerar apenas os casos de uso padronizados, como CRUD e relatórios. Esses casos de uso não devem impactar a arquitetura.

O marco final (*milestone*) da fase de elaboração é conhecido como LCA, ou *Lifecycle Architecture Milestone* (marco da arquitetura).

5.2.3 CONSTRUÇÃO

Na fase de construção, um produto completo e usável deve estar desenvolvido, testado e adequado para uso pelo usuário final. Essa fase é realizada em ciclos iterativos e o projeto é desenvolvido também de forma incremental, com novas funcionalidades sendo adicionadas ao sistema a cada ciclo.

Segundo Ambler e Constantine (2000a), os principais objetivos da fase de construção são:

- a) Descrever os requisitos que ainda faltam.
- b) Dar substância ao *design* do sistema.
- c) Garantir que o sistema atenda às necessidades dos usuários e que ele se encaixe no contexto geral da organização.
- d) Completar o desenvolvimento dos componentes e testá-los, incluindo tanto o software quanto sua documentação.
- e) Minimizar os custos de desenvolvimento pela otimização dos recursos.
- f) Obter a qualidade adequada o mais rápido possível.
- g) Desenvolver versões úteis do sistema.

A fase de construção caracteriza-se pela exploração dos casos de uso de baixa e média complexidade, ou seja, os casos de uso padronizados que não vão impactar na arquitetura. Como esses casos de uso são padronizados, o esforço de análise e *design* será menor nessa fase, ficando a maior parte do trabalho concentrada na implementação e teste dos componentes da arquitetura dedicados a esses casos de uso.

O marco final (*milestone*) da fase de construção é conhecido como IOC, ou *Initial Operational Capability Milestone* (marco da capacidade operacional inicial).

5.2.4 TRANSIÇÃO

A fase de transição consiste na colocação do sistema em uso no ambiente final. São necessários testes de aceitação e operação, treinamento de usuários e transição de dados a partir de sistemas antigos, que podem ser capturados automaticamente ou digitados.

Nessa fase, também poderá haver a execução do sistema em paralelo com sistemas legados para verificar sua adequação.

Após a conclusão da fase de transição, o sistema entra em evolução, ou seja, depois de aceito e colocado em operação no ambiente final, ele passa a receber atualizações periódicas de forma a corrigir possíveis erros ou implementar novas funcionalidades necessárias (ver Capítulo 14).

O marco final (*milestone*) da fase de transição é conhecido como PR, ou *Product Release Milestone* (marco da entrega do produto).

5.3 RUP – Rational Unified Process

A mais detalhada e mais antiga implementação do UP é conhecida como RUP (*Rational Unified Process*²), criada por Booch, Rumbaugh e Jacobson através da empresa Rational, subsidiária da IBM desde 2003. A versão RUP é tão importante que algumas vezes é confundida ou considerada sinônima de UP.

Ainda antes de pertencer à IBM, a empresa Rational adquiriu, em 1997, várias outras empresas: Verdix, Objec-tory, Requisite, SQA, Performance Awareness e Pure-Atria. A partir da experiência acumulada dessas empresas, estabeleceu algumas práticas, que seriam a base filosófica para o novo modelo de processo, mais tarde conhecido como RUP (Kruchten, 2003):

- a) *Desenvolver iterativamente tendo o risco como principal fator de determinação de iterações:* é preferível conhecer todos os requisitos antes de se iniciar o desenvolvimento propriamente dito, mas em geral isso não é viável, de forma que o desenvolvimento iterativo orientado à redução de risco é bastante adequado.
- b) *Gerenciar requisitos:* deve-se manter controle sobre o grau de incorporação dos requisitos do cliente ao produto.
- c) *Empregar uma arquitetura baseada em componentes:* quebrar um sistema complexo em componentes não só é necessário como inevitável. A organização permite diminuir o acoplamento, o que possibilita testes mais confiáveis e maior possibilidade de reuso.
- d) *Modelar software de forma visual com diagramas:* UML é indicada como padrão de modelagem de diagramas.
- e) *Verificar a qualidade de forma contínua:* o processo deve ser o mais orientado a testes quanto for possível. Se for o caso, utilizam-se técnicas de desenvolvimento orientado a testes, como TDD, ou *Test-Driven Development* (Beck, 2003).
- f) *Controlar as mudanças:* a integração deve ser contínua e gerenciada adequadamente com o uso de um sistema de gerenciamento de configuração (Capítulo 10).

O RUP é um produto que, entre outras coisas, inclui uma base de dados com *hiperlinks* com vários artefatos e *templates* necessários para usar bem o modelo. Uma descrição do processo em português pode ser encontrada na internet.³

5.3.1 Os Blocos de Construção do RUP

O RUP é baseado em um conjunto de elementos básicos (*building blocks*) identificados da seguinte forma:

- a) *Quem:* um *papel* (Seção 2.3.2) define um conjunto de habilidades necessário para realizar determinadas atividades.

²Disponível em: <www.rational.com/rup/>. Acesso em: 21 jan. 2013.

³Disponível em: <www.wthreex.com/rup/portugues/index.htm>. Acesso em: 21 jan. 2013.

- b) *O quê:* o *produto do trabalho* (*work product*) define algo produzido por alguma atividade, como diagramas, relatórios ou código funcionando, ou seja, os *artefatos* (Seção 2.3.1).
- c) *Como:* uma *atividade* (Seção 2.3) descreve uma unidade de trabalho atribuída a um papel que produz determinado conjunto de artefatos.
- d) *Quando:* os *workflows* são grafos que definem as dependências entre as diferentes atividades.

As disciplinas RUP, como requisitos, análise e *design*, implementação etc., são os contêineres para os quatro elementos mencionados, ou seja, cada disciplina é determinada a partir de um ou mais *workflows*, que estabelecem dependências entre as atividades, as quais são realizadas por pessoas representando papéis e produzindo ou transformando artefatos bem definidos.

5.3.1.1 Papéis

Segundo Kruchten (2003), um *papel* define um conjunto de comportamentos e responsabilidades para uma pessoa ou grupo de pessoas que trabalham como uma equipe. O comportamento é expresso através de um conjunto de atividades exercidas por esse papel, as quais devem ser coesas. Essas responsabilidades e atividades também são expressas em função dos artefatos que esses papéis criam ou alteram.

Papéis não são pessoas específicas nem cargos. A mesma pessoa pode exercer vários papéis em diferentes momentos, num mesmo dia, no mesmo projeto.

Os papéis são organizados em cinco categorias principais:

- a) Papéis de analista.
- b) Papéis de desenvolvedor.
- c) Papéis de testador.
- d) Papéis de gerente.
- e) Outros papéis.

As subseções a seguir apresentarão alguns dos papéis mais comuns incluídos nesses cinco grupos.

5.3.1.1.1 Papéis de Analista

Esses papéis estão relacionados principalmente ao contato com o futuro usuário ou cliente do sistema. As pessoas que os representam devem ser capazes de entender quais são as necessidades do sistema e criar descrições que sejam compreensíveis para os *designers*, desenvolvedores e testadores. Além de criarem essas descrições, devem garantir sua qualidade e especialmente sua adequação às reais necessidades e conformidade com normas e padrões estabelecidos.

Os papéis de analista no RUP subdividem-se nos seguintes tipos:

- a) *Analista de sistemas:* lidera e coordena a análise de requisitos, definindo o escopo do sistema e seus casos de uso. O analista de sistemas deve ser uma pessoa com boa capacidade de comunicação e negociação, além de ter conhecimento de negócios e tecnologia.
- b) *Designer de negócio:* faz a modelagem de negócio, em geral utilizando diagramas de atividade, máquina de estados ou BPMN. Ele atuará nas fases mais iniciais da análise com a produção do modelo de negócio ou ciclo de negócio, que vai colocar o sistema a ser desenvolvido dentro da perspectiva mais ampla da organização. A partir das atividades de negócio identificadas, os casos de uso serão identificados.
- c) *Revisor do modelo de negócios:* deve revisar o modelo de negócio, verificando se é consistente, coerente e não ambíguo. Deve ter conhecimento profundo do negócio e, se possível, da tecnologia a ser usada para sua implementação (o papel poderá ser desempenhado por duas pessoas, caso não se consiga uma única pessoa com essas características).
- d) *Analista do processo de negócio:* lidera e coordena a modelagem da organização como casos de uso de negócio (Jacobson, 1994). Essa atividade pode ser dispensada, especialmente se não houver necessidade de uma reengenharia de negócio a partir do projeto que está sendo desenvolvido.
- e) *Especificador de requisitos:* é responsável pelos requisitos que costumam ser representados como casos de uso, inicialmente de alto nível e depois expandidos na sua forma essencial. Além dos requisitos funcionais, representados nos casos de uso, o especificador de requisitos deve especificar os requisitos suplementares, ou seja, os aspectos não funcionais gerais do sistema.

- f) *Revisor de requisitos*: é responsável pela revisão minuciosa dos casos de uso e especificações suplementares, verificando se são completos, coerentes e não ambíguos. Além disso, deve garantir que o documento de casos de uso esteja escrito em conformidade com os padrões adotados na empresa.
- g) *Designer de interface com usuário*: coordena as atividades de prototipação de interfaces e de *design* de interface com usuário. Ele é o responsável pelos requisitos de interface. Não implementa a interface, e sua responsabilidade fica restrita ao *design* visual e de usabilidade.

O RUP ainda considera que o *analista de teste* é um papel de analista. Porém, uma vez que existem papéis relacionados à disciplina de teste, pode-se considerar que sua atividade é mais fortemente relacionada a esses papéis, e não aos de analista de sistemas. Dessa forma, neste livro ele foi classificado com os papéis de teste.

5.3.1.1.2 Papéis de Desenvolvedor

Os desenvolvedores são aqueles que transformam os requisitos em produto. Os papéis relacionados ao desenvolvedor, no RUP, são os seguintes:

- a) *Implementador*: é responsável pela produção e teste de unidade do código fonte.
- b) *Revisor de código*: deve garantir a qualidade do código-fonte implementado, verificando se este segue os padrões estabelecidos e as boas práticas de programação. O revisor deve ser um profundo conhecedor da linguagem de programação utilizada.
- c) *Integrador*: quando os implementadores liberam componentes já testados, cabe ao integrador incluí-los em uma versão operacional do sistema para gerar uma nova versão (ou *build*). O integrador deverá elaborar e realizar os testes de integração caso essa atividade não tenha sido atribuída ao analista de teste.
- d) *Arquiteto de software*: é responsável pelo *design* das camadas e/ou partições do sistema, ou seja, da sua estrutura em nível mais alto, incluindo o *design* de pacotes, componentes e sua distribuição em diferentes nodos de processamento.
- e) *Revisor de arquitetura*: deve avaliar se o trabalho do arquiteto de software leva a uma arquitetura sólida e estável. Como a arquitetura costuma conter as decisões mais cruciais para o sucesso de um sistema, esse papel é importante para garantir que as decisões do arquiteto sejam as mais efetivas possíveis.
- f) *Designer*: toma decisões sobre o *design* das classes, seus atributos e associações, bem como sobre os métodos a serem implementados.
- g) *Revisor de design*: deve revisar o *design*, verificando se os melhores padrões e práticas foram adotados.
- h) *Designer de banco de dados*: é responsável pelo *design* das tabelas e códigos associados ao banco de dados.
- i) *Designer de cápsula*: este papel só existe em projetos de sistemas de tempo real. Ele é responsável por assegurar que o sistema responda prontamente a eventos de acordo com os requisitos de tempo real.

O RUP inclui também o papel de *designer de teste*, o qual, neste livro, foi agrupado com os papéis de testador.

5.3.1.1.3 Papéis de Testador

Embora no RUP originalmente exista apenas um papel específico para o testador, foram agrupados aqui também os papéis de analista, *designer* e gerente de teste.

- a) *Designer de teste*: é responsável por definir as técnicas e estratégias de teste a serem adotadas. Não deve ser confundido com o *analista de teste*, que elabora os testes de sistema. O *designer de teste* define a estratégia de teste, mas não os testes reais.
- b) *Analista de teste*: é responsável pelo projeto e pela elaboração dos casos de teste a serem aplicados ao sistema. Em geral, o teste de unidade é feito pelo próprio programador e o teste de integração pelo integrador, e o analista de teste costuma atuar principalmente no teste de sistema.
- c) *Testador*: é responsável pela realização efetiva dos testes. Ele verifica a melhor abordagem a ser utilizada de acordo com as estratégias definidas pelo *designer de teste* e pelo analista de teste, implementando o teste, além de executá-lo, para verificar se o componente ou o sistema são aprovados. Ele deve registrar os resultados dos testes e, no caso de componentes que não passarem nos testes, informar os respectivos responsáveis para que providenciem sua correção.
- d) *Gerente de teste*: coordena toda a atividade de teste do sistema.

5.3.1.1.4 Papéis de Gerente

Os papéis de gerente estão relacionados especialmente às atividades de planejamento, controle e organização do projeto. Os papéis de gerente no RUP são os seguintes:

- a) *Engenheiro de processo*: é o responsável pela aplicação do processo de desenvolvimento, devendo configurar e ajustar o processo de acordo com as necessidades da equipe e dos projetos.
- b) *Gerente de projeto*: é o responsável por um ou mais projetos específicos, devendo planejar as atividades e alocar os recursos físicos e humanos, bem como acompanhar o projeto, garantindo que prazos e orçamentos sejam cumpridos e tomando decisões de correção de rumo quando necessário.
- c) *Gerente de controle de mudança*: é o responsável pelo controle das requisições de mudança, seja do cliente, seja da própria equipe de desenvolvimento, e por acompanhar o atendimento a elas.
- d) *Gerente de configuração*: planeja e disponibiliza o ambiente para que os desenvolvedores e integradores possam realizar suas atividades. Ele garante que todos tenham acesso aos artefatos necessários.
- e) *Gerente de implementação*: planeja e acompanha a implementação do sistema junto com os clientes e usuários.
- f) *Revisor do projeto*: é responsável pela revisão dos planos e avaliações do projeto ao longo do seu desenvolvimento.

Além desses papéis, há o gerente de teste, agrupado com os papéis de testador.

5.3.1.1.5 Outros Papéis

Os outros papéis do RUP, não classificados nos tipos anteriores, são os seguintes:

- a) *Interessados (ou envolvidos)*: são todos aqueles afetados pelo sistema ou seu desenvolvimento.
- b) *Desenvolvedor de curso*: é o responsável pela criação de material de treinamento para usuários.
- c) *Artista gráfico*: é o responsável pela criação da arte final do produto, de sua embalagem e de outros artefatos correlatos.
- d) *Especialista em ferramentas*: dá suporte aos desenvolvedores pela seleção, instalação e treinamento no uso das ferramentas de apoio ao desenvolvimento e gerenciamento dos projetos.
- e) *Administrador do sistema*: mantém o ambiente de desenvolvimento, cuidando do hardware, de versões de software, rede etc.
- f) *Redator técnico*: é o responsável pela redação final tanto dos manuais quanto de partes do sistema orientadas a texto.

Como o RUP é adaptável, novos papéis podem ser necessários e adicionados a um projeto ou processo específico de empresa.

5.3.1.2 Atividades

Atividades são unidades de trabalho executadas por um indivíduo que exerce um papel dentro do processo (Kruchten, 2003). Toda atividade deve produzir um resultado palpável em termos de criação ou alteração consistente de artefatos, como modelos, elementos (classes, atores, código etc.) ou planos. Em RUP, toda atividade é atribuída a um papel específico.

Assim como nos métodos ágeis, uma atividade não deve ser nem muito curta (durar poucas horas) nem muito longa (durar vários dias). Sugere-se pensar em atividades que possam ser realizadas em períodos de 1 a 3 dias, porque essa duração facilita o seu acompanhamento.

A mesma atividade pode ser repetida sobre o mesmo artefato, o que é normal ao longo dos ciclos iterativos. Contudo, não é repetida necessariamente pela mesma pessoa, embora possa ser pelo mesmo papel. Por exemplo, a arquitetura do sistema pode ser refinada e revisada diversas vezes ao longo dos ciclos iterativos.

Atividades são normalmente descritas em termos de passos. Kruchten (2003) identifica três tipos de passos:

- a) *De pensamento*: a pessoa que exerce o papel comprehende a natureza da atividade, obtém e examina os artefatos de entrada e formula a saída.
- b) *De realização*: a pessoa que exerce o papel cria ou atualiza artefatos.
- c) *De revisão*: a pessoa que exerce o papel inspeciona os resultados em função de algum critério.

Assim como os passos de casos de uso expandidos (Wazlawick, 2011), nem sempre os passos de atividades são executados. Deve existir um conjunto de passos que consista no fluxo principal da atividade, sendo que, nesse caso,

todos são obrigatórios. Também podem existir passos que pertençam a fluxos alternativos, sejam variantes, sejam tratadores de exceção.

5.3.1.2.1 Artefatos

Um artefato pode ser um diagrama, modelo, elemento de modelo, texto, código-fonte, código executável etc., ou seja, qualquer tipo de produto criado ao longo do processo de desenvolvimento de software.

Assim, artefatos podem ser compostos por outros artefatos, como uma classe contida no modelo conceitual, por exemplo. Como os artefatos mudam com o passar do tempo, pode ser interessante submetê-los a um controle de versões. Porém, normalmente esse controle é exercido no artefato de mais alto nível, e não em elementos individuais.

Os artefatos são as entradas e também as saídas para as atividades. Em geral, o objetivo de uma atividade é criar artefatos ou promover alterações consistentes e verificáveis em artefatos.

Artefatos de saída ainda podem ser classificados em *entregas*, que são artefatos entregues ao cliente.

Artefatos (especialmente os de texto) também podem ser definidos por *templates*, isto é, modelos de documentos que dão forma geral ao artefato.

Do ponto de vista do Processo Unificado, o artefato não deve ser entendido simplesmente como um documento acabado no estilo dos relatórios de revisão final dos marcos do Modelo Cascata. Os artefatos no UP e, consequentemente, no RUP são documentos dinâmicos que podem ser alterados a qualquer momento e, por isso, são mantidos sob controle de versão.

O RUP não encoraja a produção de papel. Sugere-se que todos os artefatos sejam mantidos e gerenciados por uma ferramenta adequada, de forma que sua localização e versão corrente sempre sejam conhecidas e acessíveis por quem de direito.

Ao contrário do XP, que incentiva a posse coletiva, RUP sugere que cada artefato tenha um dono ou responsável. Embora outros tenham acesso, apenas o responsável pode alterar um artefato ou conceder a outros o direito de fazer alterações.

5.3.1.2.2 Relatórios

O relatório é uma visão gerada para um artefato ou conjunto de artefatos com o propósito de servir para revisão.

Ao contrário dos artefatos, os relatórios não são sujeitos a controles de versão, porque devem poder ser gerados a qualquer tempo (de preferência, automaticamente) de acordo com a versão atual desses artefatos.

5.3.1.2.3 Grupos de Artefatos

No RUP os artefatos estão organizados em grupos de acordo com as disciplinas (Seção 5.3.2) às quais se relacionam. Basicamente, são os seguintes tipos:

- a) *Artefatos de gerenciamento*: estão relacionados ao planejamento e à execução do projeto em si. Exemplos: plano de desenvolvimento de software, caso de negócio, plano de iteração, avaliação de iteração, avaliação de *status*, plano de resolução de problemas, plano de gerenciamento de riscos, lista de riscos, ordem de trabalho, plano de aceitação do produto, plano de métricas, plano de garantia de qualidade, lista de problemas, registro de revisão e métricas de projeto.
- b) *Artefatos de gerenciamento de configuração e mudança*: apresentam informações relacionadas à disciplina de gerenciamento de configuração e mudança. Exemplos: registro de auditoria de configuração, plano de gerenciamento de configuração, repositório do projeto, espaço de trabalho de desenvolvimento, espaço de trabalho de integração e solicitação de mudança.
- c) *Artefatos de ambiente*: são usados ao longo do processo de desenvolvimento para garantir a consistência dos demais artefatos. Exemplos: caso de desenvolvimento, avaliação da organização de desenvolvimento, *templates* específicos do projeto, guia de modelagem de negócio, guia de *design*, guia de programação, guia de modelagem de casos de uso, guia de interface de usuário, guia de teste, manual de guia de estilo, guia de ferramentas, ferramentas e infraestrutura de desenvolvimento.
- d) *Artefatos de modelo de negócio*: capturam e apresentam o modelo de negócio da empresa, ou seja, o contexto no qual o sistema deverá funcionar. Exemplos: glossário de negócios, regras de negócios, modelo de casos de uso de negócios, modelo de objetos de negócios, avaliação da organização-alvo, visão de negócio, arquitetura de negócio e especificação suplementar de negócio.

- e) *Artefatos de requisitos*: estão relacionados à descrição do sistema a ser desenvolvido. Exemplos: plano de gerenciamento de requisitos, solicitações dos interessados, glossário, visão, modelo de casos de uso, especificações suplementares, atributos de requisitos, protótipos de interfaces e cenários de casos de uso.
- f) *Artefatos de design*: capturam e apresentam a solução tecnológica para atender aos requisitos dos interessados. Exemplos: prova de conceito arquitetural, arquitetura do software, modelo de análise, modelo de *design*, modelo de dados e modelo de implantação.
- g) *Artefatos de implementação*: capturam e realizam a solução tecnológica em um conjunto de artefatos executáveis. Exemplos: componentes, versão do sistema (*build*), plano de integração da versão, modelo de implementação.
- h) *Artefatos de teste*: são os planos e produtos das atividades de teste. Exemplos: plano de teste, sumário de avaliação de teste, *script* de teste, *log* de teste, lista de ideias de teste, casos de teste, modelo de análise de carga de trabalho, dados de teste, resultados de teste, arquitetura para automatização de teste, especificação da interface de teste, configuração do ambiente de teste, conjunto de testes, guia de teste, classe de teste e componente de teste.
- i) *Artefatos de implantação*: consistem na informação final entregue ao cliente, bem como em outros artefatos ligados à transição do sistema. Exemplos: plano de implantação, lista de materiais, notas de *release*, produto, artefatos de instalação, materiais de treinamento, unidade de implantação, arte-final do produto e material de suporte para o usuário.

Os artefatos não são produzidos de forma exclusiva em uma ou outra fase do RUP, já que este é um processo iterativo. O que se espera é que a maioria dos artefatos seja iniciada na fase de concepção e refinada ao longo do projeto, sendo finalizada na fase de implantação. Também se espera que os artefatos iniciais, como requisitos e *design*, sejam finalizados mais cedo do que os artefatos finais, como os de implementação e de implantação.

5.3.1.3 Workflows

As atividades a serem executadas dentro de cada disciplina são definidas a partir de grafos direcionados chamados de *workflows*. Um *workflow* define um conjunto de atividades e um conjunto de papéis responsáveis por uma atividade. Além disso, o *workflow* indica as dependências entre as diferentes atividades, ou seja, quais atividades dependem logicamente de outras atividades para poderem ser executadas. Essa dependência pode ocorrer em diferentes níveis de intensidade, sendo, porém, algumas absolutamente necessárias e outras meramente sugeridas.

O RUP define três tipos de *workflow*:

- a) *Workflow núcleo (core)*: define a forma geral de condução de uma dada disciplina.
- b) *Workflow detalhe*: apresenta um refinamento do *workflow* núcleo, indicando atividades em um nível mais detalhado, bem como artefatos de entrada e saída de cada atividade.
- c) *Planos de iteração*: consistem em uma instanciação do processo para uma iteração específica. Embora o RUP tenha uma descrição geral dos *workflows* para cada atividade, elas costumam ocorrer de forma diferente em projetos diferentes e até mesmo em ciclos diferentes dentro do mesmo projeto. Assim, o plano de iteração consiste em especificar atividades concretas a serem realizadas de fato dentro de uma iteração planejada.

5.3.1.4 Outros Elementos

Além dos elementos mencionados anteriormente, o RUP apresenta outros elementos que auxiliam na aplicação do processo a um projeto. Esses elementos são:

- a) *Procedimentos (guidelines)*: as atividades são apresentadas nos *workflows* de forma mnemônica, mas para servir de lembrança do que para orientar. Mas os procedimentos mostram um detalhamento dessas atividades para que não apenas pessoas acostumadas ao processo, mas também os novatos possam saber o que fazer. No RUP, os procedimentos são basicamente regras, recomendações e heurísticas sobre como executar a atividade. Elas devem focar também a descrição dos atributos de qualidade dos artefatos a serem feitos, como o que é um bom caso de uso, uma classe coesa etc.
- b) *Templates*: são modelos ou protótipos de artefatos e podem ser usados para criar os respectivos artefatos. Devem estar disponíveis na ferramenta usada para criar e gerenciar o artefato. Exemplos de *templates* são

os modelos de documento do Microsoft Word e os próprios *workflows*, especificados em uma ferramenta adequada, que podem ser usados como modelos para a criação dos planos de iteração.

- c) *Mentores de ferramenta*: consistem em uma descrição detalhada de como realizar uma atividade em uma ferramenta específica. Enquanto as descrições das atividades e até mesmo os procedimentos no RUP devem ser razoavelmente independentes de tecnologia para que possam ser interpretados em diferentes ferramentas, os mentores devem ser preferencialmente construídos como um tutorial na própria ferramenta, mostrando como usá-la.

5.3.2 DISCIPLINAS

Como foi visto, o UP preconiza que diferentes disciplinas sejam definidas, cada qual descrevendo uma possível abordagem ao problema de gerenciar o desenvolvimento de um sistema. As disciplinas do UP englobam diferentes atividades e papéis relacionados por área de especialidade, e suas implementações variam de acordo com o número e a descrição dessas disciplinas.

Particularmente, o RUP conta com seis disciplinas de projeto e três disciplinas de suporte. As disciplinas de projeto são as seguintes:

- a) Modelagem de negócio.
- b) Requisitos.
- c) Análise e *design*.
- d) Implementação.
- e) Teste.
- f) Implantação.

E as disciplinas de suporte são as seguintes:

- a) Gerenciamento de mudança e configuração.
- b) Gerenciamento de projeto.
- c) Ambiente.

Cada uma dessas disciplinas aparece com uma ênfase diferente ao longo das fases e dos ciclos iterativos no RUP. A Figura 5.1 apresenta graficamente as ênfases ao longo do ciclo de vida RUP.

A Figura 5.1 é também chamada de *Arquitetura do RUP*. Embora os nomes das disciplinas até lembrem as fases do Modelo Cascata, elas não são executadas de forma sequencial, visto que o RUP é um processo iterativo: suas quatro fases são sequenciais, mas as disciplinas são executadas de forma paralela ou sequencial, dependendo do caso, dentro de cada iteração individual.

5.3.2.1 Gerenciamento de Projeto

Segundo Kruchten (2003), gerenciar um projeto consiste em balancear objetivos que competem entre si, gerenciar riscos e superar restrições com o objetivo de obter um produto que atenda às necessidades dos clientes (que pagam pelo desenvolvimento) e dos usuários finais.

Os objetivos da disciplina de gerenciamento de projeto são indicar como planejar o projeto como um todo (Seção 6.4), como planejar cada iteração individual (Seção 6.5), como gerenciar os riscos do projeto (Capítulo 8) e como monitorar o progresso (Capítulo 9). Entretanto, o RUP não trata os seguintes aspectos:

- a) Gerenciamento de pessoas, incluindo contratação e treinamento.
- b) Gerenciamento de orçamento.
- c) Gerenciamento de contratos.

Tais aspectos são cobertos por uma implementação de RUP, a extensão EUP (Seção 5.6).

A disciplina de gerenciamento de projeto produz planos. No RUP, o planejamento ocorre em dois níveis: plano de fase (ou projeto) e planos de iteração.

O *plano de fase* ou *plano de projeto* procura delinear o tempo total de desenvolvimento, duração e esforço das fases, número de ciclos e objetivos gerais de cada ciclo. Ele é controlado e mensurado pelos seguintes componentes:

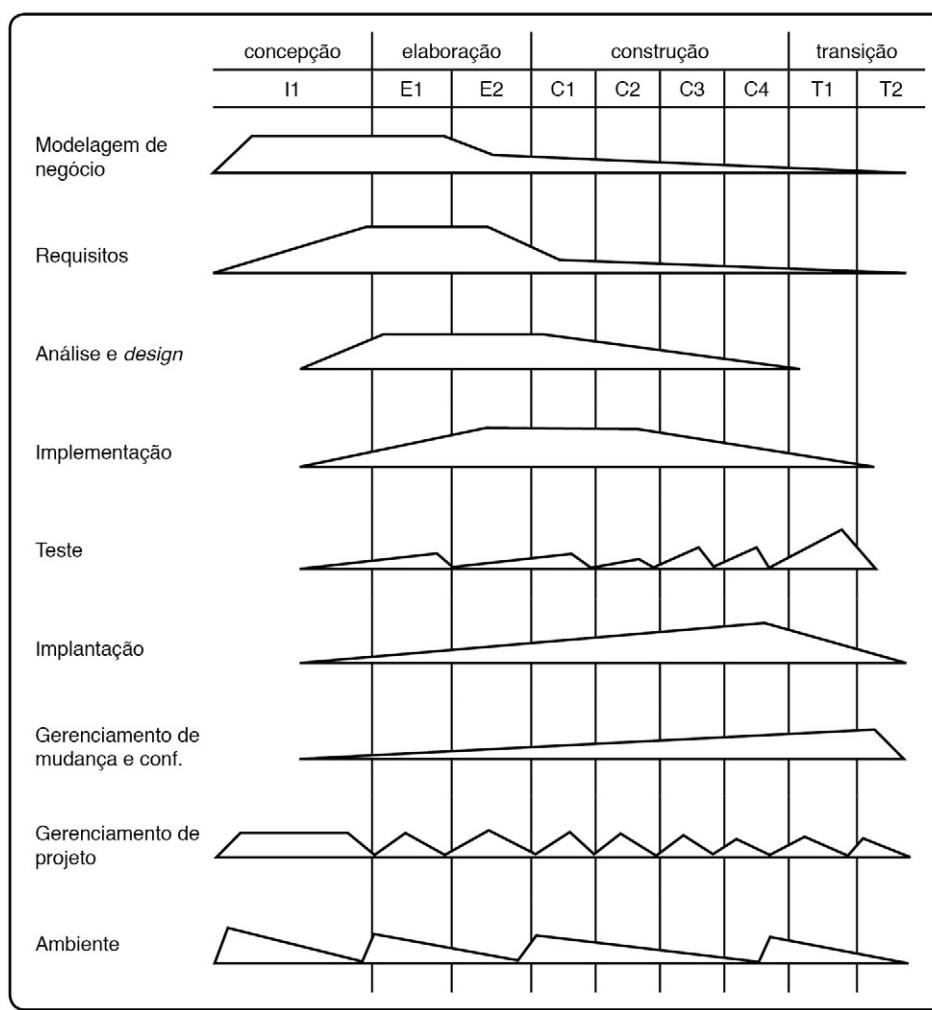


Figura 5.1 Diferentes ênfases de cada disciplina nas diferentes fases do RUP⁴.

- a) *Plano de medição (measurement plan)*: estabelece as métricas a serem usadas para definir o andamento do projeto ao longo de sua execução (Seção 9.5).
- b) *Plano de gerenciamento de riscos*: detalha como os riscos serão tratados ao longo do projeto, criando atividades de mitigação e monitoramento de riscos e atribuindo responsabilidades quando necessário.
- c) *Lista de riscos*: lista ordenada dos riscos conhecidos e ainda não resolvidos, em ordem decrescente de importância, juntamente com planos de mitigação e contingência quando for o caso.
- d) *Plano de resolução de problemas*: descreve como problemas identificados ao longo do projeto devem ser reportados, analisados e resolvidos (Seção 9.4.2).
- e) *Plano de aceitação de produto*: descreve como o produto será avaliado pelo cliente para verificar se satisfaz suas necessidades. O plano deve incluir a identificação dos testes de aceitação.

O plano de fase não precisa ser longo (Seção 6.4). Em geral, algumas poucas páginas são suficientes para a maioria dos projetos. Ele deve fazer referência ao documento de visão geral do sistema para esclarecimento sobre os objetivos do sistema e seu contexto.

Os *planos de iteração* (Seção 6.5) são mais detalhados do que o plano de fase. Cada plano de iteração inclui um cronograma de atividades atribuídas a responsáveis, com recursos alocados, prazos e dependências.

Em geral, há um plano de iteração sendo seguido enquanto o plano da iteração seguinte vai sendo delineado para ser finalizado ao final do ciclo corrente.

⁴Ambler (2005). Disponível em: <www.amblysoft.com/downloads/managersIntroToRUP.pdf>. Acesso em: 21 jan. 2013.

Para definir um plano de iteração, sugere-se observar os seguintes elementos:

- a) O *plano da fase* corrente incluído no plano de projeto.
- b) Informação sobre o *status do projeto* (por exemplo, atrasado, em dia, com grandes riscos em aberto etc.).
- c) Uma *lista de casos de uso*, que, pelo plano da fase, devem ser finalizados na iteração corrente.
- d) Uma *lista dos riscos* que devem ser tratados na iteração corrente.
- e) Uma *lista das modificações* que devem ser incorporadas ao produto na iteração corrente (defeitos e erros encontrados ou mudanças de funcionalidade).

Todas as listas mencionadas devem ser ordenadas da mais importante para a menos importante, de forma que, se a iteração for muito mais trabalhosa do que o esperado, os itens menos importantes possam ser deixados para iterações posteriores.

O plano de iteração costuma ser apresentado como um diagrama Gantt (Seção 6.5.6) e deve indicar, para cada membro da equipe, as atividades em que estará envolvido, seu início e final. Os aspectos de planejamento de projeto, medição e gerenciamento de riscos serão abordados respectivamente nos Capítulos 6, 7 e 8.

De acordo com RUP, o principal papel da disciplina de gerenciamento de projeto é o do gerente de projeto, que é responsável pelos seguintes artefatos:

- a) Caso de negócio (*business case*).
- b) Plano de desenvolvimento de software, incluindo:
 - Plano de aceitação de produto.
 - Plano de gerenciamento de riscos.
 - Lista de riscos.
 - Plano de resolução de problemas.
 - Plano de medição.
- c) Plano da iteração.
- d) Avaliação da iteração.
- e) Ordem de serviço.
- f) Avaliação de *status*.
- g) Medidas de projeto.

Além dele, outro papel importante é o de revisor de projeto, conforme será visto no detalhamento do *workflow* a seguir. O *workflow* da disciplina de gerenciamento de projeto é definido conforme a Figura 5.2.

Esse *workflow* é decomposto em uma série de detalhamentos, ou seja, cada uma das atividades é descrita por atividades ainda mais elementares, conforme a Tabela 5.1.

As subatividades de identificação e avaliação de riscos e desenvolvimento do caso de negócio aparecem repetidas nas atividades de concepção de novo projeto e avaliação de escopo e riscos do projeto, porque, quando forem executadas durante a avaliação de escopo e riscos, elas o serão na forma de uma revisão, realizada após o início e a aprovação do projeto, e também ao final das iterações.

A disciplina de planejamento de projeto, e especificamente as atividades de construção de plano de projeto e plano de iteração, será detalhada no Capítulo 6.

5.3.2.2 Modelagem de Negócio

A modelagem de negócio consiste em estudar e compreender a empresa e seus processos, visto que o sistema a ser desenvolvido não será um produto isolado, mas parte orgânica do funcionamento dessa empresa. Os objetivos dessa disciplina são os seguintes:

- a) Entender a estrutura e a dinâmica da organização-alvo na qual o software será utilizado.
- b) Entender os problemas atuais na organização-alvo e identificar potenciais melhorias que podem ser produzidas com o software.
- c) Certificar-se de que clientes, usuários e desenvolvedores tenham um conhecimento comum da organização-alvo.
- d) Derivar os requisitos para dar suporte a essas melhorias.

Um dos pontos críticos para o sucesso de um projeto de desenvolvimento de software é o seu financiamento. Para que o cliente se mantenha interessado no desenvolvimento de um produto, ele deve estar sempre convencido de

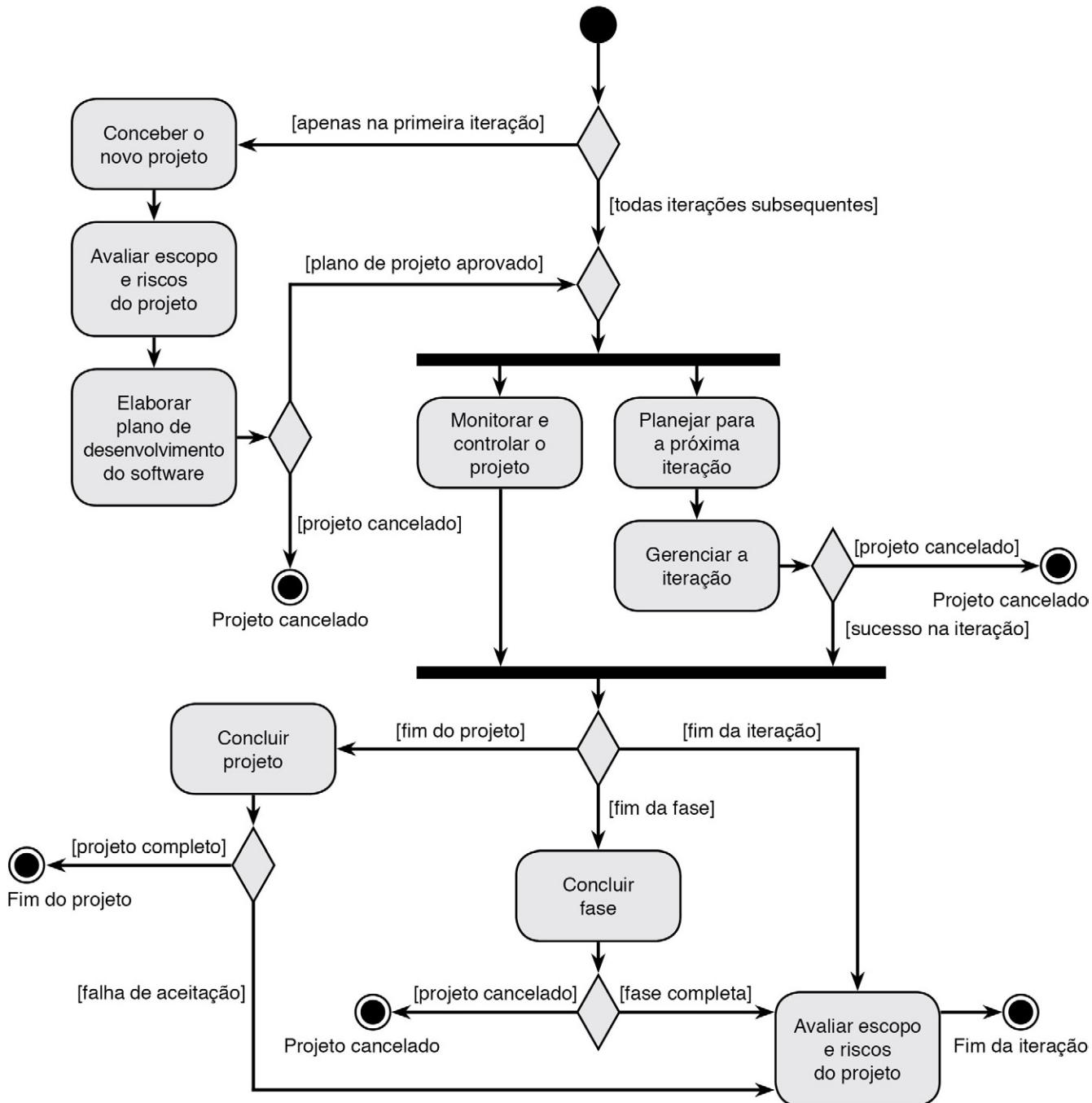


Figura 5.2 Workflow da disciplina gerenciamento de projeto⁵.

que esse investimento representará uma vantagem para ele, e não uma despesa inútil. A compreensão do modelo de negócio é fundamental para que a equipe de desenvolvimento permaneça sintonizada com essa compreensão e mantenha o cliente interessado.

Outro aspecto importante da modelagem de negócio é aproximar as equipes de engenharia de negócio e engenharia de software, de forma que os reais problemas e necessidades da organização sejam entendidos, analisados e solucionados com tecnologia da informação.

Essa disciplina, porém, nem sempre é necessária. Em alguns casos, quando o objetivo do projeto é simplesmente adicionar algumas funcionalidades a um sistema que não abrange um número significativo de pessoas, a modelagem

⁵Adaptado de: Kruchten (2003).

TABELA 5.1 Detalhamento do *workflow* para a disciplina de gerenciamento de projeto

| Atividade/objetivo | Subatividades | Responsável |
|---|--|--|
| Conceber o novo projeto. O propósito desta atividade é levar o projeto do ponto em que ele é apenas uma ideia para um ponto em que sua viabilidade possa ser avaliada. | Identificar e avaliar riscos Desenvolver o caso de negócios Iniciar o projeto Revisar a aprovação do projeto | Gerente de projeto Revisor de projeto |
| | Identificar e avaliar riscos Desenvolver o caso de negócios | Gerente de projeto |
| Avaliar escopo e riscos do projeto. Seu objetivo é reavaliar as características do projeto e os riscos associados. Isso é feito no início do projeto e ao final de cada iteração. | | |
| Elaborar plano de desenvolvimento do software. O objetivo desta atividade é desenvolver os componentes do plano de projeto e garantir que sejam formalmente revisados e validados. O maior esforço ocorre na primeira vez em que a atividade é executada. Depois, à medida que os ciclos avançam, o trabalho resume-se a revisar e atualizar os planos. | Desenvolver plano de medição Desenvolver plano de gerenciamento de risco Desenvolver plano de aceitação do produto Desenvolver plano de resolução de problemas Desenvolver organização de projeto e pessoal Definir processos de controle e monitoramento Planejar fases e iterações Compilar o plano de desenvolvimento do software Revisar planejamento do projeto | Gerente de projeto Revisor de projeto |
| Planejar para a próxima iteração. Seu objetivo é criar um plano de iteração detalhado para a iteração seguinte. | Desenvolver plano de iteração Desenvolver caso de negócios Elaborar plano de desenvolvimento do software Revisar plano de iteração | Gerente de projeto Revisor de projeto |
| Monitorar e controlar o projeto. Trata do dia a dia do gerente de projeto, incluindo gerenciamento das mudanças solicitadas, monitoramento do progresso e dos riscos ativos, relatório de <i>status</i> para a <i>Project Review Authority</i> (PRA) e lidar com problemas de acordo com o plano de resolução de problemas. | Escalonar e atribuir atividades Monitorar <i>status</i> do projeto Tratar exceções e problemas Relatar <i>status</i> Revisão para a autoridade de revisão de projeto (PRA) | Gerente de projeto Revisor de projeto |
| Gerenciar a iteração. Detalha como iniciar, terminar e revisar uma iteração. Trata da obtenção dos recursos necessários, alocação de trabalho e avaliação dos resultados da iteração. | Obter pessoal Iniciar iteração Avaliar iteração Revisão de critérios de avaliação de iteração Revisão de aceitação de iteração | Gerente de projeto Revisor de projeto |
| Concluir fase. A fase é fechada quando o gerente de projeto garante que: todos os principais problemas das iterações anteriores foram resolvidos, o <i>status</i> de todos os artefatos é conhecido, os artefatos requeridos foram entregues aos interessados, os eventuais problemas de implantação foram solucionados e todas as questões financeiras foram resolvidas, caso o contrato corrente esteja terminando. | Preparar para conclusão de fase Revisão de macro-objetivo (<i>milestone</i>) de fase | Gerente de projeto Revisor de projeto |

TABELA 5.1 Detalhamento do *workflow* para a disciplina de gerenciamento de projeto (*Continuação*).

| Atividade/objetivo | Subatividades | Responsável |
|--|---|--|
| Concluir projeto. O projeto é preparado para seu término, e a posse dos ativos é repassada ao cliente. | Preparar para conclusão do projeto Revisão de aceitação de projeto | Gerente de projeto Revisor de projeto |

de negócio pode ser desnecessária. Ela é mais importante quando mudanças significativas de comportamento são introduzidas para um grupo de pessoas. Nesse caso, a compreensão do negócio e as consequências da instalação de um novo sistema devem ser estudadas e conhecidas.

A modelagem de negócio pode ter ainda diferentes graus de ênfase, em função das necessidades do projeto no qual a equipe vai se engajar. Kruchten (2003) identifica seis cenários de complexidade crescente em termos de necessidade de modelagem de negócio:

- a) *Organograma*: pode-se querer apenas construir um organograma da organização para saber quais são os setores e as responsabilidades. Nesse caso, a modelagem de negócio em geral acontece apenas na fase de concepção.
- b) *Modelagem de domínio*: se o objetivo for construir aplicações para gerenciar e apresentar informação, então faz-se a modelagem de negócio com a modelagem do domínio, ou seja, um modelo de informação estático em que os *workflows* da empresa não são considerados. A modelagem de domínio costuma ser feita nas fases de concepção e elaboração.
- c) *Uma empresa, vários sistemas*: pode-se chegar ao ponto de desenvolver toda uma família de sistemas para uma empresa. Nesse caso, a modelagem de negócio vai tratar não apenas de um sistema, mas de vários projetos, e poderá inclusive ser tratada como um projeto à parte. Ela ajudará a descobrir os requisitos dos sistemas individuais, bem como a determinar uma arquitetura comum para a família de sistemas.
- d) *Modelo de negócio genérico*: se o objetivo for a construção de um ou mais aplicativos que sirvam a um grupo de empresas, então a modelagem de negócio poderá ser útil para ajudar a alinhar as empresas a uma visão de negócio, ou, se isso não for possível, obter uma visão de negócio em que as especificidades das empresas seja visível.
- e) *Novo negócio*: se uma organização resolve iniciar um novo negócio e todo um conjunto de sistemas de informação precisa ser desenvolvido para dar suporte a ele, então um esforço significativo de modelagem de negócio deve ser realizado. Nesse caso, o objetivo da modelagem de negócio não é apenas encontrar requisitos, mas verificar a viabilidade efetiva do novo negócio. Assim, a modelagem de negócio nessa situação costuma ser um projeto à parte.
- f) *Renovação*: se uma organização resolve renovar completamente seu modo de fazer negócio, então a modelagem de negócio será um projeto à parte, executado em várias etapas: visão do novo negócio, engenharia reversa do negócio existente, engenharia direta do novo negócio e instalação do novo negócio.

Os principais papéis RUP para modelagem de negócio são os seguintes:

- a) *Analista de processo de negócio*: lidera e coordena a visão de negócio, definindo os atores externos e processos a serem modelados.
- b) *Designer de negócio*: detalha a visão de negócio da organização, determinando como os atores colaboram e usam objetos de negócio para realizar objetivos de negócio.
- c) *Interessados (stakeholders)*: representam várias partes da organização que podem prover informações ou revisão.
- d) *Revisor de negócio*: revisa os artefatos resultantes.

Os principais artefatos dessa disciplina são os seguintes:

- a) *Documento de visão de negócio*: define os objetivos e as metas do esforço de modelagem de negócio.
- b) *Modelo de casos de uso de negócio*: modelo das funções da empresa usado para identificar papéis e entregas (*deliverables*) da organização.
- c) *Modelo de análise de negócio*: modelo de objetos que descreve a realização dos casos de uso de negócio.

Ao contrário do modelo de caso de uso do software, cujos atores são os usuários do sistema, o modelo de caso de uso de negócio modela a empresa inteira como um sistema, e os atores são as pessoas e organizações que se relacionam (fazem negócios) com ela. Da mesma forma, os objetos de negócio não são necessariamente instâncias de classes de um sistema, mas departamentos, pessoas e sistemas inteiros que operam dentro da empresa (Kroll & Kruchten, 2003).

Outros artefatos que também podem ser construídos são os seguintes:

- a) *Avaliação da organização-alvo*: descrição do *status* corrente da organização na qual o sistema será instalado.
- b) *Regras de negócio*: declaração de políticas e condições que devem ser satisfeitas.
- c) *Especificações suplementares de negócio*: documento que apresenta definições do negócio não contempladas no modelo de caso de uso ou no modelo de objetos.
- d) *Glossário de negócio*: definição de termos técnicos relevantes ao negócio.
- e) *Documento de arquitetura de negócio*: uma visão de aspectos arquiteturais do negócio a partir de vários pontos de vista. Deve ser usado apenas quando decisões sobre mudanças no negócio devem ser feitas ou quando o negócio necessita ser descrito a terceiros.

Jacobson (1994) apresenta, detalhadamente, técnicas de modelagem de negócio com casos de uso. Em geral, nos modelos de negócio, atores são entidades externas à empresa-alvo, casos de uso são os processos internos da empresa que normalmente afetam esses atores externos, e os subsistemas podem ser setores ou departamentos da empresa. Usam-se as mesmas técnicas de modelagem de caso de uso para sistemas informatizados, mas o nível de abstração é maior, porque a empresa como um todo é considerada um sistema.

O *workflow* da disciplina de modelagem de negócio é apresentado na Figura 5.3.

O modelo de casos de uso de negócio pode ser usado para gerar o modelo de casos de uso do software. Objetos de negócio ativos possivelmente serão atores ou sistemas-atores no modelo de casos de uso do software, e os processos de negócio possivelmente serão quebrados em processos individuais que serão identificados como casos de uso do software. Exemplos podem ser encontrados em Kroll e Kruchten (2003). A vantagem de se iniciar com o modelo de negócio, na maioria das vezes, é conseguir perceber mais claramente o sistema a ser desenvolvido no contexto geral da empresa.

5.3.2.3 Requisitos

Requisitos são a expressão mais detalhada sobre aquilo de que o usuário ou cliente precisa em termos de um produto de software. A disciplina de requisitos RUP, entretanto, trata não apenas dos requisitos (que são necessidades expressas pelo cliente), mas também do *design* da interface do sistema (que é uma possível solução para as necessidades apresentadas).

Os objetivos dessa disciplina são os seguintes (Kruchten, 2003):

- a) Estabelecer e manter concordância com o cliente e outros interessados sobre o que o sistema deve fazer, incluindo o porquê.
- b) Fornecer aos desenvolvedores melhor compreensão sobre os requisitos do sistema.
- c) Delimitar o escopo do sistema, ou seja, o que pertence e o que não pertence a ele.
- d) Prover uma base para o planejamento técnico das iterações.
- e) Prover uma base para a estimativa de custo e tempo de desenvolvimento.
- f) Definir uma interface com usuário focada em suas necessidades e objetivos.

O *workflow* RUP para a disciplina inclui cinco papéis:

- a) *Analista de sistemas*: lidera e coordena a elicitação de requisitos e a modelagem de casos de uso, especificando os limites do sistema.
- b) *Especificador de requisitos* ou *especificador de casos de uso*: é responsável pelo detalhamento ou expansão dos requisitos, na forma de casos de uso.
- c) *Arquiteto*: deve assegurar que uma arquitetura consistente poderá ser obtida a partir dos casos de uso.
- d) *Revisor de requisitos*: é composto por vários tipos de pessoas que revisam os requisitos, para garantir que estejam de acordo com normas e padrões ou com as necessidades dos interessados.

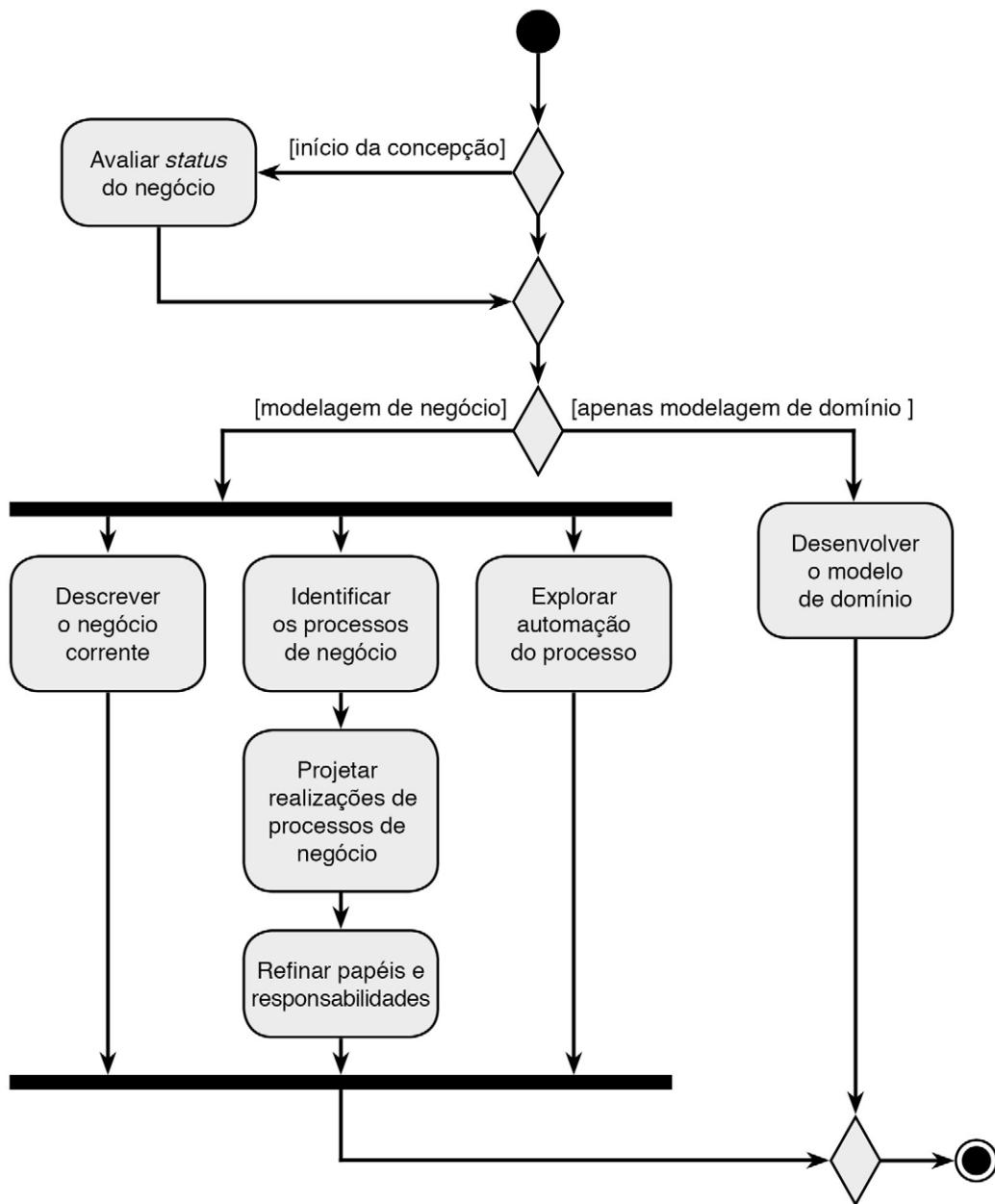


Figura 5.3 Workflow da disciplina de modelagem de negócio.

Os três principais artefatos da disciplina de requisitos são os seguintes:

- a) *Visão geral de sistema*: é um documento de alto nível que indica especialmente aos financiadores do projeto qual é seu escopo e quais são suas vantagens. O documento pode ter uma visão do que será e do que não será incluído, capacidades operacionais desejadas, perfis de usuários e interfaces com sistemas externos, quando for o caso.
- b) *Modelo de casos de uso*: os casos de uso incorporam os requisitos funcionais e boa parte dos não funcionais. Eles servem de definição das funcionalidades desejadas e são usados ao longo de todo o processo de desenvolvimento.
- c) *Especificações suplementares*: consistem nos requisitos suplementares, isto é, nas restrições gerais que se aplicam ao sistema como um todo, e não apenas a um ou outro caso de uso.

De modo complementar, a disciplina de requisitos também pode produzir:

- a) Um *glossário* que contenha os termos técnicos do jargão da aplicação.
- b) *Storyboards*, que servem de base para o *design* da interface com o usuário.

O *workflow* da disciplina de requisitos é mostrado no diagrama de atividades UML da Figura 5.4.

As atividades do *workflow* podem ser assim caracterizadas:

- Analisar o problema*: obter concordância dos interessados sobre o enunciado do problema que se está tentando resolver. Devem-se também identificar os próprios interessados, os limites e as restrições do sistema.
- Entender necessidades dos interessados*: usando variadas técnicas de elicição, obter os requisitos e as reais necessidades dos interessados.
- Definir o sistema*: baseando-se nas necessidades dos interessados, estabelecer um conjunto de funcionalidades que se pode considerar para entrega. Determinar os critérios para a priorização das necessidades mais importantes e identificar atores e casos de uso para cada funcionalidade-base do sistema.
- Gerenciar o escopo do sistema*: coletar informações importantes dos interessados e mantê-las na forma de atributos de requisitos, identificando, por exemplo, a importância do requisito, sua urgência, impacto no negócio etc.

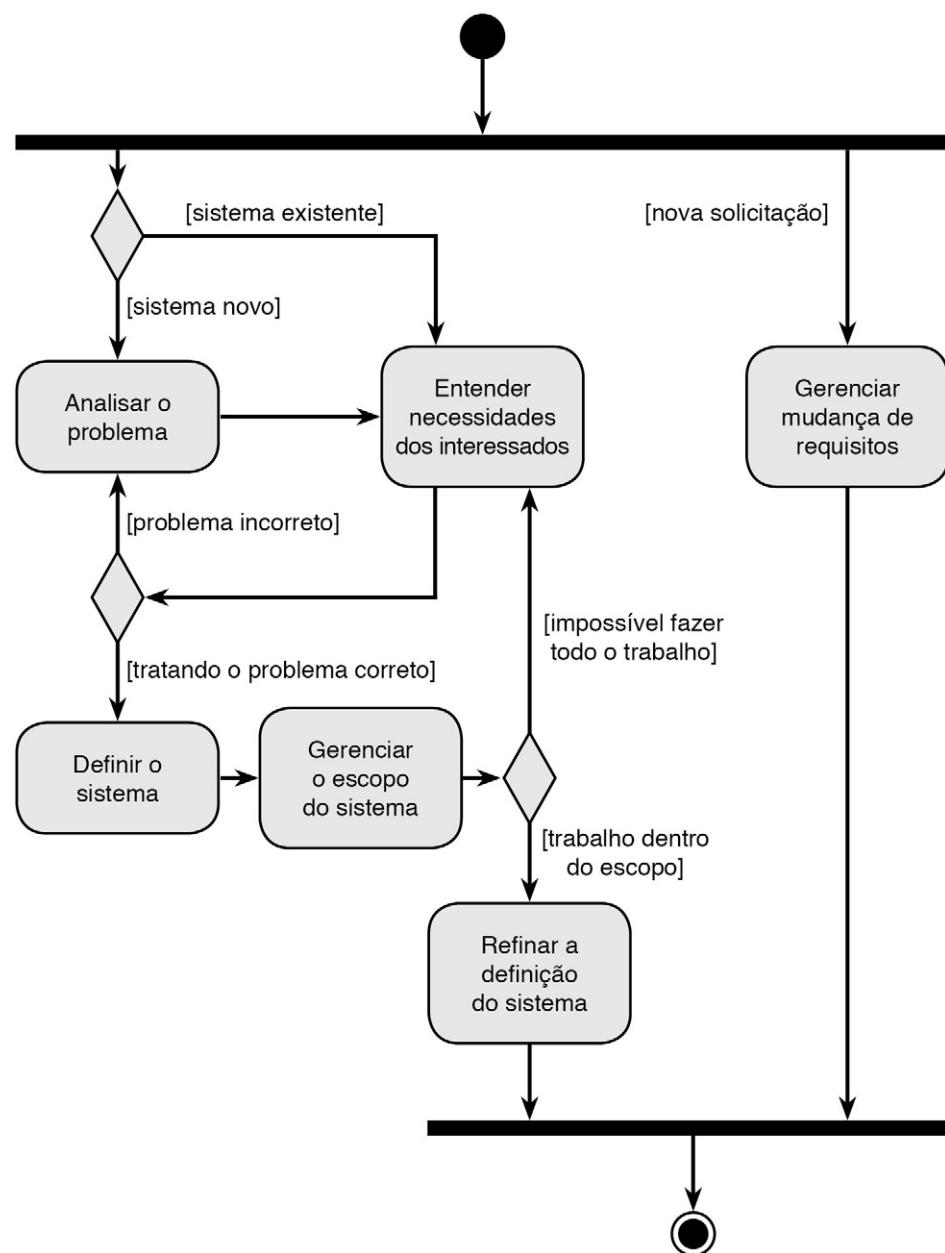


Figura 5.4 Workflow da disciplina de requisitos.

- e) *Refinar a definição do sistema:* usando o modelo de casos de uso, refinar a descrição dos requisitos funcionais e obter os não funcionais associados.
- f) *Gerenciar mudança de requisitos:* usar atributos de requisitos e rastreabilidade para avaliar o impacto da mudança de requisitos. Usar uma autoridade central para controlar as mudanças nos requisitos. Manter entendimento com o cliente e definir expectativas realistas sobre o que poderá ser feito.

5.3.2.4 Análise e Design

Análise implica o estudo mais aprofundado do problema. Os requisitos são detalhados e incluídos em modelos de análise, como o modelo conceitual, o de interação e o funcional. Já o *design* consiste em apresentar uma possível solução tecnológica para o modelo de análise. Os modelos de *design* podem ser o modelo dinâmico, de interface, de persistência, além de outros. Wazlawick (2011) apresenta muitas informações sobre como realizar essa disciplina do RUP.

Para o RUP, a disciplina de análise e *design* tem como característica principal a modelagem, ou seja, a transformação dos requisitos em modelos úteis para a geração de código. Boa parte da análise considera apenas a funcionalidade, mas não as restrições, especialmente as tecnológicas. Ou seja, a análise concentra-se no sistema ideal, independentemente de tecnologia, e o *design*, por sua vez, vai lidar com essas questões mais físicas.

Os dois principais papéis relacionados a essa disciplina são os seguintes:

- a) *Arquiteto de software:* coordena as atividades técnicas do projeto, estabelecendo a estrutura sobre a qual a arquitetura final é construída. Ele define os agrupamentos de elementos (como componentes ou pacotes) e decide quais serão suas interfaces. Entretanto, sua visão do sistema ocorre mais em extensão do que em profundidade.
- b) *Designer:* cada *designer* define as responsabilidades, operações, atributos e associações de um conjunto de classes ou pacotes colocados sob sua responsabilidade.

Pode-se ainda ter os seguintes papéis opcionais:

- a) *Designer de banco de dados:* em geral, é necessário quando o sistema utiliza banco de dados, especialmente se for complexo.
- b) *Designer de cápsula:* é necessário apenas no caso de sistemas de tempo real e é responsável por garantir que o sistema responderá da forma apropriada.
- c) *Revisor de arquitetura e revisor de design:* são especialistas que revisam os artefatos produzidos nessa disciplina.

Os principais artefatos dessa disciplina são os seguintes:

- a) *Modelo de design:* é o principal *design* do sistema que está sendo construído. Em geral, é um diagrama de componentes ou pacotes incluindo as principais classes do sistema.
- b) *Documento de arquitetura de software:* captura vários aspectos da arquitetura do sistema sob diferentes pontos de vista. Por exemplo, visão de módulos (como o sistema é estruturado como unidades de código), visão de tempo de execução (como o sistema é estruturado como elementos com comportamento e interações), visão de implantação ou *deployment* (como o sistema é implantado no hardware), visão de implementação (como os artefatos se organizam em um sistema de arquivos) e modelo de dados (qual a estrutura do repositório de dados).

O *workflow* da disciplina de análise e *design* é mostrado na Figura 5.5.

A Figura 5.5 mostra que algumas atividades são dependentes da fase em que o projeto se encontra. Na fase de concepção poderá ser feita a *síntese arquitetural*, que é uma prova de conceito de que a arquitetura é viável. No início da elaboração será, então, definida uma arquitetura candidata inicial, possivelmente baseada nessa prova de conceito. Nas iterações subsequentes, a ênfase mudará para a adição ou o refinamento de funcionalidades sobre a arquitetura, que vai se estabilizando até a fase de construção.

A Tabela 5.2 mostra o detalhamento das atividades desse *workflow*.

As atividades de análise e *design* e seus conceitos correlatos são detalhadamente apresentadas por Wazlawick (2011).

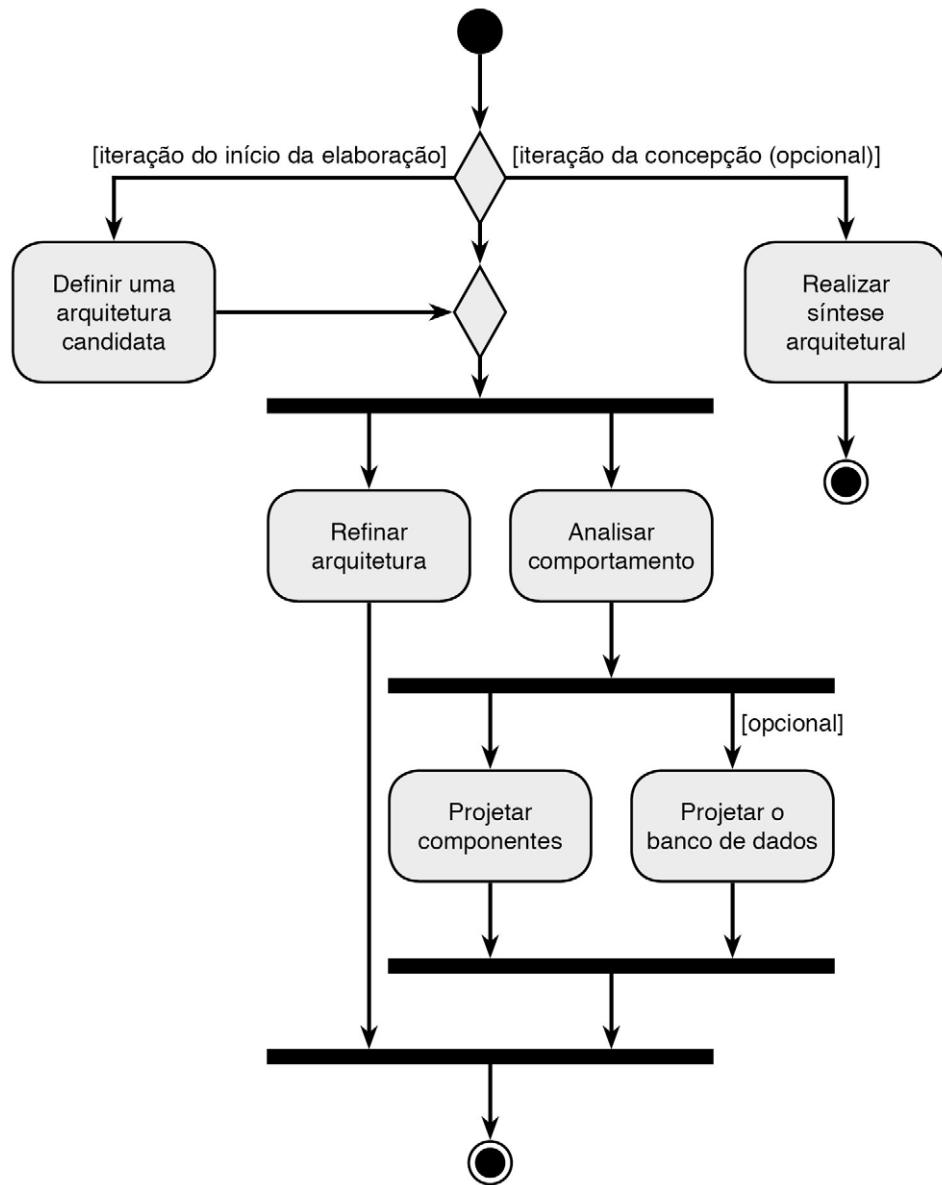


Figura 5.5 Workflow da disciplina de análise e design.

5.3.2.5 Implementação

Implementar é mais do que simplesmente produzir um código. Implica também organizar esse código em componentes ou pacotes, definir possíveis camadas de implementação, realizar testes de unidade e integrar o código de forma incremental.

Apenas o teste de unidade é tratado na disciplina de implementação. Os demais tipos de teste, inclusive os testes de integração, são tratados na disciplina de teste.

A implementação de código em RUP pode ser de três tipos:

- a) *Versões (builds)*: são versões operacionais de um sistema que introduzem novas funcionalidades. Versões devem ser controladas e rastreadas para que alterações possam ser desfeitas em caso de necessidade. Projetos típicos apresentam novas versões regularmente; pelo menos uma versão por semana é desejável, podendo chegar a uma por dia.
- b) *Integração*: a integração consiste na combinação de pelo menos duas partes de software independentes. A integração no RUP deve ser contínua, como nos métodos ágeis e diferentemente da integração por fase, que ocorre no Modelo Cascata com Subprojetos. Uma das maiores vantagens da integração contínua consiste na

TABELA 5.2 Detalhamento do *workflow* para a disciplina de análise e *design*

| Atividade/objetivo | Subatividades | Responsável |
|---|---|--|
| Realizar síntese arquitetural. Deve-se construir uma prova de conceito para mostrar que a arquitetura é viável. | Análise arquitetural Construir prova de conceito arquitetural Avaliar viabilidade de prova de conceito arquitetural | Arquiteto |
| Definir uma arquitetura candidata. Deve-se criar um esboço inicial da arquitetura a partir da modelagem conceitual e dos casos de uso. | Análise arquitetural Análise de casos de uso | Arquiteto <i>Designer</i> |
| Refinar a arquitetura. Deve-se fazer a transição natural da análise para o <i>design</i> , identificando os elementos e mecanismos apropriados e mantendo a consistência e a integridade da arquitetura. | Identificar mecanismos de <i>design</i> Identificar elementos de <i>design</i> Incorporar elementos de <i>design</i> existentes Descrever a arquitetura de tempo de execução Descrever a distribuição do sistema Revisar a arquitetura | Arquiteto Revisor de arquitetura |
| Analizar comportamento. Devem-se transformar as descrições de comportamento do caso de uso em elementos arquiteturais. | Análise de casos de uso Identificar elementos de <i>design</i> Revisar o <i>design</i> | <i>Designer</i> Arquiteto Revisor de <i>design</i> |
| Projetar componentes. Devem-se detalhar completamente classes e relações, interfaces de subsistemas e casos de uso reais. | <i>Design</i> de casos de uso <i>Design</i> de classes <i>Design</i> de subsistemas Revisar o <i>design</i> | <i>Designer</i> Revisor de <i>design</i> |
| Projetar o banco de dados. Devem-se identificar as classes persistentes, projetar estruturas de BD adequadas para armazená-las e definir mecanismos de salvamento e recuperação que atendam a eventuais requisitos suplementares. | <i>Design</i> de banco de dados <i>Design</i> de classes Revisar o <i>design</i> | <i>Designer</i> de banco de dados <i>Designer</i> Revisor de <i>design</i> |

maior facilidade de encontrar erros, já que a maior fonte potencial de erros durante o processo será o novo módulo que estiver sendo integrado, bem como seus pontos de interação com o restante do sistema. RUP prevê que deve acontecer, no mínimo, uma integração por ciclo, mas o ideal é que aconteçam várias integrações por dia.

- c) *Protótipos*: devem ser usados primordialmente para reduzir riscos, seja para ter melhor compreensão dos requisitos, seja para obter provas de conceito de arquitetura ou usabilidade.

Os principais papéis relacionados com a disciplina de implementação no RUP são os seguintes:

- a) *Implementador*: desenvolve os componentes e artefatos relacionados e realiza o teste de unidade.
- b) *Integrador*: constrói as versões (*builds*).
- c) *Arquiteto de software*: define a estrutura ou modelo de implementação, estabelecendo, por exemplo, camadas ou partições de sistema.
- d) *Revisor de código*: inspeciona o código de acordo com padrões de codificação estabelecidos e as boas práticas.

Os principais artefatos relacionados a essa disciplina são os seguintes:

- a) *Subsistema de implementação*: coleção de elementos de implementação e outros subsistemas de implementação usados para estruturar o modelo em subsistemas que dividem o todo em partes menores.
- b) *Elementos de implementação*: um artefato de código (fonte, binário ou executável) ou arquivo contendo informação (*startup*, *readme* etc.), podendo agrregar outros elementos de implementação, como uma aplicação contendo vários executáveis.
- c) *Plano de integração de implementação*: documento que define a ordem em que os elementos e subsistemas devem ser implementados e especifica as versões a serem criadas à medida que as integrações ocorrem.

A Figura 5.6 mostra o *workflow* da disciplina de implementação no RUP.

Em cada iteração, o plano de integração deve indicar quais sistemas serão implementados e em que ordem eles serão integrados. O responsável pelo subsistema definirá a ordem em que as classes serão implementadas (Seção 13.2.2).

Se os implementadores encontrarem erros de *design*, deverão submeter um *feedback* de retrabalho sobre o *design*.

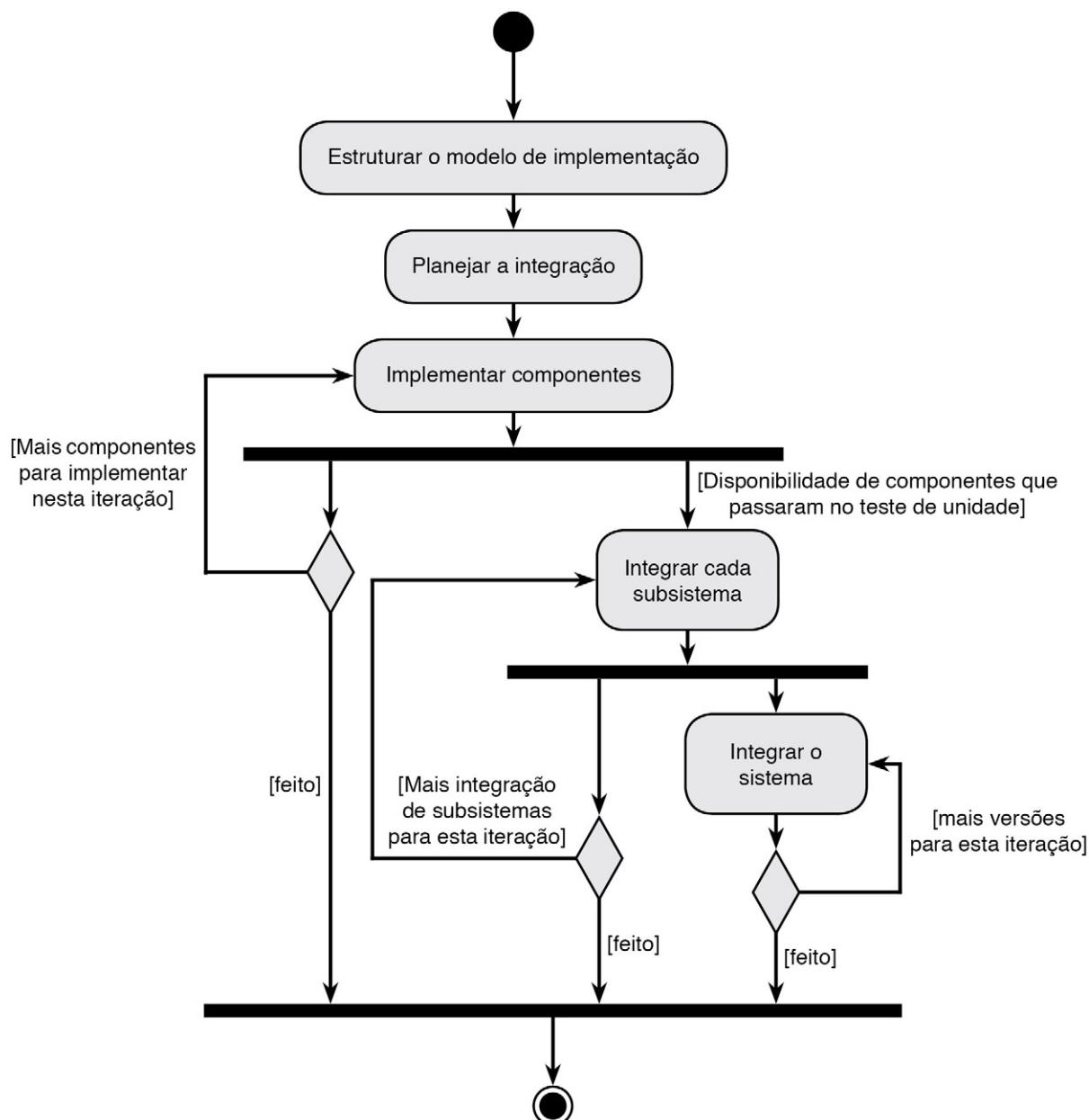


Figura 5.6 Workflow da disciplina de implementação.

5.3.2.6 Teste

A disciplina de teste no RUP exclui os testes de unidade, que são executados pelo programador na disciplina de implementação. O propósito da disciplina de testes é:

- a) Verificar a interação entre objetos.
- b) Verificar se todos os componentes foram integrados adequadamente.
- c) Verificar se todos os requisitos foram corretamente implementados.
- d) Verificar e garantir que defeitos tenham sido identificados e tratados antes da entrega do produto final.
- e) Garantir que todos os defeitos tenham sido consertados, retestados e estancados.

No RUP, a disciplina de teste é executada ao longo de todos os ciclos, o que facilita a detecção prematura de erros de requisitos e de implementação. Ao contrário das demais disciplinas, a disciplina de teste visa encontrar as fraquezas do sistema, isto é, verificar como e quando o sistema poderia falhar.

Os principais papéis envolvidos com a disciplina de teste são os seguintes:

- a) *Gerente de teste*: é o responsável geral pela disciplina de teste. Deve planejar o uso de recursos e gerenciar o andamento dos testes, ajudando a resolver conflitos. Além disso, é responsável pelo plano de teste e pelo sumário de avaliação de testes.
- b) *Analista de teste*: é responsável pela identificação e definição dos testes requeridos, pela lista de ideias de teste, pelos casos de teste, pelo modelo de análise de carga, pelos dados de teste e pelos resultados dos testes.
- c) *Designer de teste*: é responsável pela definição da abordagem de teste e pela garantia do sucesso em sua implementação. Além disso, é responsável pela estratégia de teste, pela arquitetura de automação de testes, pela especificação da interface de teste, pela configuração do ambiente de teste e pela sequência de testes.
- d) *Testador*: é responsável pela execução dos testes, pelos *scripts* de teste e pelo *log* de testes.

Em relação aos artefatos mencionados, podem-se destacar os seguintes:

- a) *Plano de teste*: contém informação sobre os objetivos e metas do teste, bem como sobre seu cronograma. Identifica as estratégias a serem usadas, os recursos necessários e as configurações de teste requeridas.
- b) *Lista de ideias de teste*: é uma lista mantida pelo analista de teste que enumera ideias, às vezes parcialmente formadas, sobre como executar testes que possam ser úteis.
- c) *Casos de teste*: são a evolução natural de algumas ideias de teste, especificando de forma mais detalhada o teste, suas condições e os dados a serem usados.
- d) *Scripts de teste*: são procedimentos manuais ou automáticos usados pelo testador para proceder aos testes.
- e) *Modelo de análise de carga*: é um tipo especial de teste de *performance* que identifica variáveis e seus valores a serem usados em diferentes testes de *performance* para simular ou emular características dos atores.
- f) *Log de testes*: é o conjunto de dados crus capturados como resultado da execução das sequências de testes.
- g) *Resultados de testes*: consistem na análise do *log*, que produz um relatório sobre os defeitos encontrados e as melhorias necessárias.

Um típico *workflow* da disciplina de teste em RUP é mostrado na Figura 5.7.

As atividades apresentadas no *workflow* são detalhadas da seguinte forma:

- a) *Definir missão de avaliação*: devem-se definir o foco e os objetivos dos testes para a iteração e conseguir a concordância dos interessados.
- b) *Verificar abordagem de teste*: deve-se demonstrar que as várias abordagens de teste delineadas na estratégia de teste são adequadas e podem produzir resultados confiáveis.
- c) *Validar estabilidade da versão*: antes de iniciar o ciclo de teste para uma nova versão, deve-se verificar se ela é suficientemente estável para que se inicie o ciclo de teste. Isso evita que se perca tempo fazendo testes sistemáticos em uma versão que ainda não está suficientemente madura.
- d) *Testar e avaliar*: consiste nas atividades clássicas de teste, ou seja, implementação, execução e avaliação dos resultados do teste.
- e) *Obter missão aceitável*: deve produzir, para os interessados, resultados úteis e compreensíveis de acordo com a missão de avaliação.

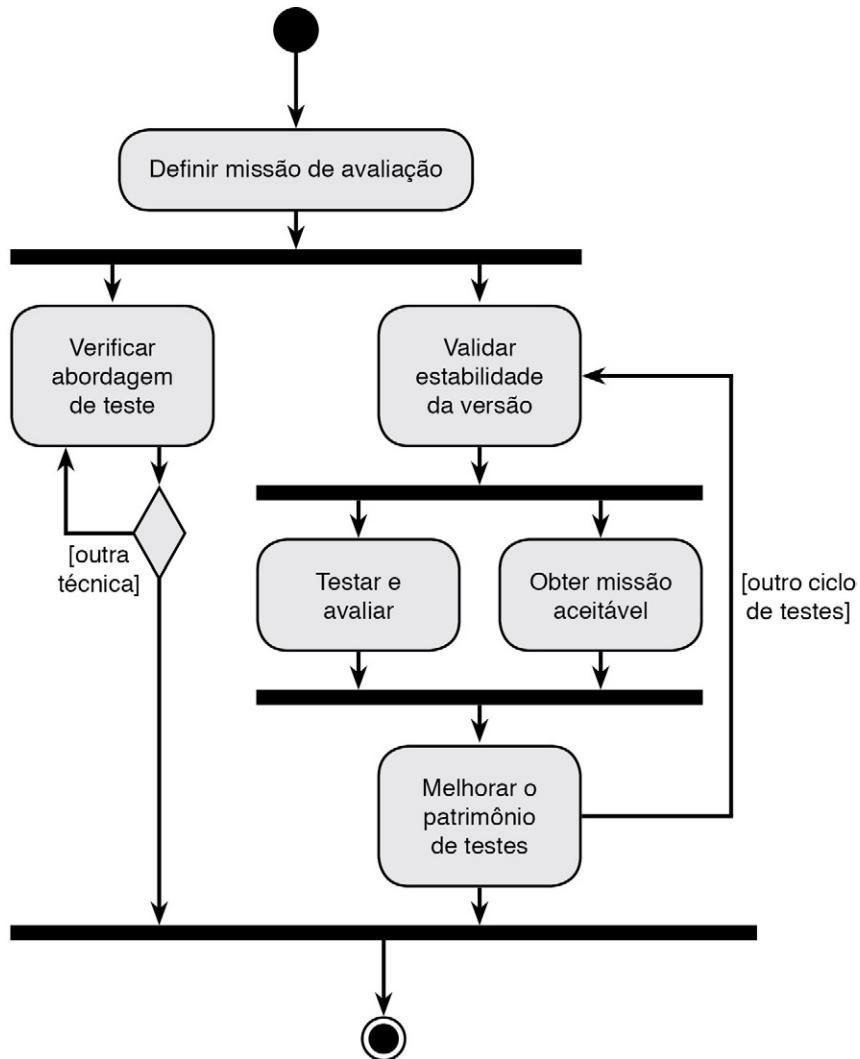


Figura 5.7 Workflow da disciplina de testes.

- f) *Melhorar o patrimônio de testes*: deve manter e aprimorar o patrimônio de testes, que compreende os seguintes artefatos: lista de ideias de teste, casos de teste, *scripts* de teste etc. É importante capitalizar esses elementos, porque eles podem ser úteis em testes futuros.

A disciplina de teste é detalhadamente explorada no Capítulo 13 e está fortemente relacionada com a noção de qualidade de software (Capítulo 11). Basicamente, a qualidade é uma característica que não pode ser adicionada a um sistema no final de sua produção; ela precisa estar presente todo o tempo. A disciplina de teste é, assim, exercitada em todas as fases do projeto.

5.3.2.7 Implantação

O propósito da disciplina de implantação é a produção de versões (*releases*) do produto a serem entregues aos usuários finais. Entre outras atividades, incluem-se a produção do software, seu empacotamento, instalação, migração de dados e apoio aos usuários (treinamento e suporte).

Apesar de a disciplina de implantação se concentrar na fase de transição, quando se está entregando o produto definitivo, podem existir várias *releases* ao longo do projeto, em que produtos preliminares podem ser entregues mesmo nas fases de elaboração ou construção.

Existem vários tipos de implantação de software, de acordo com a maneira como ele é distribuído. Vão desde sistemas personalizados a serem implantados diretamente nos computadores do cliente até software disponibilizado

para *download* pela internet. A diferença entre os casos consiste no grau de envolvimento da empresa desenvolvedora do software com a instalação do produto no ambiente final.

Os seguintes papéis podem ser citados na disciplina de implantação:

- a) *Gerente de implantação*: planeja e organiza a implantação. É responsável pelos resultados do beta-teste e pela verificação do empacotamento correto do produto para envio.
- b) *Gerente de projeto*: é o responsável pela interface com o cliente e deve aprovar a versão final do sistema com base nos resultados dos beta-testes.
- c) *Redator técnico*: planeja e produz o material escrito de apoio aos usuários.
- d) *Desenvolvedor de cursos*: planeja e produz material de treinamento em conjunto com o redator técnico.
- e) *Artista gráfico*: responsável pela produção artística, quando necessária.
- f) *Testador*: executa os testes de aceitação e operação.
- g) *Implementador*: cria os *scripts* de instalação e artefatos relacionados à instalação do produto.

Dependendo do tipo de implantação, diferentes artefatos poderão ser ou não necessários. O artefato principal, em todos os casos, é a *release*, ou seja, a versão final do software, que poderá ser composto pelos seguintes artefatos:

- a) O software executável.
- b) Artefatos de instalação, como *scripts*, ferramentas, arquivos, informação de licenciamento e orientações.
- c) Notas sobre a versão (*release notes*) descrevendo-a para o usuário final.
- d) Material de suporte, como manuais de operação e de manutenção do sistema.
- e) Material de treinamento.

No caso de software empacotado para venda em prateleira, os seguintes artefatos ainda poderão ser necessários:

- a) *Lista de materiais*: lista completa dos itens incluídos no produto.
- b) *Arte do produto*: parte do empacotamento que permite a identificação visual do produto.

Outros artefatos importantes da disciplina de implantação, mas que não são usualmente enviados ao cliente, são os resultados dos testes.

O *workflow* da disciplina de implantação é apresentado na Figura 5.8.

Entre essas atividades destacam-se:

- a) *Planejar implantação*: é necessário ter uma previsão adequada das atividades e recursos que serão necessários para a implantação, bem como contar com a participação intensa do cliente nesse planejamento.
- b) *Desenvolver material de suporte*: esse material cobre todos os documentos e treinamentos que serão necessários para que o usuário final possa instalar, operar, manter e usar o sistema efetivamente.
- c) *Producir unidade de implantação*: a unidade de implantação é o produto empacotado para uso com todos os artefatos necessários. Pode se tratar de versões beta ou mesmo de versões finais do produto, dependendo de seu estágio de desenvolvimento.
- d) *Gerenciar teste de operação*: o teste de operação é semelhante ao teste de sistema e ao teste de aceitação, mas é feito no local definitivo de uso do software com máquinas-alvo e dados reais.
- e) *Beta-teste*: o beta-teste é feito pelo usuário, que vai avaliar a funcionalidade, a *performance* e a usabilidade do sistema, dando *feedback* para uma nova iteração de desenvolvimento.
- f) *Empacotar produto*: nesse caso, o sistema e seus anexos são salvos em uma matriz para posterior reprodução. Todos os artefatos que o acompanham são efetivamente produzidos e reunidos para envio.
- g) *Prover acesso ao download*: o produto não só deve ser disponibilizado como deve-se tentar garantir acesso contínuo a ele, mesmo no caso das conexões mais lentas. Além disso, a internet é uma boa fonte de *feedback* para o produto, mesmo depois de testado, e esse retorno dos usuários pode ser capitalizado através do *site de download*.

5.3.2.8 Gerenciamento de Mudança e Configuração

A disciplina de gerenciamento da mudança e configuração tem como principal objetivo manter a integridade do conjunto e artefatos produzidos ao longo do projeto. Muito esforço é despendido na produção desses artefatos e, por isso, eles devem ser rastreáveis e disponíveis para reúso futuro. No RUP, a disciplina lida com três diferentes subáreas:

- a) *Gerenciamento de configuração*: responsável pela estruturação sistemática dos produtos. Artefatos como documentos e diagramas devem estar sob controle de versão, e mudanças feitas devem ser visíveis e localizáveis.

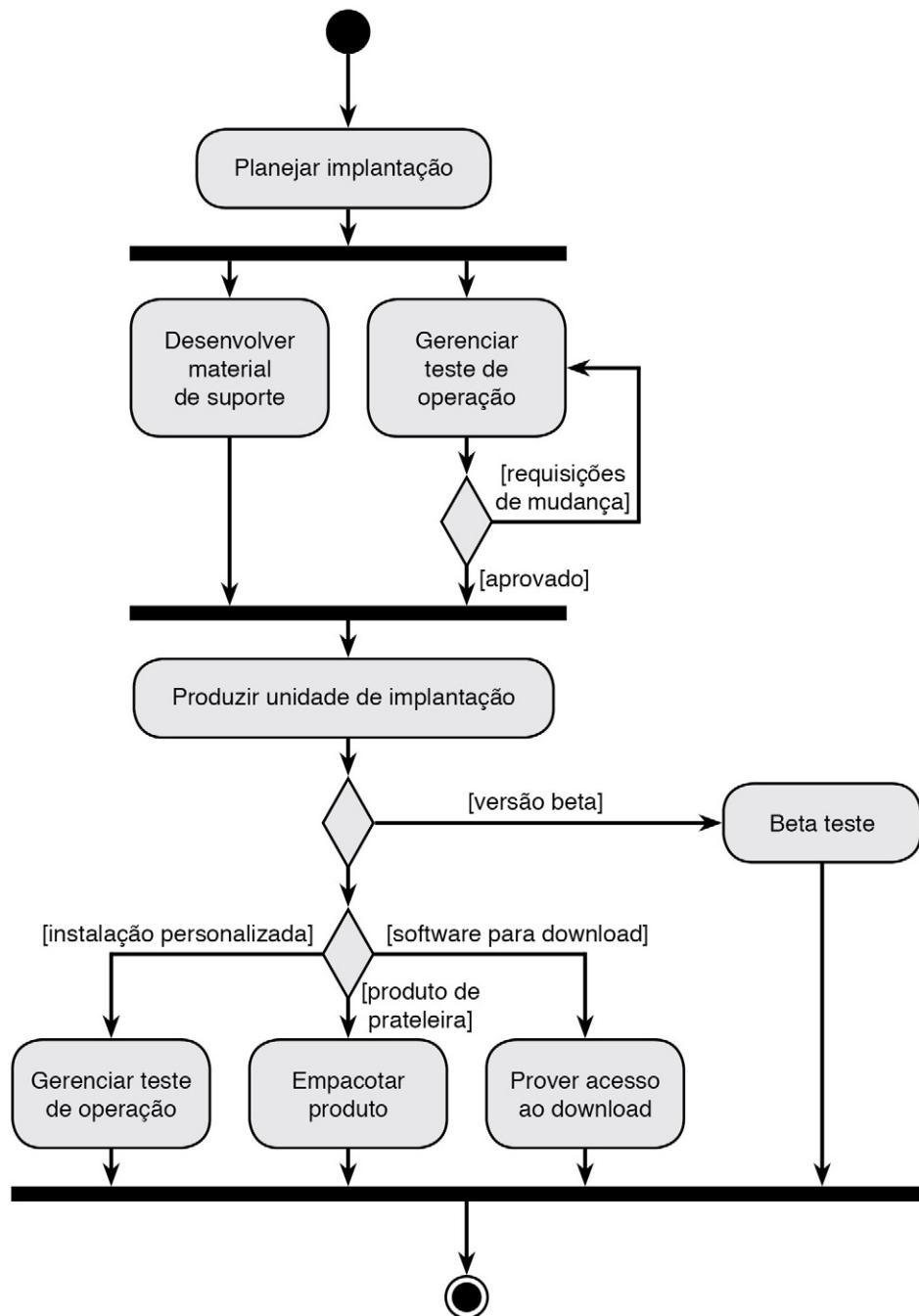


Figura 5.8 Workflow da disciplina de implantação.

Também é necessário manter um registro de dependências ou rastreabilidade entre os artefatos, de forma que artefatos relacionados sejam atualizados quando necessário.

- b) *Gerenciamento de requisições de mudança:* a área de CRM (*Change Request Management*) cuidará do controle das requisições de mudança em artefatos que produzem suas diferentes versões. Nem todas as requisições de mudança são efetivamente realizadas; elas dependem de uma decisão de gerência que considere custos, impacto e a real necessidade da mudança.
- c) *Gerenciamento de status e medição:* requisições de mudança têm *status* como: *novo*, *atribuído*, *retido*, *concluído* e *entregue*. Elas também podem ter atributos como *causa*, *fonte*, *natureza*, *prioridade* etc. O gerenciamento de *status* coloca todos esses atributos em um sistema de informação, tal que o andamento de cada solicitação seja verificável a todo momento pelo gerente do projeto.

Essas três dimensões formam o assim chamado *Cubo CCM*⁶ (*Configuration and Change Management*), que indica a forte inter-relação entre as três subáreas.

Os principais papéis relacionados a essa disciplina no RUP são os seguintes:

- a) *Gerente de configuração*: é o responsável por definir a estrutura do produto (versões de elementos e suas inter-relações) e por definir e alocar espaços de trabalho para desenvolvedores e integração. Ele também extrai os relatórios de *status* apropriados para o gerente do projeto.
- b) *Gerente de controle de mudança*: supervisiona o processo de controle de mudança. Em projetos grandes, esse papel pode ser executado por um comitê com representantes dos interessados, e, em projetos pequenos, por uma única pessoa, em geral o gerente do projeto ou o arquiteto de software.

Os principais artefatos dessa disciplina são os seguintes:

- a) *O plano de gerenciamento de configuração*: descreve as políticas e práticas a serem usadas para definir versões, variantes, espaços de trabalho e procedimentos para gerenciamento de mudança. É parte do plano de desenvolvimento de software.
- b) *Requisições de mudança*: podem ser de vários tipos, como relatórios de defeitos, mudança de requisitos ou incremento de um ciclo a outro. Cada mudança é associada a um originador e a uma causa principal. Mais tarde, a mudança também terá uma análise de impacto.

O *workflow* da disciplina de gerenciamento de mudança e configuração é apresentado na Figura 5.9.

As atividades descritas no *workflow* podem ser assim resumidas:

- a) *Planejar configuração de projeto e controle de mudança*: o plano de configuração de projeto descreve todas as atividades referentes ao controle de mudança, que devem ser atribuídas a responsáveis e ter recursos alocados durante o projeto. O plano também deve apresentar os meios para garantir que todos os interessados sejam informados sobre mudanças que ocorrerem em artefatos.
- b) *Criar ambientes de gerenciamento de mudança de projeto*: deve-se criar um ambiente de trabalho onde todos os artefatos em desenvolvimento possam ser acessados, modificados, integrados ou arquivados para uso posterior. O ambiente de gerenciamento de mudança oferece aos desenvolvedores e integradores espaços de trabalho privados e compartilhados para construção e integração de software.
- c) *Mudar e entregar itens de configuração*: essa atividade se refere à forma como cada papel pode criar um espaço de trabalho. Dentro de um espaço de trabalho, um papel pode acessar e alterar artefatos e disponibilizá-los para integração. A entrega é feita em um espaço de integração que pode receber mudanças de vários papéis. A partir do espaço de integração, o integrador constrói o produto, criando suas *baselines*.
- d) *Gerenciar baselines e releases*: a *baseline* é a descrição de todas as versões de artefatos que formam o produto em um dado momento. Costumam ser criadas ao final das iterações. Cada vez que uma entrega (*release*) for feita, deve haver necessariamente uma *baseline*.
- e) *Monitorar e relatar status da configuração*: o relatório de *status* da configuração indica a versão de cada artefato e suas relações com as versões anteriores, o que pode ser útil para verificar onde e quando ocorreram modificações no sistema.
- f) *Gerenciar requisições de mudança*: deve-se garantir que as mudanças em um projeto sejam feitas de maneira consistente.

O gerenciamento de mudança e configuração é uma tarefa árdua que talvez seja impossível de se realizar sem as ferramentas adequadas. Existem implementações livres de sistemas de controle de versões, como CVS,⁷ Git,⁸ Mercurial⁹ e Subversion.¹⁰ Entre as implementações comerciais, podem-se citar o Microsoft Visual SourceSafe¹¹ e o IBM Rational ClearCase.¹² Mais detalhes sobre essa disciplina serão apresentados no Capítulo 10.

⁶Disponível em: <flylib.com/books/en/1.560.1.134/1/>. Acesso em: 21 jan. 2013.

⁷Disponível em: <savannah.nongnu.org/projects/cvs/>. Acesso em: 21 jan. 2013.

⁸Disponível em: <git-scm.com/> . Acesso em: 21 jan. 2013.

⁹Disponível em: <mercurial.selenic.com/>. Acesso em: 21 jan. 2013.

¹⁰Disponível em: <subversion.apache.org/>. Acesso em: 21 jan. 2013.

¹¹Disponível em: <msdn.microsoft.com/pt-br/library/ms181038%28v=vs.80%29.aspx>. Acesso em: 21 jan. 2013.

¹²Disponível em: <www-01.ibm.com/software/awdtools/clearcase/>. Acesso em: 21 jan. 2013.

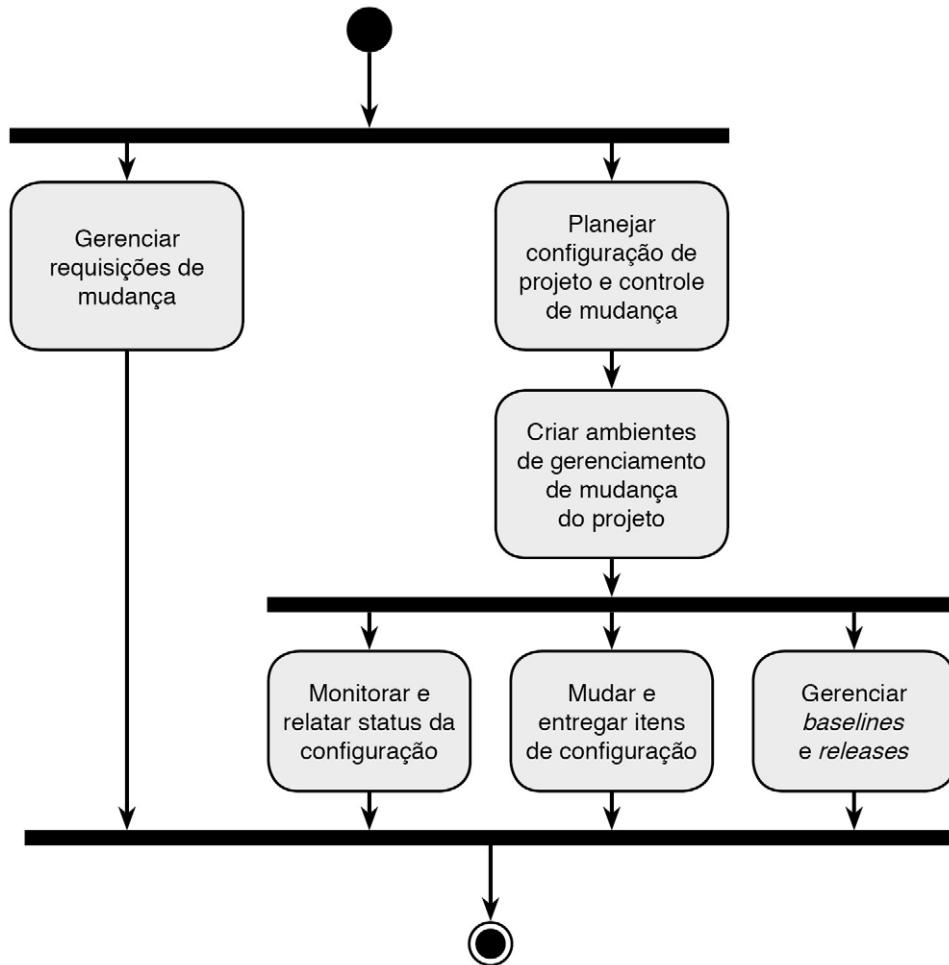


Figura 5.9 Workflow da disciplina de gerenciamento de mudança e configuração.

5.3.2.9 Ambiente

A disciplina de *ambiente* trata principalmente da configuração do próprio processo a ser usado para desenvolver o projeto. Em função do processo específico escolhido, essa disciplina deve também tratar das ferramentas de apoio necessárias para que a equipe tenha sucesso no projeto.

Se uma equipe tenta implementar RUP integralmente, sem perceber que na verdade ele é um *framework* para adaptação de processos, poderá ficar a impressão de que se trata de um processo altamente complexo (o que de fato é) e quase impraticável. É necessário que um especialista em RUP avalie o escopo do projeto e a realidade da equipe para decidir quais elementos do RUP precisam ser usados para que sua adoção seja facilitada (ver também Seção 12.5).

Além disso, algumas versões mais simples de RUP, como AUP e OpenUP, já se estabeleceram como implementações independentes e podem ser consideradas no lugar da original.

Em relação à disciplina de ambiente, convém mencionar que ela não se refere ao produto, como as outras, mas à engenharia do processo em si, visando ao seu aprimoramento. Os principais papéis relacionados a essa disciplina são os seguintes:

- a) *Engenheiro de processo*: é o responsável pelo processo de desenvolvimento, caso de desenvolvimento, regras específicas de projeto e *templates* específicos de projeto.
- b) *Especialista em ferramentas*: é o responsável pelas ferramentas a serem usadas.
- c) *Administrador de sistema*: é o responsável pela infraestrutura de desenvolvimento.
- d) *Redator técnico*: é o responsável pelo manual de guia de estilos.

Dentre os artefatos mencionados, talvez o principal para um projeto específico seja o *caso de desenvolvimento*, que é a versão do processo geral especialmente adaptada para o projeto em questão, disciplina por disciplina.

O *workflow* para essa disciplina é apresentado na Figura 5.10.

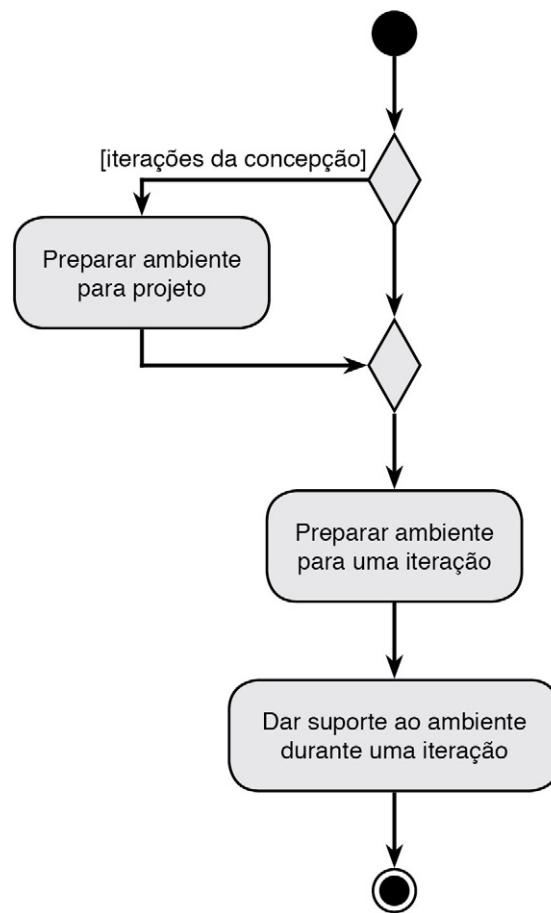


Figura 5.10 Workflow da disciplina de ambiente.

Em relação às atividades do *workflow*, pode-se indicar o seguinte:

- Preparar o ambiente para o projeto*: seus objetivos incluem avaliar a organização de desenvolvimento atual e o *status* das ferramentas de suporte, esboçar um rascunho do caso de desenvolvimento, produzir uma lista de ferramentas candidatas e uma lista de artefatos-chave candidatos ao projeto específico.
- Preparar o ambiente para uma iteração*: seus objetivos incluem refinar ou completar o caso de desenvolvimento para a iteração corrente, preparar e, se necessário, personalizar as ferramentas, verificar se as ferramentas foram preparadas e instaladas corretamente, produzir o conjunto de *templates* específicos a serem usados na iteração e treinar as pessoas para usar as ferramentas e compreender o caso de desenvolvimento. Além disso, para cada iteração, pode ser necessário produzir orientações sobre como fazer modelagem de negócio e de casos de uso, *design*, programação, manuais de estilo, interfaces com usuário, testes e ferramentas.
- Dar suporte ao ambiente durante uma iteração*: é necessário porque, apesar das instruções e do treinamento, os desenvolvedores podem precisar de orientação à medida que executarem seu trabalho durante a iteração.

5.4 AUP – Agile Unified Process

O *Agile Unified Process*, ou AUP, é uma versão simplificada do RUP desenvolvida por Ambler (Ambler & Jeffries, 2002). AUP aplica técnicas ágeis como desenvolvimento dirigido por testes (*TDD – Test Driven Development*), modelagem ágil e refatoração.

Ao contrário do RUP, o AUP tem apenas sete disciplinas:

- Modelagem*: entender o negócio da empresa através de modelos produzidos, que buscam identificar possíveis soluções para os problemas. Um guia de modelagem AUP usando diagramas UML pode ser encontrado na página do AUP na internet.¹³

¹³Disponível em: <www.agilemodeling.com/style/>. Acesso em: 21 jan. 2013.

- b) *Implementação*: transformar os modelos em código executável e criar testes de unidade.
- c) *Teste*: efetuar testes de integração, de sistema e de aceitação para garantir que o sistema tenha qualidade e implemente os requisitos adequados corretamente.
- d) *Implantação*: planejar as entregas de sistema para torná-lo acessível para os usuários.
- e) *Gerenciamento de configuração*: gerenciar as versões e o acesso aos artefatos do projeto.
- f) *Gerenciamento de projeto*: controlar as atividades de projeto, garantindo que elas sejam atribuídas às pessoas certas e executadas no prazo.
- g) *Ambiente*: dar suporte à equipe, garantindo que as ferramentas, guias e padrões estejam disponíveis quando necessário.

O Agile Unified Process baseia-se na seguinte filosofia:

- a) *Sua equipe sabe o que está fazendo*: as pessoas dificilmente gostarão de ler instruções detalhadas sobre como fazer seu trabalho, mas, de tempos em tempos, precisarão de alguma instrução para guiá-las. É importante identificar o ponto de equilíbrio entre as necessidades da equipe e a saturação com instruções demasiadas.
- b) *Simplicidade*: tudo deve poder ser descrito com simplicidade em algumas poucas páginas, e não em milhares delas.
- c) *Agilidade*: o AUP aceita e se conforma aos princípios ágeis.
- d) *Foco em atividades de alto valor*: o foco está nas atividades que realmente produzem valor, não em qualquer coisa que eventualmente poderia ser feita.
- e) *Independência de ferramentas*: a equipe pode usar qualquer ferramenta para desenvolver o projeto, desde que esteja bem adaptada às suas necessidades.
- f) *AUP pode ser ajustado para satisfazer suas necessidades*: como outros processos ágeis, o AUP não possui cláusulas pétreas e pode ser ajustado de acordo com a necessidade.

Um refinamento importante do AUP em relação ao RUP é que o AUP distingue dois tipos de iteração:

- a) *De desenvolvimento*: têm como objetivo desenvolver resultados para garantia de qualidade, prova de conceito ou redução de risco.
- b) *De produção*: têm como objetivo a produção de uma nova versão de código potencialmente entregável.

Os principais papéis AUP são os seguintes:

- a) *Administrador ágil de base de dados*: trabalha colaborativamente com a equipe para implementar, testar e evoluir a base de dados.
- b) *Modelador ágil*: cria e desenvolve modelos de baixa cerimônia, ou seja, modelos que sejam bons o suficiente, mas não excessivamente detalhados.
- c) *Gerente de configuração*: é o responsável pelo sistema de gerência de mudança e configuração.
- d) *Implantador*: é o responsável pela instalação do sistema.
- e) *Desenvolvedor*: desenvolve o produto, escrevendo seu código.
- f) *Engenheiro de processo*: cuida do processo de desenvolvimento e de artefatos anexos, como padrões, guias, templates etc.
- g) *Gerente de projeto*: gerencia e protege os membros da equipe, constrói e coordena relacionamentos com os interessados, planeja, gerencia e aloca recursos, define prioridades e mantém a equipe focada.
- h) *Revisor*: avalia os produtos de trabalho, fornecendo feedback para a equipe.
- i) *Redator técnico*: é o responsável pela escrita de documentos e textos necessários em interfaces do sistema.
- j) *Gerente de teste*: é o responsável pelo planejamento e gerenciamento dos esforços de teste.
- k) *Testador*: é o responsável pela criação e execução dos testes.
- l) *Especialista em ferramentas*: é o responsável pela instalação e manutenção das ferramentas necessárias ao processo de desenvolvimento.

A filosofia de produção de artefatos AUP é minimalista, ou seja, deve-se gerar o mínimo necessário de artefatos para produzir um sistema com qualidade. Os seguintes artefatos são, porém, considerados necessários pelo método em ordem de prioridade:

- a) *Sistema*: software, hardware e documentação necessários para o cliente.
- b) *Código-fonte*: código-fonte do programa, que segue um dos padrões de codificação existentes.¹⁴
- c) *Conjunto de testes de regressão*: sequência (suíte) de testes de unidade, integração e sistema que possa ser executada sempre que houver alguma modificação no código.
- d) *Scripts de instalação*: código necessário para automatizar a instalação do produto no ambiente de produção.
- e) *Documentação de sistema*: toda a documentação que possa ser útil para o processo de operação e evolução do sistema.
- f) *Notas de versão*: resumo das características da atual versão do sistema, incluindo problemas conhecidos quando for o caso.
- g) *Modelo de requisitos*: modelos gerados durante o processo de análise, incluindo o modelo de negócio, casos de uso, modelo conceitual, entre outros.
- h) *Modelo de design*: modelos gerados durante o processo de *design*, incluindo modelos de segurança, base de dados, arquitetura do software, interface com usuário, entre outros.

A documentação completa referente ao processo AUP pode ser baixada gratuitamente da internet.¹⁵

5.5 OpenUp – Open Unified Process

Anteriormente conhecido como *Basic Unified Process* (BUP) ou *OpenUP/Basic*, o *Open Unified Process* (OpenUP) é uma implementação aberta do UP desenvolvida como parte do *Eclipse Process Framework* (EPF).¹⁶

A primeira versão do modelo, conhecida como BUP, foi originada pela IBM, que abriu a definição de uma versão mais leve do RUP. Entre 2005 e 2006, essa versão foi abraçada pela Fundação Eclipse¹⁷ e passou a ser um de seus projetos.

O OpenUP aceita, embora de forma simplificada, a maioria dos princípios UP. Porém, é um método independente de ferramenta e de baixa cerimônia, ou seja, não são exigidos grande precisão e detalhes nos documentos.

O processo se baseia em quatro princípios:

- a) *colaboração* para alinhar interesses e compartilhar entendimentos;
- b) *evolução* para continuamente obter *feedback* e melhorar;
- c) *balanceamento* de prioridades que competem entre si de forma a maximizar valor para os interessados;
- d) *foco* na articulação da arquitetura.

O ciclo de vida também é estruturado em quatro fases, como no UP. Essas fases são igualmente divididas em iterações, mas aqui as equipes se auto-organizam para planejar cada uma delas. O esforço pessoal é organizado em microincrementos, que representam pequenas unidades de trabalho que produzem um ritmo mais fino e mensurável para o projeto. A Figura 5.11 apresenta, de forma esquemática, o ciclo de vida do OpenUP em seus três níveis: projeto, iteração e esforço pessoal.

Em relação ao RUP, a maioria das práticas opcionais foi eliminada e outras foram mescladas. O resultado é um processo mais simples, mas ainda fiel aos princípios de RUP.

Em OpenUP, cada prática pode ser adotada como um princípio independente que agrupa valor à equipe. Dessa forma, as práticas podem ser adotadas de forma progressiva, ao longo de vários projetos. Além disso, como nos modelos de software livre de código aberto, novas práticas podem ser sugeridas pela comunidade e passar a ser adotadas se houver interesse de outros.

OpenUP sugere um microciclo de vida para a iteração que é semelhante ao mostrado na Figura 5.12.

OpenUP é acessível gratuitamente a partir do site da Fundação Eclipse.¹⁸

¹⁴Disponível em: <www.ambysoft.com/essays/codingGuidelines.html>. Acesso em: 21 jan. 2013.

¹⁵Disponível em: <www.ambysoft.com/unifiedprocess/agileUP.html>. Acesso em: 21 jan. 2013.

¹⁶Disponível em: <www.methodsandtools.com/archive/archive.php?id=69p2>. Acesso em: 21 jan. 2013.

¹⁷Disponível em: <epf.eclipse.org/wikis/openup/index.htm>. Em português: <epf.eclipse.org/wikis/openuppt/index.htm> . Acesso em: 21 jan. 2013.

¹⁸Disponível em: <epf.eclipse.org/wikis/openup/>. Acesso em: 21 jan. 2013.

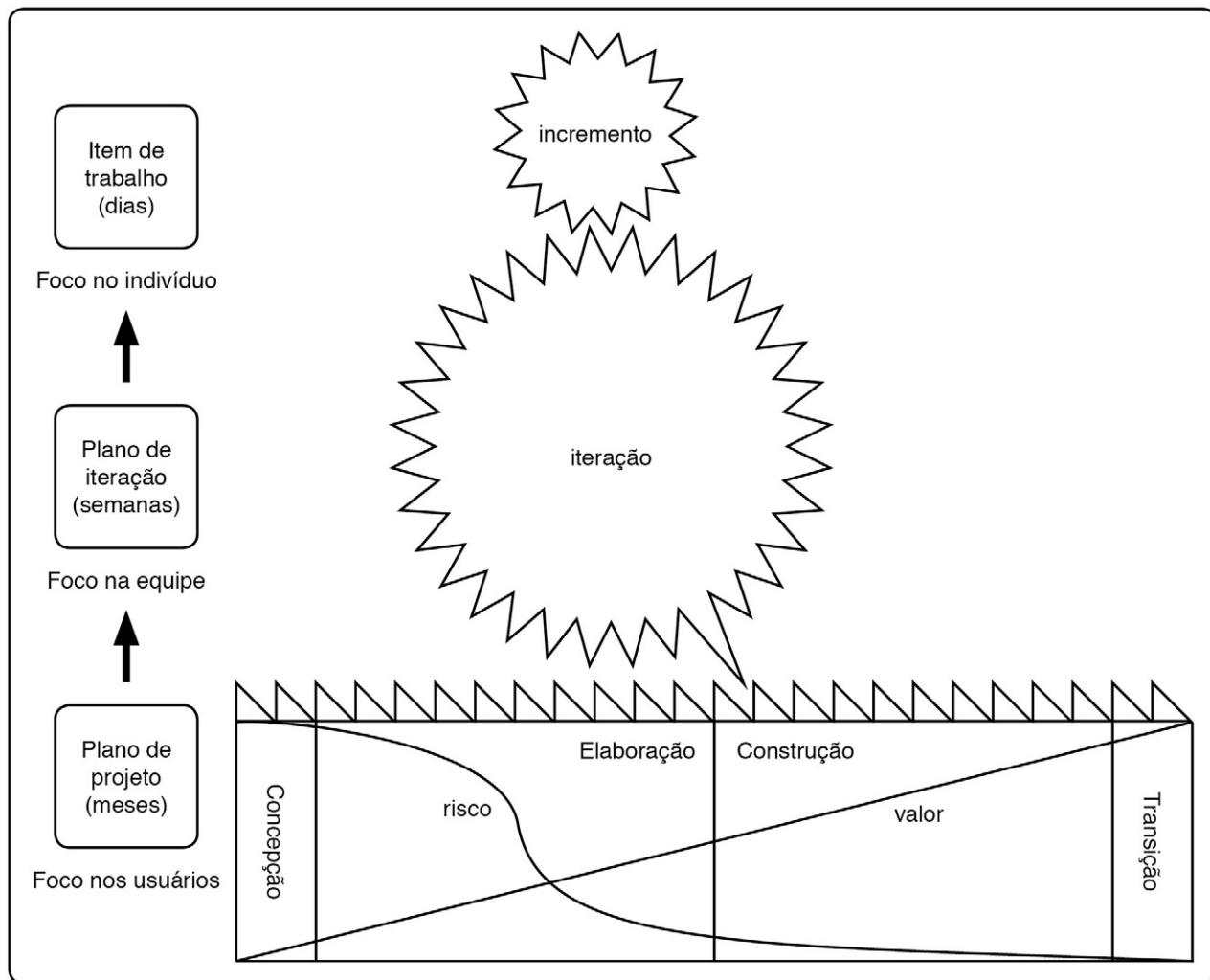


Figura 5.11 Ciclo de vida OpenUP.

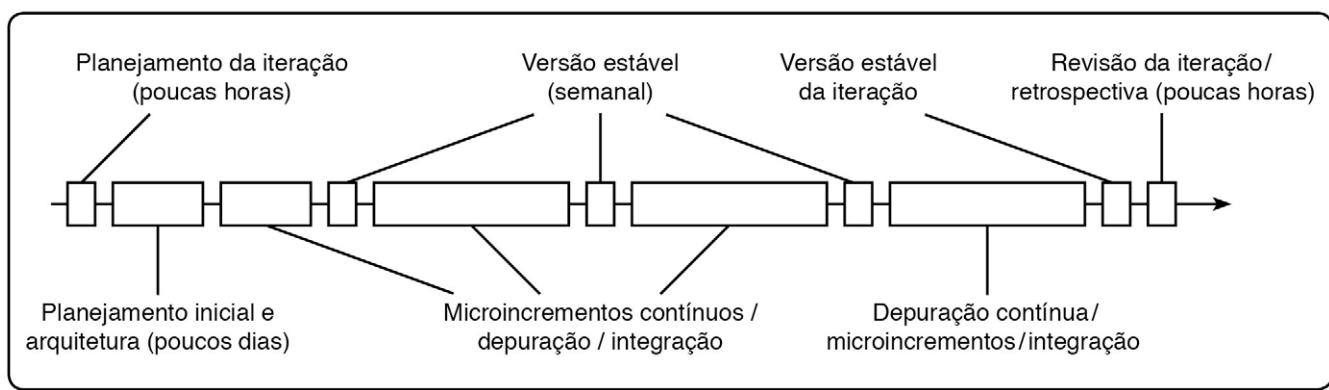


Figura 5.12 Modelo de ciclo de vida de uma iteração segundo OpenUP.

5.6 EUP – Enterprise Unified Process

O *Enterprise Unified Process*, ou EUP, foi definido por Ambler e Constantine em 1999 e, posteriormente, refinado por Ambler, Nalbone e Vizdos (2005). O modelo EUP vê o desenvolvimento de software não apenas como um projeto a ser executado, mas como algo intrínseco ao ciclo de vida da própria empresa.

EUP foi proposto como uma extensão ao modelo RUP para prover, além das fases de RUP, duas novas fases para tratar a evolução ou suporte ao sistema e a aposentadoria do sistema. Além dessas duas fases, várias novas disciplinas relacionadas à empresa foram adicionadas. As novas fases introduzidas no modelo EUP são as seguintes:

- a) *Produção*: em geral, o desenvolvimento de sistemas não acaba quando o produto é entregue e colocado em uso. A fase de produção trata exatamente das atividades que ocorrem após a transição, incluindo suporte, correção e ajustes e evolução do sistema.
- b) *Aposentadoria*: a fase de aposentadoria consiste na retirada de um sistema de operação. É a fase final de qualquer sistema. Sistemas antigos retirados de operação para serem substituídos por sistemas novos podem causar sérios danos à empresa se o processo não for gerenciado adequadamente.

Além das nove disciplinas de RUP, o modelo introduz uma nova disciplina de projeto, chamada *operação e suporte*, além de um novo grupo, as *disciplinas de empresa*, que são sete:

- a) *Modelagem de negócio de empresa*: o RUP já apresenta uma disciplina de modelagem de negócio, mas do ponto de vista do sistema a ser desenvolvido. A modelagem de negócio de empresa do EUP é mais abrangente, incluindo todos os processos da empresa e, dessa forma, relações entre diferentes sistemas.
- b) *Gerenciamento de portfólio*: um portfólio é uma coleção de projetos de software em andamento e concluídos. Trata-se de projetos de alto risco e alto retorno, de baixo risco e baixo retorno que devem ser gerenciados como ações para satisfazer as necessidades estratégicas da empresa.
- c) *Arquitetura de empresa*: a arquitetura de empresa define como ela trabalha. Essa disciplina é especialmente útil se a empresa possui muitos produtos de software. Deve haver consistência na forma como eles são desenvolvidos, negociados e entregues. A arquitetura de empresa é a chave para compreender isso como um processo.
- d) *Reúso estratégico*: o reúso estratégico vai além do reúso que se consegue dentro de um único projeto. Ele se estende entre diferentes projetos. Esse tipo de reúso costuma produzir mais valor do que o reúso simples dentro de um único projeto.
- e) *Gerenciamento de pessoas*: essa disciplina define uma abordagem para gerenciamento de recursos humanos da área de tecnologia de informação. É preciso gerenciar o pessoal, contratar, demitir, substituir, alocar pessoas a projetos e investir em seu crescimento.
- f) *Administração de empresa*: essa disciplina define como a empresa cria, mantém, gerencia e entrega produtos físicos e informações de forma segura.
- g) *Melhoria de processo de software*: essa disciplina trata da adequação e evolução do processo de software para a empresa como um todo, não apenas da adequação do processo a cada projeto.

O objetivo principal da disciplina de *operação e suporte* é manter o produto funcionando no ambiente de produção. Deve-se garantir que o software funcione corretamente, que a rede esteja funcionando, que os dados sejam guardados em *backups* e possam ser restaurados se necessário. Planos de recuperação de desastre devem ser criados e, se for o caso, executados (Ambler S. W., 2010).¹⁹

5.7 OUM – Oracle Unified Method

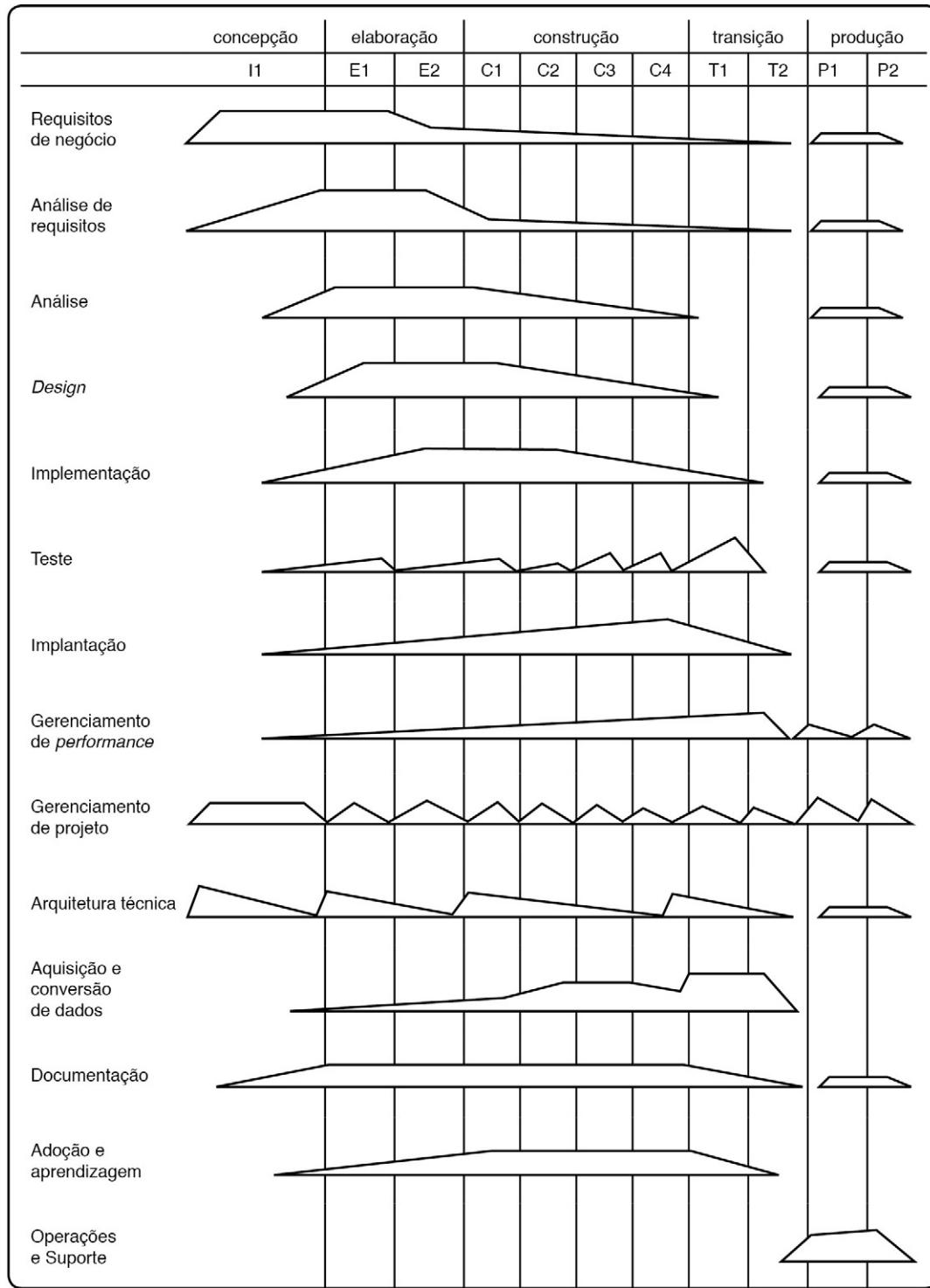
O *Oracle Unified Method*, ou OUM (Oracle, 2009),²⁰ é um *framework* de processo de desenvolvimento de software iterativo e incremental adequado a uso com produtos Oracle: bancos de dados, aplicações e *middleware*.

OUM é uma implementação do Processo Unificado que suporta, entre outras características, *Service Oriented Architecture* (SOA), *Enterprise Integration*, software personalizado, gerenciamento de identidade (*Identity Management*, IdM), governança, risco e adequação (Governance, Risk and Compliance, GRC), segurança de banco de dados, gerenciamento de *performance* e inteligência empresarial.

A Figura 5.13 apresenta esquematicamente o ciclo de vida OUM, que é composto por cinco fases e 14 disciplinas (Oracle, 2007).

¹⁹Disponível em: <www.enterpriseunifiedprocess.com/essays/operationsAndSupport.html>. Acesso em: 21 jan. 2013.

²⁰Disponível em: <www.oracle.com/consulting/library/briefs/oracle-unified-method.pdf>. Acesso em: 21 jan. 2013.

**Figura 5.13 Ciclo de vida OUM.**

Em relação ao RUP, observa-se a introdução da fase de produção, com o mesmo significado que tem no EUP, e também a eliminação das disciplinas de ambiente (visto que o ambiente já é definido pelas ferramentas Oracle) e gerenciamento de configuração (também já disponibilizado pelas ferramentas).

Além disso, a disciplina RUP de Análise e *Design* é dividida em duas partes. Novas disciplinas são também acrescentadas, como:

- Gerenciamento de performance.

- b) Arquitetura técnica.
- c) Aquisição e conversão de dados.
- d) Documentação.
- e) Adoção e aprendizagem.
- f) Operações e suporte.

OUM é, ao mesmo tempo, uma instanciação do Processo Unificado e um modelo orientado a ferramentas (Seção 3.13). Em resumo, as disciplinas OUM podem ser assim caracterizadas:

- a) *Requisitos de negócio*: os requisitos de negócio da nova aplicação ou de sua evolução são identificados e modelados. As principais saídas dessa disciplina são objetivos e metas de negócio, e a lista de requisitos funcionais e não funcionais.
- b) *Análise de requisitos*: os requisitos são organizados em um modelo de casos de uso. As principais saídas são modelo de caso de uso, protótipos de interface e descrição em alto nível da arquitetura do sistema.
- c) *Análise*: o modelo de casos de uso é usado para a descoberta dos conceitos e seus atributos e associações, formando assim o modelo conceitual, ou modelo de classes de análise. As principais saídas são o modelo conceitual e a revisão da arquitetura de sistema.
- d) *Design*: o modelo de análise é instanciado em um modelo de *design* derivado da arquitetura inicial, que, além de informações sobre classes e funcionalidades, vai indicar os aspectos técnicos da implementação, a maioria dos quais é mencionada nos requisitos suplementares (não funcionais). As principais saídas são modelo de *design*, arquitetura detalhada e modelo de implantação (*deployment*).
- e) *Implementação*: visa produzir os elementos de código necessários para o funcionamento do sistema, bem como os testes de unidade. As principais saídas são a versão do software, pronta para os testes de sistema, e a arquitetura do software, enriquecida com os aspectos de implementação.
- f) *Teste*: os testes de sistema e de aceitação devem ser feitos para garantir a qualidade e conformação do sistema aos requisitos. As principais saídas são os casos de teste e a versão do sistema validada.
- g) *Gerenciamento de performance*: são atividades integradas de garantia de qualidade da aplicação em relação aos requisitos suplementares de *performance*.
- h) *Arquitetura técnica*: sua meta é criar uma arquitetura que dê suporte à visão de negócios da empresa. Ela é fundamental para sistemas distribuídos e não triviais, como os sistemas com grande número de acessos.
- i) *Aquisição e conversão de dados*: em geral, novos sistemas substituirão outros, sejam manuais, sejam informatizados, e seus dados precisarão ser adquiridos ou convertidos de alguma forma. Normalmente, não é um processo simples, por isso, OUM acrescenta essa disciplina.
- j) *Documentação*: as ferramentas Oracle dão suporte à produção de documentação-chave ao longo do projeto.
- k) *Adoção e aprendizagem*: focam o uso e a aceitação de novas práticas associadas às novas ferramentas ou à evolução das antigas.
- l) *Transição*: inclui as atividades necessárias para a instalação do produto.
- m) *Operações e suporte*: essa disciplina trata do monitoramento e resposta aos problemas do sistema, atualização e correção de defeitos.

Uma das características do OUM é que não existe um fim abrupto após a fase de transição. A fase de operação é vista como a continuação natural do projeto, e vários requisitos de baixa prioridade que eventualmente não foram implementados até a fase de transição poderão sê-lo durante a operação.

5.8 RUP-SE – Rational Unified Process-Systems Engineering

O RUP-SE é uma extensão do modelo RUP para Engenharia de Sistemas (Cantor, 2003)²¹. Em outras palavras, é uma versão de RUP especialmente adequada para o desenvolvimento de sistemas de grande porte, envolvendo software, hardware, pessoas e componentes de informação.

²¹Disponível em: <www.ibm.com/developerworks/rational/library/content/RationalEdge/aug03/f_rupse_mc.pdf>. Acesso em: 21 jan. 2013.

Inclui um *framework* de arquitetura que permite considerar o sistema a partir de diferentes perspectivas (lógica, física, informacional etc.). Isso pode ser bastante útil quando se deseja demonstrar ou discutir aspectos de um sistema com diferentes interessados (cliente, analistas programadores, engenheiro de informação etc.).

O RUP-SE é especialmente adequado a projetos:

- a) Que são grandes o suficiente para comportar várias equipes de desenvolvimento trabalhando em paralelo.
- b) Que necessitam de desenvolvimento concorrente de hardware e software.
- c) Cuja arquitetura é impactada por questões relativas à implantação.
- d) Que incluem a reengenharia de uma infraestrutura de tecnologia de informação para dar suporte à evolução do negócio.

O *framework* é disponibilizado como um *plugin*, ou anexo, ao modelo RUP original.



PART E 2

PLANEJAMENTO E GERÊNCIA DE PROJETOS

Esta parte deste livro aborda os seguintes tópicos do SWEBOK: Gerenciamento de Engenharia de Software, Gerenciamento de Configuração de Software e outros aspectos de Ferramentas e Métodos de Engenharia de Software que não foram abordados na Parte 1.

A área de gerenciamento, neste livro, é subdividida em duas grandes áreas: *planejamento* e *gerenciamento* propriamente dito. Assim, o Capítulo 6 inicia esta parte apresentando os conceitos relacionados ao *planejamento* de um projeto de software.

O planejamento, em software, necessita de *estimativas de esforço* que são muito particulares dessa área. O Capítulo 7, portanto, apresenta técnicas para que um planejador consiga calcular, antes de iniciar um projeto, quanto tempo vai levar para ele ser concluído e quanto vai custar. Dessa forma, minimizará um dos maiores problemas nessa indústria, que é a imprevisibilidade, já que em geral os projetos atrasam e custam mais do que o previsto.

O Capítulo 8 apresenta outro subtema da área de planejamento e gerenciamento, que é o tratamento dos riscos de projeto. É sabido que um projeto falha em atingir seus objetivos por conta de riscos que não são devidamente tratados. Assim, esse capítulo vai mostrar como o assunto pode ser abordado de forma organizada para que os riscos sejam mantidos sob controle e os projetos possam ser bem-sucedidos.

Por fim, o Capítulo 9 vai apresentar os aspectos de condução de um projeto de software, ou seja, de seu *gerenciamento*. Entre outras coisas, esse capítulo vai apresentar técnicas para que um gerente de projeto consiga manter seu projeto nos trilhos e possa se recuperar de desastres, caso eles ocorram, da forma mais organizada possível.

O Capítulo 10 trata de um subaspecto da gerência de processo de software que merece atenção especial: o *gerenciamento de mudança e configuração*. Essa prática deve ser fortemente incentivada nas empresas de software, porque permite aumento de produtividade e segurança no processo de desenvolvimento, reduzindo riscos inerentes importantes do projeto.

Planejamento

Este capítulo apresenta os principais conceitos de planejamento de projeto de software, iniciando com algumas reflexões sobre *seleção de projetos* (Seção 6.1), um passo importante a ser tomado antes de se iniciar qualquer ação. Depois serão conceituados o *termo de abertura* (Seção 6.2) de um projeto e sua *declaração de escopo* (Seção 6.3). Na sequência, será mostrado como *planejar um projeto* que adota um modelo de processo iterativo (Seção 6.4), ou seja, o planejamento de longo prazo, e também como *planejar um ciclo iterativo* (Seção 6.5), ou planejamento detalhado de curto prazo.

Já foi visto que o desenvolvimento de software e as atividades relacionadas estruturam-se a partir de um modelo de ciclo de vida, escolhido pelo engenheiro de software para servir à empresa. A partir desse modelo de ciclo de vida, a empresa deve instanciar um processo próprio de desenvolvimento a ser seguido e constantemente aprimorado pela equipe de desenvolvimento, sob supervisão ou orientação do engenheiro de software.

Cabe agora discutir a prática de um processo para produzir um produto, ou seja, como planejar e executar um projeto de desenvolvimento de software.

Neste capítulo será considerado que o ciclo de vida utilizado é iterativo. Assim, dois níveis de planejamento serão abordados:

- a) Planejamento de projeto, de longa duração.
- b) Planejamento de iteração, de curta duração e mais detalhado.

A literatura de planejamento de projetos em geral não considera esses dois níveis de planejamento, que, embora típicos de projetos de desenvolvimento de software, não são comuns em outras áreas.

6.1 Seleção de Projetos

Uma empresa de desenvolvimento de software vai executar um projeto que normalmente servirá a outra empresa ou grupo de usuários. Em geral, existe mais de uma possibilidade de projeto, e nem sempre todos eles podem ser desenvolvidos. Assim, a empresa desenvolvedora deverá pesar alguns pontos antes de decidir iniciar um projeto:

- a) A empresa tem competência para desenvolver esse tipo de produto?
- b) A empresa está dando conta dos projetos atuais, ou seja, tem folga operacional para assumir um novo projeto?

- c) O cliente é conhecido e confiável?
- d) O produto dará um bom retorno financeiro?

Essas e outras perguntas normalmente são avaliadas pela gerência superior da empresa antes de ela assumir um compromisso para o desenvolvimento de um projeto.

De outro lado, a empresa cliente não tem recursos ilimitados, e os projetos de desenvolvimento de sistemas poderão competir entre si ou com outros projetos que necessitem de investimento. Assim, o compromisso da empresa cliente com o projeto de desenvolvimento de software possivelmente será afetado pelos seguintes fatores (Xavier, 2011):¹

- a) Retorno financeiro em relação ao investimento.
- b) Grau de incremento da participação da empresa no mercado.
- c) Melhoria da imagem da empresa.
- d) Utilização de capacidade ociosa.
- e) Aquisição de novas tecnologias.

As empresas clientes tenderão a pontuar essas e outras questões antes de se comprometerem com o desenvolvimento de um projeto.

No caso de desenvolvimento de software para o mercado em geral (COTS – *Commercial off the Shelf*), as mesmas questões consideradas pela empresa cliente deverão ser consideradas pela empresa desenvolvedora, já que ela é que vai investir seus recursos para gerar um produto que poderá ter ou não sucesso.

6.2 Termo de Abertura

Havendo comprometimento entre as duas empresas, ou a decisão da empresa desenvolvedora de COTS de que o projeto será iniciado, isso deve ser oficializado em um termo de abertura (*project charter*).

Segundo o PMBOK (PMI, 2004), o termo de abertura deverá conter ou referenciar documentos externos com as seguintes informações:

- a) Objetivo e justificativa do projeto.
- b) Descrição em alto nível do projeto.
- c) Requisitos de alto nível que satisfazem os principais interessados.
- d) Nomeação do gerente de projeto e definição do nível de autoridade conferida (Pode usar os recursos sem aprovação superior? Pode contratar pessoal?).
- e) Cronograma de marcos (*milestones*) resumido.
- f) Definição dos papéis e responsabilidades das partes interessadas.
- g) Organização funcional do projeto.
- h) Premissas ou hipóteses (são perguntas para as quais ainda não se tem resposta, mas que são aceitas, a princípio, para iniciar o projeto. Por exemplo, haverá um especialista disponível na tecnologia X?).
- i) Restrições.
- j) Estudo de viabilidade (*business case*) indicando o retorno previsto, seja ele financeiro ou não.
- k) Orçamento previsto em linhas gerais.

O termo de abertura deverá ser aprovado e assinado por um gerente de nível superior ao gerente de projeto, pois isso é o que lhe dará autoridade para iniciar o projeto.

6.3 Declaração de Escopo

Inicialmente, o planejador de um projeto deve estabelecer quais são seus objetivos finais. O produto nem sempre é apenas o software funcionando; outros elementos costumam ser necessários e desejáveis.

¹Disponível em <www.administracaovirtual.com/administracao/downloads/apostilas/Apostila-Gerencia-Escopo-Magno-FGV.pdf>.

Sem definir claramente aonde o projeto vai chegar, é muito difícil estabelecer um bom plano. Como escolher o caminho, se não se sabe aonde se quer chegar? Infelizmente, muitos planejadores de projetos se esquecem dessa importante etapa. Por exemplo, o projeto termina com a entrega do software ou com a confirmação de sua correta utilização pelo cliente?

O objetivo de um projeto (e também das iterações) deve ser sempre um conjunto de artefatos, ou seja, coisas palpáveis. Um objetivo não pode ser descrito como “executar tal ação”, porque isso não define um artefato palpável. “Gerar tal diagrama ou tal relatório” seria muito mais adequado nesse sentido, ou ainda “implementar este e aquele requisitos”.

Segundo Xavier (2011), a declaração de escopo do projeto deve conter as seguintes informações:

- a) *Descrição do produto do projeto*: embora o termo de abertura já contenha uma definição do produto em alto nível, a declaração de escopo deverá refinar essa descrição. É importante mencionar que, normalmente, a declaração não pode relacionar características novas em relação ao termo de abertura. Se for necessária uma alteração de escopo em relação ao inicialmente previsto, isso deverá ser negociado entre as partes.
- b) *Principais entregas do projeto*: devem ser definidas as principais entregas do projeto, ou seja, os momentos em que o cliente estará recebendo algum tipo de entrega dos desenvolvedores. Normalmente, trata-se de versões implementadas do sistema, mas essa lista poderá incluir outros itens, como projeto, manuais, discos de instalação, treinamento etc.
- c) *Objetivos do projeto*: itens quantificáveis que serão usados para determinar se o projeto foi um sucesso ou não. Os objetivos do projeto devem incluir pelo menos métricas relacionadas a prazo, custo e qualidade do produto. Objetivos não quantificáveis (por exemplo, “cliente satisfeito” ou “sistema fácil de usar”) representam um fator de alto risco para a determinação do sucesso do projeto. Os objetivos devem ser claramente avaliáveis a partir de uma métrica definida. Devem ser evitados a todo custo objetivos vagos e de avaliação subjetiva, como “desenvolver tecnologia de última geração”.
- d) *Critérios de aceitação do produto*: é preciso definir o processo e os critérios para que o produto, como um todo, seja aceito, e o projeto, finalizado.

Outras informações poderão ser adicionadas à declaração de escopo, se houver necessidade (por exemplo, principais riscos, tecnologias a serem usadas etc.). A declaração de escopo é o documento-base em que deve haver concordância entre o cliente e o gerente de projeto para que, a partir dele, o projeto como um todo possa ser planejado.

É importante mencionar que nesse momento, normalmente, ainda não foi feita uma análise de requisitos, portanto as informações aqui contidas são fruto de entendimentos prévios. Entende-se que a análise de requisitos que virá depois deverá aprofundar o escopo, mas não aumentá-lo em abrangência. Por exemplo, na análise de requisitos pode-se detalhar como será feito o processo de venda, mas, se não estava prevista a implementação de uma folha de pagamento na declaração de escopo, então, a necessidade de inclusão desse item tornará necessária a renegociação do escopo com o cliente.

6.4 Planejamento de Projeto com Ciclos Iterativos

O objetivo do planejamento de projeto é criar um *plano* para o projeto como um todo. Entre outras coisas, é importante que o responsável por esse planejamento utilize as melhores ferramentas possíveis para avaliar a quantidade de esforço a ser despendido no projeto. Tal estimativa poderá dar origem tanto ao cronograma geral do projeto quanto à estimativa de seu custo total.

Considera-se que a declaração de escopo já definiu os objetivos do projeto e os critérios de aceitação do produto. Assim, as atividades necessárias ao planejamento de um projeto são as seguintes:

- a) Estimar o *esforço total* para realizar o projeto.
- b) Em função do esforço total, calcular o *tempo linear* necessário e o *tamanho médio da equipe*.
- c) Estimar a *duração* e o *esforço* nas diferentes fases do projeto.
- d) Estimar a *duração* e o *número dos ciclos iterativos*.

No Capítulo 7 são apresentadas algumas técnicas para estimar o esforço necessário para desenvolver um projeto de software, bem como para estimar seu tempo linear e o tamanho médio da equipe. As subseções seguintes apresentam um detalhamento das outras duas atividades mencionadas.

6.4.1 ESTIMAÇÃO DA DURAÇÃO E DO ESFORÇO NAS DIFERENTES FASES DO PROJETO

Se o modelo de processo utilizado for baseado no UP, após estimar o esforço total do projeto, sua duração linear e o tamanho médio da equipe (Capítulo 7), pode-se tentar refinar um pouco mais essa estimativa com relação às quatro fases do UP. O tamanho médio da equipe, por exemplo, não significa que sempre o mesmo número de desenvolvedores estará trabalhando no projeto. Em geral, há mais pessoas trabalhando nas fases de elaboração e construção do que nas fases de concepção e transição.

A Figura 6.1 indica um perfil aproximado de tempo e esforço despendido em cada uma das fases do UP. Evidentemente, esse perfil pode ser alterado de acordo com as características de cada projeto ou das ferramentas de automatização de projeto, geração de código e teste que se utilize.

Essa figura considera que um projeto típico, de tamanho e esforço moderados, sem arquitetura predefinida e com poucos riscos críticos pode ser desenvolvido aproximadamente com as seguintes estimativas de tempo e esforço:

- a) Concepção: 10% do tempo e 5% do esforço.
- b) Elaboração: 30% do tempo e 20% do esforço.
- c) Construção: 50% do tempo e 65% do esforço.
- d) Transição: 10% do tempo e 10% do esforço.

Assim, aplicando-se a equação de cálculo de tempo linear da Seção 7.4.4, um projeto típico de desenvolvimento cujo esforço foi estimado em 40 desenvolvedores-mês deverá ter uma duração linear ideal de 8,5 meses.

A duração das fases calculada como a porcentagem de tempo definida acima, aplicada à duração linear do projeto em meses, ficará assim:

- a) Concepção: 10% de 8,5, ou seja, cerca de 0,85 meses.
- b) Elaboração: 30% de 8,5, ou seja, cerca de 2,55 meses.
- c) Construção: 50% de 8,5, ou seja, cerca de 4,25 meses.
- d) Transição: 10% de 8,5, ou seja, cerca de 0,85 meses.

Já o cálculo do tamanho médio da equipe para cada fase deve ser feito da seguinte forma: toma-se o valor do esforço estimado (40 desenvolvedores-mês) e aplica-se a porcentagem de esforço da fase, conforme definido acima. Depois, divide-se o resultado pela duração linear da fase, conforme obtido logo acima. No exemplo, fica-se com:

- a) Concepção: 5% de 40, ou seja, 2 desenvolvedores-mês, o que, dividido por 0,85 meses, dá cerca de 2,35 desenvolvedores em média na fase.
- b) Elaboração: 20% de 40, ou seja, 8 desenvolvedores-mês, o que, dividido por 2,55 meses, dá cerca de 3,13 desenvolvedores em média na fase.

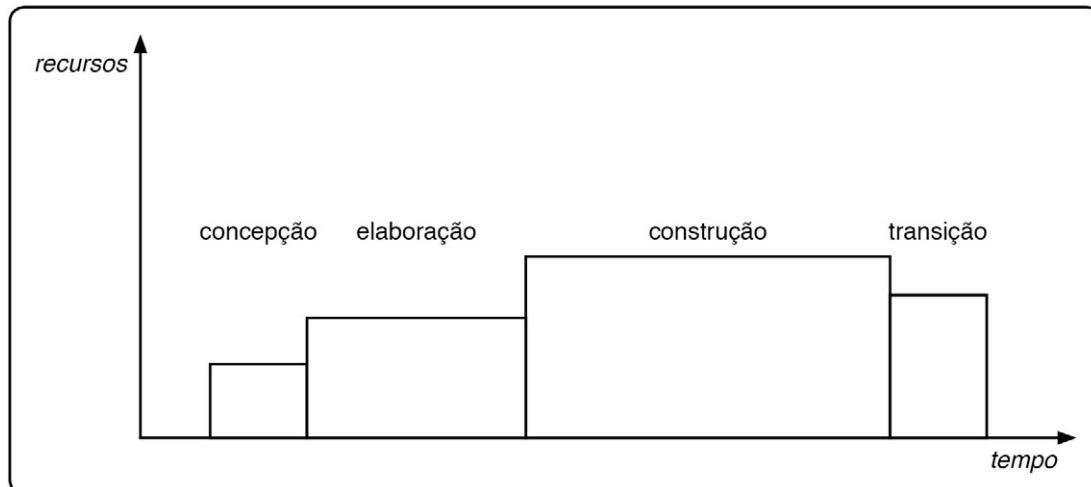


Figura 6.1 Perfil de duração e esforço típicos para um projeto usando UP.

- c) Construção: 65% de 40, ou seja, 26 desenvolvedores-mês, o que, dividido por 4,25 meses, dá cerca de 6,11 desenvolvedores em média na fase.
- d) Transição: 10% de 40, ou seja, 4 desenvolvedores-mês, o que, dividido por 0,85 meses, dá cerca de 4,7 desenvolvedores em média na fase.

Os valores em meses podem ser convertidos em semanas, bastando multiplicá-los por 4 (ou por 3,9, segundo alguns autores). Assim, a concepção teria cerca de 3,5 semanas; a elaboração, 10,2 semanas; a construção, 17 semanas; e a transição, 3,5 semanas.

Um valor fracionado de desenvolvedor como 2,35 indica uma média, ou seja, espera-se que uma parte da fase necessite de 2 desenvolvedores e uma parte menor de 3 desenvolvedores. Isso também pode significar que, se apenas 2 desenvolvedores estiverem disponíveis para a fase, possivelmente será necessário esticar mais o tempo, pois eles não darão conta do trabalho, ou ainda que, se três desenvolvedores estiverem disponíveis, talvez seja possível diminuir um pouco o tempo, pois haverá certa folga.

Contudo, algumas observações podem alterar esse perfil típico (Kruchten, 2003):

- a) Se for necessário mais tempo para estabelecer o projeto, achar financiadores, fazer pesquisa de mercado ou construir provas de conceito, a fase de concepção deve ser prolongada.
- b) Se houver altos riscos técnicos ou de pessoal, ou se houver restrições de desempenho importantes e nenhuma arquitetura prévia definida, então a fase de elaboração deve ser prolongada, porque serão necessários mais ciclos de elaboração para definir a arquitetura e/ou mitigar os riscos conhecidos.
- c) Se essa não for a primeira geração do produto (pode ser um ciclo de evolução) e se não forem feitas maiores alterações na arquitetura, as fases de concepção e elaboração poderão ser encolhidas.
- d) Se o objetivo for atingir o mercado rapidamente por causa de concorrentes ou porque se está criando esse mercado, a fase de construção pode ser encolhida, e a fase de transição, aumentada. Assim, versões executáveis serão liberadas mais cedo e gradativamente no mercado.
- e) Se houver necessidade de uma transição complicada, como substituir um sistema em funcionamento sem interromper os serviços, ou no caso de domínios que exigem certificações ou regulamentos a serem avaliados (medicina, aeronáutica etc.), a fase de transição deve ser aumentada.

Assim, essas e outras questões devem ser avaliadas pelo planejador de projetos, que, a partir da previsão de esforço nominal, vai prever esforços específicos para as diferentes fases em seu projeto específico.

Entretanto, a principal fonte de informação para esse tipo de previsão deve ser sempre o histórico de medições da empresa desenvolvedora, pois, como cada empresa tem seu próprio estilo de trabalho, ferramentas e competências, diferentes valores de esforço nas diferentes fases poderão ser obtidos. Por exemplo, empresas que usam intensivamente modelos baseados em ferramentas e geração automática de código em geral terão uma fase de construção bem menor do que a fase de elaboração.

6.4.2 ESTIMAÇÃO DA DURAÇÃO E NÚMERO DOS CICLOS ITERATIVOS

Uma iteração se inicia com planejamento e termina com uma nova versão do sistema disponibilizada internamente ou até mesmo uma *release* ao cliente. A duração estimada de um ciclo no Processo Unificado ou métodos ágeis costuma variar de 2 a 8 semanas e depende basicamente da complexidade do projeto e da equipe.

Equipes pequenas com até 5 pessoas poderão fazer o planejamento juntas numa manhã de segunda-feira, executar o trabalho ao longo da semana e gerar uma *release* na sexta-feira.

Equipes com mais de 20 pessoas precisarão de mais tempo para distribuir e sincronizar as atividades, até porque a carga de trabalho será naturalmente bem maior. Além disso, a geração da *release* tomará mais tempo, pois haverá um volume maior de partes a serem integradas. Assim, nesse caso, uma iteração de 3 a 4 semanas seria mais recomendável.

Equipes com mais de 40 pessoas precisarão trabalhar em um ambiente muito mais formal e com mais documentação intermediária, de forma que o fluxo de informação será naturalmente mais lento. Dessa forma, um ciclo de 6 a 8 semanas seria recomendável nesse caso.

Outros fatores que podem afetar a duração de um ciclo são os seguintes:

- a) Quanto mais automatização no processo de geração de código e no ambiente de desenvolvimento em geral, mais curtos poderão ser os ciclos.

- b) Quanto mais familiaridade a equipe tiver com o Processo Unificado e com as técnicas de análise e *design*, mais curtos poderão ser os ciclos.
- c) Quanto mais crítico for o fator “qualidade” no desenvolvimento e quanto mais críticas forem as revisões e testes que precisarem ser feitos, mais longos deverão ser os ciclos.

6.4.3 NÚMERO DE ITERAÇÕES

O número de iterações de um projeto dependerá do tempo linear a ser despendido, especialmente nas fases de elaboração e construção, dividido pelo tamanho planejado para as iterações. Por exemplo, um projeto com iterações de duas semanas, cujas fases de elaboração e construção devem durar 6 meses no total (24 semanas), terá 12 ciclos de elaboração e construção.

A quantidade de ciclos de elaboração e de construção dependerá da quantidade de casos de uso complexos a serem abordados, lembrando que os ciclos de elaboração terminam quando a arquitetura estabiliza, ou seja, quando os últimos casos de uso considerados mais críticos e os principais riscos já foram devidamente tratados e incorporados à arquitetura do sistema. A Seção 6.4.1 indica que haverá cinco ciclos de construção (50% do tempo) para cada três ciclos de elaboração (30% do tempo), em média, para projetos nominais, sem características especiais.

Em geral, a fase de concepção não é organizada em ciclos, a não ser que mais de um protótipo seja necessário ou que um número significativo de riscos muito importantes deva ser tratado antes de se iniciar o projeto propriamente dito. Deve-se lembrar, porém, que o objetivo da fase de concepção não é produzir código funcionando, mas gerar rapidamente protótipos (se necessário) que ajudem a compreender melhor os verdadeiros requisitos do sistema.

A fase de transição também não costuma ser organizada em mais de um ou dois ciclos. Apenas transições mais complexas deverão ser organizadas em mais de um ciclo, com diferentes objetivos definidos para cada um deles.

6.4.4 DEFINIÇÃO DOS MARCOS OU ENTREGAS

Uma vez definido o tamanho das iterações, o tamanho da equipe em cada fase e a duração de cada fase (em número de ciclos), o planejador deverá retomar a declaração de escopo para definir os marcos de projeto e as datas de entregas. O Processo Unificado já estabelece marcos-padrão ao final de cada fase, mas convém que no plano de projeto esses marcos, bem como outros momentos importantes do projeto, sejam claramente identificados.

Será considerado novamente o projeto do exemplo das seções anteriores, resumido na Tabela 6.1.

Note que, na tabela acima, os arredondamentos procuraram fazer que as fases de elaboração e construção ficassem com um número par de semanas, em razão do fato de os ciclos iterativos terem sido definidos com duas semanas. O arredondamento do número de desenvolvedores foi feito para cima nas fases de concepção e transição, porque o arredondamento da duração dessas fases foi feito para baixo. De outro lado, o arredondamento do número de desenvolvedores da fase de construção foi feito para baixo porque sua duração foi aumentada em uma semana. Apenas a fase de elaboração teve o número de desenvolvedores e o de duração arredondados para baixo, indicando que poderá haver algum aperto nessa fase ou que o planejador do projeto estima que a elaboração será um pouco mais simples do que em um projeto típico.

TABELA 6.1 Esforço e duração de um projeto típico por fase do UP

| Fase | Duração (semanas) | Duração arredondada | Número médio de desenvolvedores | Número de desenvolvedores arredondado |
|------------|-------------------|---------------------|---------------------------------|---------------------------------------|
| Concepção | 3,5 | 3 | 2,35 | 3 |
| Elaboração | 10,2 | 10 | 3,13 | 3 |
| Construção | 17 | 18 | 6,11 | 6 |
| Transição | 3,5 | 3 | 4,7 | 5 |
| Total | 34,2 | 34 | | |

TABELA 6.2 Um exemplo de plano de projeto simplificado com definição de entregas

| Fase | Prazo (semana) | Desenvolvedores | Entregas |
|------------|----------------|-----------------|---|
| Concepção | 3 | 3 | Modelo de casos de uso preliminar para revisão. |
| Elaboração | 5 | 3 | 20% dos casos de uso de maior risco incorporados na arquitetura. |
| | 7 | 3 | 40% dos casos de uso de maior risco incorporados na arquitetura. |
| | 9 | 3 | 60% dos casos de uso de maior risco incorporados na arquitetura. |
| | 11 | 3 | 80% dos casos de uso de maior risco incorporados na arquitetura. |
| | 13 | 3 | 100% dos casos de uso de maior risco incorporados na arquitetura. |
| Construção | 15 | 6 | 11% dos demais casos de uso incorporados na arquitetura. |
| | 17 | 6 | 22% dos demais casos de uso incorporados na arquitetura. |
| | 19 | 6 | 33% dos demais casos de uso incorporados na arquitetura. |
| | 21 | 6 | 44% dos demais casos de uso incorporados na arquitetura. |
| | 23 | 6 | 55% dos demais casos de uso incorporados na arquitetura. |
| | 25 | 6 | 66% dos demais casos de uso incorporados na arquitetura. |
| | 27 | 6 | 77% dos demais casos de uso incorporados na arquitetura. |
| | 29 | 6 | 88% dos demais casos de uso incorporados na arquitetura. |
| | 31 | 6 | 100% dos demais casos de uso incorporados na arquitetura. |
| Transição | 34 | 5 | Sistema instalado. Migração de dados concluída. |

Um plano simplificado possível para esse projeto seria parecido com o da Tabela 6.2. Note que foram definidos ciclos iterativos de duas semanas, exceto para as fases de concepção e transição.

À primeira vista pode parecer que a fase de elaboração será mais difícil, porque haverá menos desenvolvedores e estes deverão desenvolver uma grande porcentagem de casos de uso mais complexos a cada iteração, mas em geral a quantidade de casos de uso mais críticos em relação ao total de casos de uso de um sistema é menor. Assim, possivelmente haverá menos casos de uso a serem desenvolvidos em cada ciclo da fase de elaboração do que nos ciclos da fase de construção.

Além disso, as porcentagens linearmente crescentes de casos de uso são um tanto simplistas, pois existem casos de uso de complexidade distinta. Um plano de projeto mais realista consideraria porcentagens de *pontos de caso de uso* (Seção 7.5) ou, preferivelmente, os nomes dos casos de uso que deveriam ser desenvolvidos em cada iteração, se estes já forem conhecidos (talvez isso só seja possível no final da fase de concepção), ordenados por prioridade.

6.5 Planejamento de Iteração

Concluído o planejamento do projeto, se este for feito com ciclos iterativos, apenas o primeiro ciclo será planejado detalhadamente de início. Apenas quando esse ciclo estiver em andamento deve-se iniciar o planejamento do segundo ciclo, e assim por diante. Esta seção tratará do planejamento detalhado, ou seja, do planejamento dos ciclos iterativos.

Caso se esteja trabalhando com um método ágil, os objetivos da iteração serão definidos pelas histórias de usuário ou requisitos a serem implementados. No caso do UP, os objetivos de uma iteração poderão ser de três tipos:

- a) Implementar total ou parcialmente um ou mais *casos de uso*.
- b) Mitigar um *risco* conhecido, gerando ou executando um plano de redução de probabilidade, redução de impacto ou ainda de recuperação de desastre.
- c) Implementar total ou parcialmente uma ou mais *modificações* solicitadas. À medida que a arquitetura do sistema evoluir nas iterações, modificações poderão ser solicitadas em função da não adequação aos requisitos ou, ainda, à sua mudança. Incorporar essas solicitações de mudança ao software pode ser um dos objetivos de uma iteração.

Para cada caso de uso, risco ou modificação deve haver uma estimativa total de esforço de desenvolvimento. Os elementos serão selecionados considerando-se em primeiro lugar sua prioridade. A maior prioridade deve ser dada aos elementos mais complexos, de maior risco ou com os quais mais se possa aprender em relação à arquitetura do sistema. A sugestão é escolher em primeiro lugar:

- a) casos de uso que representem os processos de negócio mais críticos para a organização, por exemplo, aqueles através dos quais a organização realiza seus objetivos, como obtenção de lucros;
- b) riscos de alta importância, ou seja, com alto impacto e alta probabilidade de ocorrer;
- c) modificações urgentes, como refatorações da arquitetura.

Considerados os elementos de maior prioridade, outros elementos de prioridade não tão alta, mas com certa afinidade, poderão ser colocados no mesmo ciclo por conveniência. O importante é que o esforço total estimado não ultrapasse a quantidade de desenvolvedor-mês que se pode alocar dentro da duração prevista do ciclo.

Selecionados os elementos a serem tratados no ciclo, deve-se estabelecer claramente qual o objetivo da iteração, ou seja, até que ponto os elementos selecionados deverão ser desenvolvidos. É importante que o objetivo da iteração também seja detalhado a ponto de poderem ser identificados os artefatos que serão produzidos ao final da iteração.

O objetivo da iteração deve ser buscado inicialmente no plano de projeto, mas poderá ser alterado em função de seu *status* (atrasado, por exemplo) e das listas de priorização de casos de uso, riscos e modificações.

Na sequência, deve-se estabelecer a estrutura analítica da iteração (Seção 6.5.1), ou seja, o conjunto de atividades que devem ser executadas para obter os artefatos que constituem o objetivo da iteração. Se o método de desenvolvimento for ágil, esse inventário de atividades será feito pela própria equipe de maneira mais informal.

Contudo, no caso de processos prescritivos, o inventário deve ser obtido a partir da instanciação dos *workflows* das disciplinas necessárias para a iteração. Cada atividade prevista no *workflow* deverá ser atribuída a uma pessoa com capacidade de exercer o papel previsto. As atividades deverão ter sua duração estimada e, em função de suas dependências, um diagrama PERT e/ou Gantt deverá ser construído (Seções 6.5.5 e 6.5.6).

6.5.1 WBS – ESTRUTURA ANALÍTICA DA ITERAÇÃO

A estrutura analítica da iteração (WBS ou *Work Breakdown Structure*) (Tausworthe, 1980) apresenta as atividades que devem ser executadas para se atingirem os objetivos determinados para o projeto (quando se vai planejar o projeto como um todo, atividade por atividade) ou iteração (no caso do planejamento por ciclos). Várias resoluções de trabalho do governo norte-americano exigem a apresentação de uma WBS para a execução de um projeto.

Além da lista de atividades, é importante utilizar um método de estimativa de esforço para prever a duração de cada atividade. A Seção 7.3.3 apresenta um conjunto de estimativas individualizadas por fase para as disciplinas do RUP.

Se for usado um ciclo de vida prescritivo, os *workflows* vão indicar quais são as atividades a serem executadas e quais as dependências entre elas. Dependendo do processo adotado, o *workflow* poderá até indicar formas de estimativa de esforço para cada atividade individual.

Se for usado um método ágil, recomenda-se que a equipe decida quais atividades serão desenvolvidas. Isso não impede que equipes usando métodos ágeis se baseiem em *workflows* existentes, se o grupo entender que isso poderá ser útil ao projeto.

Seja qual for o modelo de processo adotado, a WBS pode ser definida em uma reunião de planejamento com toda a equipe para que as várias visões do projeto sejam pesadas no momento de se estabelecerem atividades e estimar esforço.

É importante que cada atividade caracterize muito bem o produto de trabalho ou artefato de saída a ser entregue ao final. Se um processo for usado, ele próprio vai estabelecer esses artefatos.

A WBS é uma estrutura exaustiva, ou seja, deve incluir todas as atividades necessárias para a execução do projeto ou iteração. A WBS poderá ser estruturada como uma árvore, isto é, as atividades podem ser aglutinadas ou detalhadas estabelecendo-se uma árvore de decomposição entre elas. As atividades nas folhas dessa árvore são as atividades *terminais* e devem seguir a regra 8-80 especificada na subseção seguinte.

É muito importante que o planejador do projeto determine artefatos de saída, e não meramente ações. Atividades devem necessariamente produzir algo físico, e não apenas a execução de ações do responsável ou dos participantes. Tentar modelar um projeto baseado em ações vai necessariamente levar ao detalhamento excessivo das atividades e, consequentemente, à impossibilidade de gerenciar de modo adequado o trabalho. Então, a regra de ouro do planejamento é: “Cada atividade deve gerar pelo menos um produto palpável”.

Estilos de WBS que preveem diferentes estágios de um artefato (por exemplo, versão inicial, versão intermediária e versão final) devem caracterizar exatamente o que esperam de cada uma dessas versões.

Por exemplo, a versão inicial de um documento de requisitos poderia ter apenas uma lista de funções identificadas. Uma versão intermediária desse documento poderia exigir que as funções fossem agrupadas por similaridade e que requisitos não funcionais tivessem sido adicionados. Uma versão final do mesmo documento poderia exigir que estivesse organizado e revisado dentro de determinado padrão e que os requisitos tivessem sido verificados em relação a sua completeza e consistência por algum processo padrão definido.

6.5.1.1 Regra 8-80

Atividades que, de acordo com as estimativas, levarão muito tempo para ser completadas devem ser subdivididas em atividades mais curtas. Atividades estimadas para levar pouco tempo devem ser aglutinadas com outras. A regra 8-80 estabelece que nenhuma atividade terminal deve durar nem mais de 80 horas (duas semanas ou dez dias de trabalho ideais), nem menos de 8 horas (um dia de trabalho ideal).

Não se deve ter atividades com duração muito longa, porque fica muito difícil gerenciá-las e acompanhar seu andamento. Também não se deve ter atividades muito curtas, porque microgerenciá-las pode provocar um *overhead* de gerenciamento que, em vez de ajudar, vai atrapalhar o projeto.

Métodos como XP são ainda mais restritivos em relação ao tamanho das atividades, pois exigem que sua duração seja de um a três dias ideais de trabalho, ou seja, de 8 a 24 horas. Mas, embora isso seja mais restritivo, não contradiz a regra 8-80.

A WBS deve ser organizada, precisa e pequena o suficiente para que possa servir de base para a gerência do projeto durante a iteração sem ser um estorvo.

6.5.1.2 Regra dos Níveis e do Número Total de Atividades

Além de respeitar a regra 8-80, a estruturação de uma boa WBS não deve ter mais de três ou quatro níveis de decomposição de atividades. Os elementos terminais, ou seja, os elementos não decompostos (no nível mais baixo) são também chamados de *pacotes de trabalho*.

O número total de pacotes de trabalho em uma WBS não deve ultrapassar o limite de 200 elementos, embora 100 já seja considerado um número muito alto.

Considerando-se que cada atividade terminal poderá ter no máximo 80 horas, essa regra estabelece que, no pior dos casos, uma iteração ou projeto gerenciável deverá ter $80 \times 200 = 16.000$ horas de trabalho (mas o típico são iterações bem abaixo desse limite). Qualquer projeto ou iteração com carga horária maior do que essa deve necessariamente ser subdividido em projetos ou iterações menores. Nesse sentido, as iterações de duas semanas dos métodos ágeis e do UP garantem que o número de horas total nunca seja muito grande. Com apenas 80 horas de atividade por desenvolvedor em duas semanas, seriam necessários 200 desenvolvedores trabalhando numa iteração para atingir tal limite. Acima disso (por exemplo, 60 pessoas trabalhando em ciclos de 8 semanas, ou seja, uma carga de 19.200 horas por ciclo) seria altamente recomendável a subdivisão do projeto e/ou da equipe.

6.5.1.3 Regra dos 100%

Essa regra estabelece que uma WBS deve incluir 100% de todo o trabalho que deve ser feito na iteração. Nenhum artefato será produzido se não estiver definido como saída de alguma das atividades da WBS e nenhuma atividade deixará de produzir algum artefato de saída.

A regra dos 100% vale em todos os níveis da hierarquia de decomposição da WBS. Além disso, quando uma atividade se decompõe em subatividades, o trabalho definido pela atividade será exatamente igual a 100% do trabalho definido nas subatividades (sempre em termos de artefatos de saída).

6.5.2 Os Dez MANDAMENTOS DA WBS

Xavier (2011) apresenta uma lista com dez recomendações, ou dez mandamentos, para a elaboração de WBS:

- Cobiçarás a WBS do próximo:* o ideal é que uma WBS não seja iniciada do zero. Possivelmente existem projetos anteriores semelhantes ou *templates* do próprio processo para que a WBS seja elaborada a partir de experiências passadas.

- b) *Explicitarás todos os subprodutos, inclusive os necessários ao gerenciamento do projeto:* o subproduto que não estiver na WBS não será desenvolvido. Então, nenhum artefato pode ficar de fora da WBS quando for o momento de construí-lo ou revisá-lo. Se em algum momento um desenvolvedor estiver trabalhando em algo que não contribui para nenhum subproduto da WBS, ele estará trabalhando fora do escopo e será preciso decidir se o subproduto deverá ser incluído na WBS ou a atividade do desenvolvedor deverá ser revista.
- c) *Não usarás os nomes em vão:* não devem ser usados nomes vagos que deixem dúvida sobre o subproduto a ser gerado pela atividade. Devem-se usar substantivos para definir o subproduto, e não verbos. Por exemplo, deve-se usar “relatório de teste do módulo” em vez de “testar o módulo”.
- d) *Guardarás a descrição dos pacotes de trabalho no dicionário da WBS:* o dicionário da WBS é o documento que define cada um dos pacotes de trabalho. Na WBS aparece apenas seu nome curto, mas o dicionário deve descrever esse subproduto com detalhes.
- e) *Decomporás as atividades até o nível de detalhe que permita o planejamento e o controle do trabalho necessários para a entrega do produto:* respeitadas as regras 8-80, dos 100% e dos níveis vistas anteriormente, o planejamento deve criar atividades que efetivamente possam ser verificadas.
- f) *Não decomporás em demasia, de forma que o custo/tempo de planejamento e controle não traga o benefício correspondente:* embora possa haver variações, em geral a regra 8-80 é uma boa medida para evitar que se detalhem demais as atividades da WBS, evitando assim a sobrecarga de trabalho de gerência.
- g) *Honrarás o pai:* cada atividade da WBS decomposta em subatividades funciona como um todo com partes. Deve-se verificar que as subatividades realmente sejam um detalhamento do pacote de trabalho da atividade-pai.
- h) *Decomporás de forma que a soma dos subprodutos das atividades-filhas seja igual a 100% do subproduto da atividade-pai:* aplica-se a regra dos 100%.
- i) *Não decomporás somente um subproduto:* nenhum elemento da WBS deve ter um único filho. Pela regra dos 100%, nesse caso, ele seria igual ao pai. Então por que representar duas vezes o mesmo elemento? Caso isso aconteça, a atividade não deve ser decomposta ou deve-se verificar se não estão faltando outras subatividades que foram esquecidas.
- j) *Não repetirás o mesmo elemento como componente de mais de um subproduto:* nenhuma atividade pode ter dois pais. Podem existir atividades com o mesmo nome, mas, cada vez que elas aparecerem como componentes de outra atividade, serão atividades diferentes, executadas em outro período de tempo e/ou por outro desenvolvedor.

6.5.3 IDENTIFICAÇÃO DOS RESPONSÁVEIS POR ATIVIDADE

Um *workflow* costuma definir que o responsável por uma atividade é um papel, ou seja, uma pessoa com uma ou mais habilidades desejáveis. Quando uma iteração for planejada a partir desse *workflow*, deve-se atribuir as atividades a pessoas reais que atendam ao papel desejado sempre que possível.

Cada atividade da WBS deverá ser atribuída a um ou mais responsáveis. Essas atribuições poderão ter efeito sobre o cronograma de projeto, pois, embora certas atividades possam ser executadas em paralelo, não é possível fazê-lo assim caso estejam atribuídas ao mesmo responsável.

6.5.4 IDENTIFICAÇÃO DOS RECURSOS NECESSÁRIOS E CUSTO

É possível que a maioria das atividades a serem executadas, além de recursos humanos (responsáveis e participantes), também tenha recursos físicos consumíveis ou não consumíveis a serem alocados.

No momento do planejamento da iteração é necessário prever e alocar o uso desses recursos. O custo de uma atividade individual será, portanto, o custo das pessoas que se dedicam a ela somado ao custo dos recursos alocados.

6.5.5 IDENTIFICAÇÃO DAS DEPENDÊNCIAS ENTRE ATIVIDADES

As dependências entre atividades são dadas em função do *workflow* ou identificadas caso a caso pela equipe de planejamento. Em geral, essas dependências existem porque as entradas de uma atividade são as saídas de outra. Não havendo essa condição, as atividades podem ser executadas potencialmente em paralelo.

| (i) | Nome | Duração | Início | Término | Predecessoras |
|-----|---------------------------------------|---------|----------------|----------------|---------------|
| 1 | Desenvolver visão geral do sistema | 4 dias | 08/09/10 08:00 | 13/09/10 17:00 | |
| 2 | Eliciar necessidades dos interessados | 5 dias | 14/09/10 08:00 | 20/09/10 17:00 | 1 |
| 3 | Gerenciar dependências | 2 dias | 14/09/10 08:00 | 15/09/10 17:00 | 1 |
| 4 | Capturar vocabulário comum | 1 dia | 16/09/10 08:00 | 16/09/10 17:00 | 3 |
| 5 | Encontrar atores e casos de uso | 1 dia | 21/09/10 08:00 | 21/09/10 17:00 | 4;2 |
| 6 | Estruturar o modelo de casos de uso | 2 dias | 22/09/10 08:00 | 23/09/10 17:00 | 5 |
| 7 | Priorizar os casos de uso | 1 dia | 22/09/10 08:00 | 22/09/10 17:00 | 5 |
| 8 | Detalhar os casos de uso | 3 dias | 23/09/10 08:00 | 27/09/10 17:00 | 7 |
| 9 | Modelar interface com usuário | 3 dias | 28/09/10 08:00 | 30/09/10 17:00 | 8 |
| 10 | Prototipar interface com usuário | 6 dias | 01/10/10 08:00 | 08/10/10 17:00 | 9 |
| 11 | Revisar requisitos | 2 dias | 11/10/10 08:00 | 12/10/10 17:00 | 6;10 |

Figura 6.2 WBS em uma ferramenta de gerência de projetos, com duração prevista e dependências.

A partir da estruturação das atividades, o planejador do projeto deverá estimar os tempos necessários para a execução de cada atividade. Costuma ser difícil estimar tempo com grande precisão. Trabalha-se, então, com o *timeboxing* da iteração. O esforço total é o número de dias multiplicado pelo número de desenvolvedores. Devem-se determinar as sequências de atividades mais difíceis primeiro e alocar desenvolvedores a elas.

É necessário também verificar se as dependências entre as atividades criam um caminho crítico (Seção 6.5.5.2) cujo comprimento seja maior que a duração da iteração. Nesse caso, talvez seja necessário replanejar as atividades de forma que o caminho crítico e quaisquer outros caminhos caibam no *timeboxing* da iteração. Depois, distribui-se o tempo restante para as outras atividades. Eventuais erros para mais ou para menos nas estimativas podem compensar-se mutuamente.

6.5.5.1 Rede PERT

O grafo de dependências entre atividades com a duração prevista para cada atividade constitui-se na *rede PERT* do projeto ou iteração.

Há várias ferramentas que permitem a elaboração quase automática de um diagrama PERT. Nos exemplos a seguir, os gráficos foram gerados com a ferramenta gratuita OpenProj.²

Em geral, basta que se definam o conjunto das atividades, suas dependências e sua duração, e a ferramenta calcula as datas de início e de finalização prováveis de cada atividade, conforme mostrado na Figura 6.2.

Na Figura 6.2, o planejador preencheu as colunas “Nome”, “Duração” e “Predecessoras”, e as colunas “Início” e “Término” foram calculadas automaticamente pela ferramenta. Esse WBS foi gerado a partir do *workflow* da Figura 5.4.

Um exemplo de rede PERT gerado a partir da WBS da Figura 6.2 é apresentado na Figura 6.3.

6.5.5.2 Caminho Crítico

Um conceito importante no diagrama PERT é o caminho crítico, que consiste no mais longo caminho que leva do início ao fim do projeto ou iteração. Esse caminho crítico é importante porque, se qualquer atividade prevista nele atrasar por algum motivo, todo o projeto vai atrasar. Esse é um caminho sem folga.

Entretanto, as atividades que não pertencem ao caminho crítico podem ser adiadas até certo limite sem prejuízo ao projeto como um todo. Na Figura 6.3, as atividades do caminho crítico são mostradas em uma cor mais escura. Essa é uma característica automática da ferramenta. As atividades que não estão no caminho crítico são mostradas em uma cor mais clara.

O caminho crítico pode não ser simplesmente uma linha, mas um caminho composto, ou seja, atividades paralelas podem estar no caminho crítico.

²Disponível em <openproj.org/>.

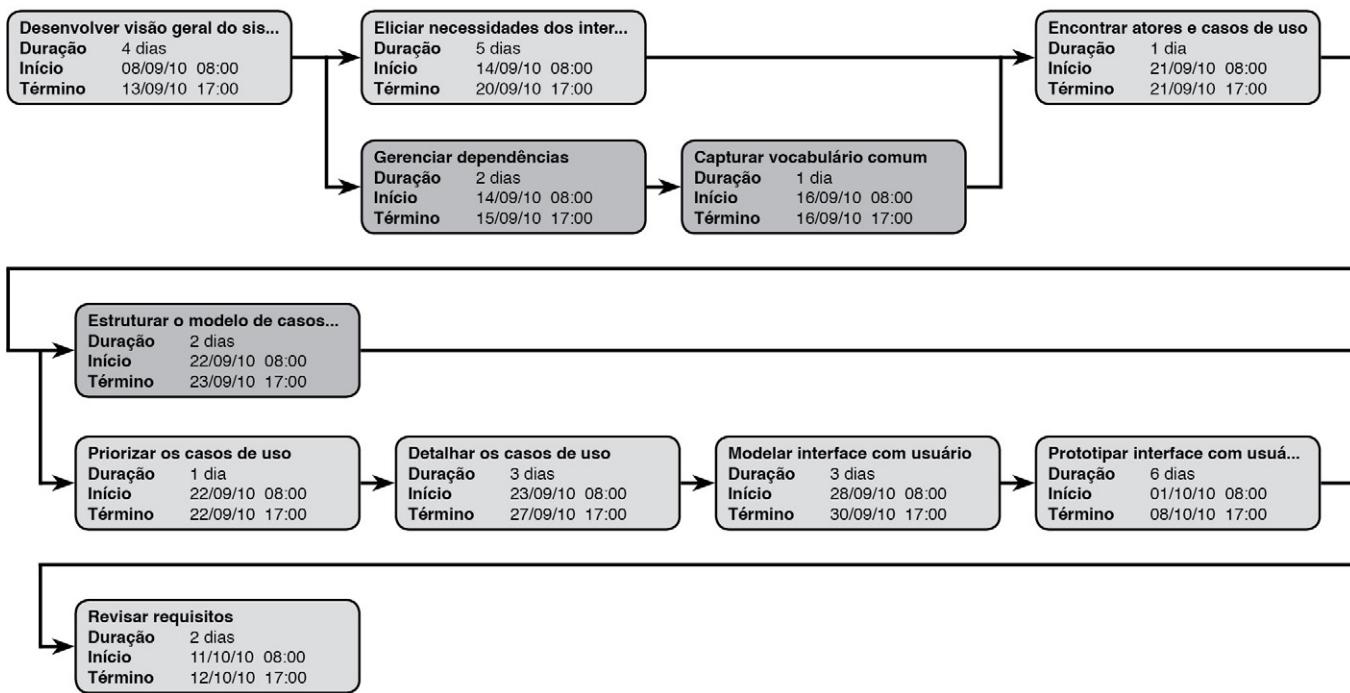


Figura 6.3 Uma rede PERT para as atividades da WBS.

Quando uma atividade do caminho crítico atrasa, é necessário acelerar alguma atividade posterior no caminho crítico para manter a iteração dentro do cronograma. A forma de obter essa aceleração será definida a critério do gerente de projeto. Existem três opções usuais:

- Aumentar a jornada da equipe (o que não pode se transformar em rotina).
- Aumentar o tamanho da equipe (o que pode causar transtornos de gerência em função da colocação de pessoas novas no projeto, possivelmente com menos experiência).
- Eliminar alguns objetivos (artefatos) ou características de artefatos da iteração. Por exemplo, em vez de implementar três casos de uso, caso haja atrasos, implementam-se apenas dois, deixando para outra iteração a implementação do terceiro.

O aumento da jornada ou intensificação do foco pode ajudar a recolocar nos trilhos um projeto ou iteração atrasados, mas, se isso ocorrer com muita frequência, o moral da equipe vai baixar e, possivelmente, atrasos serão cada vez mais frequentes.

Já o aumento do tamanho da equipe costuma produzir apenas resultados positivos em médio prazo, ou seja, duas ou três iterações depois daquela em que um ou mais novos membros foram adicionados. Por isso, inicialmente, essa solução deve atrasar ainda mais o projeto.

A eliminação de artefatos ou características de artefatos, que são remanejados para a lista de mudanças solicitadas a fim de serem resolvidos oportunamente em uma iteração futura, costuma ser a recomendação mais acertada nesses casos. Assim, a equipe se concentra em terminar algumas funcionalidades, obtém uma vitória relativa de curto prazo e consegue se reorganizar para retomar as funcionalidades faltantes em um momento de maior folga, de forma organizada.

6.5.6 CRONOGRAMA

Em geral, o cronograma do projeto é mostrado em um diagrama Gantt, que consiste em uma visualização do tempo linear transcorrido e da ocorrência das diferentes atividades ao longo desse tempo.

Em relação ao diagrama PERT, o diagrama Gantt apresenta o andamento das atividades ao longo de uma linha de tempo, permitindo visualizar claramente as atividades que devem ser executadas a cada dia.

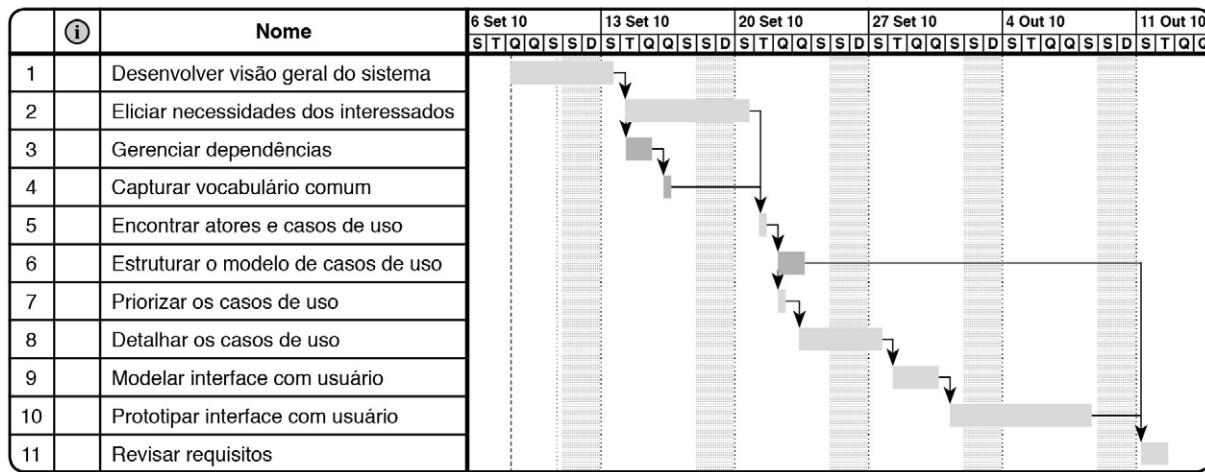


Figura 6.4 Diagrama Gantt para as atividades da WBS.

A Figura 6.4 mostra um diagrama Gantt para o diagrama PERT da Figura 6.3. Para que as atividades em paralelo como 2 e 3-4 possam, de fato, ser executadas ao mesmo tempo, elas devem ser alocadas a responsáveis diferentes.

Essas atividades de planejamento, porém, de nada adiantarão se não forem levadas a sério pelos desenvolvedores e pelo próprio gerente. Os capítulos seguintes indicam como fazer para que as estimativas de tempo sejam efetivamente realistas e como se preparar para possíveis problemas ao longo do projeto. Além disso, é mostrado mais adiante como o gerente deve fazer para bem conduzir um projeto durante seu desenvolvimento.



Estimativas de Esforço

Este capítulo vai apresentar algumas técnicas importantes e largamente usadas para estimar o esforço de desenvolvimento de software, porque não se pode planejar um projeto sem saber quanto tempo ele vai durar e quanto vai custar. Inicialmente, é apresentada a técnica mais antiga, baseada em linhas de código ou SLOC (Seção 7.1). Em seguida, são apresentados os conceitos fundamentais de uma técnica muito simples, também baseada em linhas de código, o COCOMO (Seção 7.2). Depois, sua sucessora mais avançada, COCOMO II, ou CII, adequada ao uso com o Processo Unificado, é apresentada em maior detalhe (Seção 7.3). Posteriormente são definidas as técnicas que não se baseiam em linhas de código, mas em funcionalidade aparente: *Análise de Pontos de Função* (Seção 7.4), a qual é bastante usada na indústria, *Pontos de Caso de Uso* (Seção 7.5), que vem se firmando por sua compatibilidade com o Processo Unificado, e *Pontos de História* (Seção 7.6), a técnica preferida dos adeptos dos métodos ágeis.

Uma das questões fundamentais em um projeto de software é saber, antes de executá-lo, quanto esforço, em horas de trabalho, será necessário para levá-lo a termo. Essa área, chamada de *estimativa de esforço*, conta com algumas técnicas que têm apresentado resultados interessantes ao longo dos últimos anos.

A maioria das técnicas de estimação de esforço utiliza pelo menos um parâmetro como base, por isso elas são chamadas de *técnicas paramétricas*.

Um exemplo de técnica *não paramétrica* é estimar que qualquer projeto de desenvolvimento sobre o qual não se sabe quase nada vai levar *seis meses* para ser executado. À medida que mais informações sobre o projeto vão sendo disponibilizadas, esse tempo é ajustado para cima ou para baixo, baseando-se na opinião de especialistas. Apesar de essa técnica ser bastante usada, poucos engenheiros de software conseguem obter boas previsões com ela.

Existem técnicas *paramétricas* baseadas em uma predição do número de linhas que o programa deverá ter e técnicas baseadas em requisitos, descritos como funções, casos de uso ou histórias de usuário.

As técnicas de pontos de função, caso de uso e histórias baseiam-se em um conjunto de requisitos, aos quais é atribuído um peso que determinará, a partir de certas transformações matemáticas, o esforço necessário para seu desenvolvimento.

Já as técnicas baseadas em linhas de código necessitam de uma estimativa de quantas linhas de código deverão ser produzidas.

Existem também mecanismos que permitem converter parâmetros como pontos de função em estimativas de linhas de código, fazendo que as técnicas acabem sendo compatíveis entre si.

Neste capítulo, inicialmente, serão vistas as técnicas baseadas em linhas de código. Depois, pontos de função, pontos de caso de uso e pontos de histórias.

7.1 SLOC e KSLOC

A técnica conhecida como LOC (*Lines of Code*) ou SLOC (*Source Lines of Code*) foi possivelmente a primeira a surgir e consiste em estimar o número de linhas que um programa deverá ter, normalmente com base na opinião de especialistas e no histórico de projetos passados.

Esse tipo de técnica surgiu numa época em que as linguagens de programação, como FORTRAN, eram fortemente orientadas a linhas de código. Naquele tempo, programas eram registrados em cartões perfurados, uma linha de código por cartão, de forma que a altura da pilha de cartões era uma medida bastante natural para a complexidade de um programa.

Rapidamente, a técnica evoluiu para a forma conhecida como KSLOC (*Kilo Source Lines of Code*), tendo em vista que o tamanho da maioria dos programas passou a ser medido em milhares de linhas. Uma unidade KSLOC, portanto, vale mil unidades SLOC. Além disso, também são usados os termos MSLOC para milhões de linhas e GSLOC para bilhões de linhas de código.

As linguagens atuais, porém, não são mais tão restritivas em termos de linhas de código, de forma que talvez fizesse mais sentido falar em comandos em vez de linhas. Usuários de linguagens como C, Java ou Pascal, por exemplo, poderão contar o número de comandos terminados por “;”, obtendo assim uma boa medida de complexidade de um programa. Mesmo assim, a noção de comando só funciona bem para linguagens imperativas. No caso de linguagens declarativas ou funcionais, outras formas de medir complexidade precisam ser usadas.

Mesmo assim, a noção de linha de código continua sendo uma medida popular para complexidade de programas, que podem variar de 10 a 100.000.000 de linhas, com complexidade inerente diretamente proporcional na maioria das vezes.

De fato, a medida faz sentido quando se comparam ordens de magnitude. Um programa com 100.000 linhas possivelmente será mais complexo do que um programa com 10.000 linhas. Mas pouco se pode concluir ao se comparar um programa com 10.000 linhas e um programa com 11.000 linhas.

7.1.1 ESTIMAÇÃO DE KSLOC

Uma técnica para estimação de KSLOC é reunir a equipe para discutir o sistema a ser desenvolvido. Cada participante dará a sua opinião sobre a quantidade de KSLOC que será necessária para desenvolver o sistema. Em geral, a equipe não chega a um valor único, por isso devem ser considerados pelo menos três valores:

- O KSLOC *otimista*, ou seja, o número mínimo de linhas que se espera desenvolver se todas as condições forem favoráveis.
- O KSLOC *pessimista*, ou seja, o número máximo de linhas que se espera desenvolver em condições desfavoráveis.
- O KSLOC *esperado*, ou seja, o número de linhas que efetivamente se espera desenvolver em uma situação de normalidade.

A partir desses valores, o KSLOC pode ser calculado assim:

$$\text{KSLOC} = (4 * \text{KSLOC}_{\text{esperado}} + \text{KSLOC}_{\text{otimista}} + \text{KSLOC}_{\text{pessimista}}) / 6$$

Essas estimativas devem ser comparadas com a informação real, ao final do projeto. Por isso, a equipe deve ter um *feedback* para ajustar futuras previsões. Por exemplo, um membro da equipe que costuma prever valores muito abaixo do real deverá perceber que está sendo muito otimista e tentar ajustar suas previsões para valores mais altos no futuro.

Uma técnica que pode ser empregada para ajustar a capacidade de previsão da equipe é tomar projetos já desenvolvidos, para os quais se saiba de antemão a quantidade de linhas, e exercitar a estimativa com a equipe, comparando mais tarde os valores reais com os valores estimados por seus membros.

Embora o objetivo seja acertar a previsão, isso na prática é quase impossível. Erros de previsão de 20% ou 30% não chegam a ser muito significativos, ou seja, é perfeitamente aceitável fazer uma previsão de

10.000 linhas para um projeto de 12.000 ou 13.000 linhas. Mas seria um problema caso a equipe fizesse uma previsão de 10.000 linhas para um projeto que, ao final, tivesse 30.000 ou 40.000 linhas (erro de 200 e 300%, respectivamente).

A medida de número de linhas, porém, ainda pode ser traíçoeira. Possivelmente, programadores experientes produzirão software com menos linhas do que programadores menos experientes.

Alguns cuidados devem ser tomados. Não se deve considerar SLOC uma boa medida de produtividade individual, pois, muitas vezes, a complexidade inerente de um programa vai muito além da quantidade de linhas: poucas linhas de código poderão realizar funcionalidades difíceis e complexas, enquanto muitas linhas poderão efetuar apenas funcionalidades triviais. É o caso de quem desenvolve software científico ou jogos em contraponto com a produção de meros cadastros e relatórios.

Outra situação a considerar é o fato de que refatorações do software poderão remover muitas linhas de código inútil ou redundante, e isso não deve ser considerado uma produtividade negativa.

Assim, a medida do número de linhas de código pode ser útil no longo prazo ou para um projeto como um todo, mas seu uso para avaliar atividades diárias dos desenvolvedores ou em pequenas partes de projeto é bastante arriscado.

7.1.2 TRANSFORMANDO PONTOS DE FUNÇÃO EM KSLOC

Existem estudos (Jones C., 1996) que apresentam uma relação entre o número de pontos de função não ajustados (UFP – Seção 7.3.3) e o número de linhas de código que se pode esperar em média, conforme a linguagem de programação (*backfire table*), como mostrado na Tabela 7.1.

Assim, um sistema que, pela técnica de Análise de Pontos de Função, tem seu tamanho estimado em 1.000 pontos de função terá um tamanho em SLOC estimado em 53.000 linhas Java, ou 91.000 linhas Pascal, ou ainda 55.000 linhas C++.

Esses valores são médios e foram obtidos a partir da análise estatística de vários projetos feita por Jones.

TABELA 7.1 Backfire table para conversão de UFP em SLOC¹

| Linguagem | Default SLOC/UFP | Linguagem | Default SLOC/UFP |
|-------------------|------------------|--------------------------|------------------|
| Access | 38 | Jovial | 107 |
| Ada 83 | 71 | Lisp | 64 |
| Ada 95 | 49 | Código de máquina | 640 |
| AI Shell | 49 | Modula 2 | 80 |
| APL | 32 | Pascal | 91 |
| Assembly – básico | 320 | Perl | 27 |
| Assembly – macro | 213 | PowerBuilder | 16 |
| Basic – ANSI | 64 | Prolog | 64 |
| Basic compilado | 91 | Query | 13 |
| C | 128 | Gerador de relatórios | 80 |
| C++ | 55 | Linguagem de simulação | 46 |
| Cobol | 91 | Planilha | 6 |
| Forth | 64 | Scripts da Shell do Unix | 107 |
| HTML 3.0 | 15 | Visual Basic | 29 |
| Java | 53 | Visual C++ | 34 |

¹Jones (1996).

7.1.3 COMO CONTAR LINHAS DE CÓDIGO

Dependendo da linguagem de programação, pode-se perguntar o que efetivamente conta como linha de código. A ideia é que apenas linhas lógicas contem. Mas ainda assim ficariam algumas dúvidas: “else” conta? Declaração de variável conta? Park (1992) apresenta, para várias linguagens de programação, os tipos de comandos que devem ser efetivamente contados e quais não devem ser contados como SLOC. Uma versão atualizada e revisada dessas orientações para uso com COCOMO II (Seção 7.3) que podem ser encontradas na forma de tabela no trabalho de Boehm (2000)² é resumida a seguir.

Em relação ao tipo de comando, devem ser contados:

- Comandos executáveis.
- Declarações.
- Diretivas de compilação.

De outro lado, não devem ser contados:

- a) Comentários.
- b) Linhas em branco.

Em relação à forma como o código é produzido, devem ser contadas as linhas:

- a) Programadas.
- b) Copiadas ou reusadas.
- c) Modificadas.

Não devem ser contadas linhas:

- a) Geradas por geradores automáticos de código.
- b) Removidas.

Em relação aos comandos presentes na maioria das linguagens de programação, devem ser contados:

- a) Comandos *null*, *continue* e *no-op*.
- b) Comandos que instanciam elementos genéricos.
- c) Pares *begin-end* ou {...} usados em comandos estruturados.
- d) Comandos *elseif*.
- e) Palavras-chave como *division*, *interface* e *implementation*.

Não devem ser contados:

- a) Comandos vazios como “;”, quando colocados sozinhos em uma linha.
- b) pares *begin-end* ou {...} usados para delimitar o bloco principal ou procedimentos e funções.
- c) Expressões passadas como argumentos para chamadas de procedimentos ou função (conta-se apenas a chamada).
- d) Expressões lógicas em comandos IF, WHILE ou REPEAT (conta-se apenas o comando que contém as expressões).
- e) Símbolos que servem de finalizadores de comandos executáveis, declarações ou subprogramas.
- f) Símbolos THEN, ELSE e OTHERWISE, quando aparecem sozinhos em uma linha (mas conta-se o comando que os sucede).

São entendidos como *comandos executáveis*, e devem ser contados, todos os comandos que sejam atribuições, GOTO, chamada de procedimento, chamada de macro, retorno, *break*, *exit*, *stop*, *continue*, *null*, *noop* etc. Também devem ser contadas separadamente as estruturas de controle, como estruturas de repetição e seleção, e seus blocos *begin-end* ou {...}, se existirem, contam separadamente. Por exemplo, o fragmento de código abaixo deve contar como 4 linhas de código:

²Disponível em: <csse.usc.edu/csse/research/COCOMOII/cocomo2000.0_CII_modelman2000.0.pdf>. Acesso em: 21 jan. 2013.

```

BEGIN                               0
  IF (a <> b) OR (b < c) THEN      +1
    BEGIN                           +1
      X := b;                      +1
    END                             0
  ELSE                            0
    X := factorial(a)            +1
  ;
END;                                0
                                         0
                                         4

```

As declarações são comandos não executáveis que também devem ser contados. Por exemplo, declarações de nomes, números, constantes, objetos, tipos, subtipos, programas, subprogramas, tarefas, exceções, pacotes, genéricos e macros. Blocos *begin-end* ou {...}, quando são parte obrigatória da declaração de subprogramas, não devem ser contados separadamente (apenas a declaração do subprograma conta). Assim, um bloco como o seguinte deve contar como 15 linhas:

```

procedure fib(num:integer):integer;          +1
var ultimoFib, cont, penultimoFib : integer; +1
begin                                         0
  if num = 0 then                         +1
    fib := 0                                +1
  else                                         0
    if num = 1 then                         +1
      fib := 1                                +1
    else                                         0
      begin                                    +1
        ultimoFib := 1;                      +1
        cont := 1;                          +1
        repeat                                 +1
          penultimoFib := ultimoFib;        +1
          ultimoFib := fib;                +1
          fib := penultimoFib + ultimoFib; +1
          cont := cont + 1;                +1
        until cont = num;                  +1
      end                                     0
    ;
  end                                       0
end;                                         0
                                         15

```

Note que, apesar de três variáveis terem sido declaradas, elas o foram em uma única linha, contando, portanto, uma única vez. Se fossem dois tipos de variável (por exemplo, inteiros e reais), seriam necessárias duas declarações e se contaria uma linha para cada uma:

```
VAR x, y : integer;    +1
      z : real;      +1
                      2
```

7.2 COCOMO

COCOMO ou *Constructive Cost Model* (também conhecido como COCOMO 81) é um modelo de estimativa de esforço baseado em KSLOC. Esse modelo já é obsoleto e foi substituído por COCOMO II em aplicações reais, mas ainda assim é apresentado aqui porque sua simplicidade conceitual ajuda a esclarecer conceitos usados por outras técnicas. Além disso, pode ser usado como uma ferramenta de estimativa grosseira, caso pouquíssima informação sobre o sistema esteja disponível.

O modelo COCOMO foi criado por Boehm (1981) a partir de um estudo empírico sobre 63 projetos na empresa TRW Aerospace. Os programas examinados tinham de 2 a 100 KSLOC e eram escritos em linguagens diversas, como Assembly e PL/I.

O modelo COCOMO 81 apresenta-se em três implementações de complexidade crescente, de acordo com o grau de informações que se tenha a respeito do sistema a ser desenvolvido:

- a) *Implementação básica*: quando a única informação sobre o sistema efetivamente disponível é o número estimado de linhas de código.
- b) *Implementação intermediária*: quando certos fatores relativos ao produto, suporte computacional, pessoal e processo são conhecidos e podem ser avaliados para o sistema a ser produzido.
- c) *Implementação avançada*: quando for necessário subdividir o sistema em subsistemas e distribuir as estimativas de esforço por fase e atividade.

Para o cálculo do esforço, todas as implementações do COCOMO consideram também o tipo de projeto a ser desenvolvido:

- a) *Modo orgânico*: aplica-se quando o sistema a ser desenvolvido não envolve dispositivos de hardware e a equipe está acostumada a desenvolver esse tipo de aplicação, ou seja, sistemas de baixo risco tecnológico e baixo risco de pessoal.
- b) *Modo semidestacado*: aplica-se a sistemas com maior grau de novidade para a equipe e que envolvem interações significativas com hardware, mas sobre os quais a equipe ainda tem algum conhecimento, ou seja, sistemas nos quais a combinação do risco tecnológico e de pessoal seja média.
- c) *Modo embutido*: aplica-se a sistemas com alto grau de interação e diferentes dispositivos de hardware, ou que sejam embarcados, e para os quais a equipe tenha considerável dificuldade de abordagem. São os sistemas com alto risco tecnológico e/ou de pessoal.

A Tabela 7.2 apresenta uma sugestão para determinar o tipo de projeto a partir dos riscos gerais identificados. Nos riscos de pessoal incluem-se também aspectos referentes ao cliente e aos requisitos. Requisitos altamente voláteis indicam um projeto de alto risco.

As três implementações do modelo COCOMO permitem determinar três informações básicas:

- a) O esforço estimado em desenvolvedor-mês: E .
- b) O tempo linear de desenvolvimento sugerido em meses corridos: D .
- c) O número médio de pessoas recomendado para a equipe: P .

O número de pessoas para a equipe será sempre dado por $P = E/D$. A medida parece simplista, porque aparentemente considera que a relação entre o número de pessoas e o tempo de projeto é linear. Mas o cálculo do esforço estimado (E) já leva em conta a não linearidade dessa relação, pois é uma função exponencial, como será visto adiante.

TABELA 7.2 Combinação de risco de pessoal e risco tecnológico para a escolha do tipo de projeto no contexto de COCOMO

| Risco tecnológico/ risco de pessoal | Alto | Médio | Baixo |
|--|---------------|---------------|---------------|
| Alto | Embutido | Embutido | Semidestacado |
| Médio | Embutido | Semidestacado | Orgânico |
| Baixo | Semidestacado | Orgânico | Orgânico |

TABELA 7.3 Valores de *ab*, *bb*, *cd* e *db* em função do tipo de projeto

| Tipo de projeto | <i>ab</i> | <i>bb</i> | <i>cb</i> | <i>db</i> |
|-----------------|-----------|-----------|-----------|-----------|
| Orgânico | 2,4 | 1,05 | 2,5 | 0,38 |
| Semidestacado | 3,0 | 1,12 | 2,5 | 0,35 |
| Embutido | 3,6 | 1,2 | 2,5 | 0,32 |

Além disso, é importante que nas fórmulas de COCOMO e CII seja usada sempre a unidade *desenvolvedor-mês*. Variações como *desenvolvedor-semana*, *desenvolvedor-dia* ou *desenvolvedor-hora* poderão provocar distorções nos resultados, em razão do uso de exponenciais nas fórmulas.

7.2.1 MODELO BÁSICO

A implementação mais simples de COCOMO é capaz de calcular esforço, tempo e tamanho de equipe a partir de uma simples estimativa de KSLOC.

Inicialmente, calcula-se o esforço (*E*) a partir da seguinte fórmula:

$$E = ab * KSLOC^{bb}$$

onde *ab* e *bb* são obtidos a partir da Tabela 7.3.

Já o tempo linear ideal recomendado para o desenvolvimento é dado por:

$$T = cb * E^{db}$$

onde *cb* e *db* são constantes também dadas na Tabela 7.3.

Por exemplo, um projeto com KSLOC estimado em 20 (20.000 linhas de código) com baixo risco (modo orgânico) produzirá as seguintes estimativas:

$$E = 2,4 * 20^{1,05} = 56 \text{ desenvolvedores-mês}$$

$$T = 11,5 \text{ meses}$$

$$P = 5 \text{ pessoas}$$

Todos os valores são aproximados, em razão das casas decimais, mas a conclusão do modelo COCOMO básico é que um projeto orgânico com previsão de 20.000 linhas de código será desenvolvido em cerca de 1 ano por uma equipe de cerca de 5 pessoas.

O modelo básico é bom por ser simples e rápido, mas sua capacidade preditiva é limitada, em razão do fato de que o esforço de desenvolvimento não é função apenas do número de linhas de código, mas também de outros fatores, que são considerados a partir do modelo intermediário.

7.2.2 MODELO INTERMEDIÁRIO

O modelo intermediário de COCOMO vai considerar uma avaliação sobre vários aspectos do projeto, além das simples linhas de código. A Tabela 7.4 apresenta os 15 fatores do modelo original organizados em quatro grupos. A tabela exige que se indique, para cada fator, uma nota que varia de “muito baixa” a “extra alta”. A partir dessa nota, um valor numérico correspondente é obtido para cada fator. Por exemplo, se a dimensão da base de dados esperada é “muito alta”, então o fator DATA terá valor numérico 1,16.

Note que nem todas as notas são aplicáveis a todos os fatores. Por exemplo, o fator DATA não pode ter notas “muito baixa” nem “extra-alta”.

Os 15 valores numéricos obtidos para os fatores são combinados através de multiplicação, produzindo o valor *EAF*:

$$EAF = RELY * DATA * CPLX * TIME * \dots * SCED$$

O modelo intermediário combina essa estimativa com *KSLOC* da seguinte forma:

$$E = ai * KSLOC^{bi} * EAF$$

em que os índices *ai* e *bi* são dados de acordo com a Tabela 7.5.

TABELA 7.4 Fatores influenciadores de custo do modelo COCOMO intermediário

| Fatores influenciadores de custo | Acrônimo | Muito baixa | Baixa | Média | Alta | Muito alta | Extra-alta |
|--|----------|-------------|-------|-------|------|------------|------------|
| Relativos ao produto | | | | | | | |
| Nível de confiabilidade requerida | RELY | 0,75 | 0,88 | 1,00 | 1,15 | 1,40 | |
| Dimensão da base de dados | DATA | | 0,94 | 1,00 | 1,08 | 1,16 | |
| Complexidade do produto | CPLX | 0,70 | 0,85 | 1,00 | 1,15 | 1,30 | 1,65 |
| Suporte computacional | | | | | | | |
| Restrições ao tempo de execução | TIME | | | 1,00 | 1,11 | 1,30 | 1,66 |
| Restrições ao espaço de armazenamento | STOR | | | 1,00 | 1,06 | 1,21 | 1,56 |
| Volatilidade da máquina virtual | VIRT | | 0,87 | 1,00 | 1,15 | 1,30 | |
| Tempo de resposta do computador | TURN | | 0,87 | 1,00 | 1,07 | 1,15 | |
| Pessoal | | | | | | | |
| Capacidade dos analistas | ACAP | 1,46 | 1,19 | 1,00 | 0,86 | 0,71 | |
| Experiência no domínio da aplicação | AEXP | 1,29 | 1,13 | 1,00 | 0,91 | 0,82 | |
| Capacidade dos programadores | PCAP | 1,42 | 1,17 | 1,00 | 0,86 | 0,70 | |
| Experiência na utilização da máquina virtual | VEXP | 1,21 | 1,10 | 1,00 | 0,90 | | |
| Experiência na linguagem de programação | LEXP | 1,14 | 1,07 | 1,00 | 0,95 | | |
| Processo | | | | | | | |
| Adoção de boas práticas de programação | MODP | 1,24 | 1,10 | 1,00 | 0,91 | 0,82 | |
| Uso de ferramentas atualizadas | TOOL | 1,24 | 1,10 | 1,00 | 0,91 | 0,83 | |
| Histórico de projetos terminados no prazo | SCED | 1,23 | 1,08 | 1,00 | 1,04 | 1,10 | |

TABELA 7.5 Valores de ai e bi em função do tipo de projeto

| Tipo de projeto | ai | bi |
|-----------------|------|------|
| Orgânico | 2,8 | 1,05 |
| Semidestacado | 3,0 | 1,12 |
| Embutido | 3,2 | 1,2 |

TABELA 7.6 Aplicação de notas aos fatores influenciadores de custo em cenário otimista e pessimista

| Fator | Otimista | Nota | Pessimista | Nota |
|----------------|------------|------|-------------|------|
| RELY | Média | 1,00 | Alta | 1,15 |
| DATA | Muito alta | 1,16 | Muito alta | 1,16 |
| CPLX | Média | 1,00 | Média | 1,00 |
| TIME | Alta | 1,11 | Muito alta | 1,30 |
| STOR | Alta | 1,06 | Muito alta | 1,21 |
| VIRT | Baixa | 0,87 | Baixa | 0,87 |
| ACAP | Média | 1,00 | Média | 1,00 |
| AEXP | Média | 1,00 | Média | 1,00 |
| PCAP | Média | 1,00 | Média | 1,00 |
| VEXP | Alta | 0,90 | Média | 1,00 |
| LEXP | Alta | 0,95 | Média | 1,00 |
| MODP | Baixa | 1,10 | Baixa | 1,10 |
| TOOL | Baixa | 1,10 | Muito baixa | 1,24 |
| SCED | Baixa | 1,08 | Muito baixa | 1,23 |
| Produto | | 1,33 | | 3,06 |

No modelo COCOMO intermediário, a avaliação dos fatores influenciadores do custo é dada de forma intuitiva, o que pode levar a certa subjetividade da avaliação. O modelo COCOMO II é bem mais detalhado em relação à forma como essas notas são atribuídas, mas a sua aplicação também é bem mais trabalhosa. Uma sugestão a ser seguida é que somente seja atribuída uma nota extrema caso não se consiga imaginar uma situação ainda mais extrema. Por exemplo, qual o nível de confiabilidade requerido para um sistema de videolocadora? Muito alto? Nesse caso, que avaliação deveria ser dada ao sistema embarcado em uma aeronave ou a um sistema de telecirurgia? De outro lado, a atribuição do conceito muito baixo para RELY no caso da videolocadora não significa que será gerado um sistema não confiável, apenas que não serão tomadas atitudes especiais de garantia de confiabilidade que seriam tomadas no caso de sistemas que colocam vidas em risco.

Além disso, é sempre possível fazer duas avaliações para os fatores influenciadores de custo, quando não se tem certeza sobre a melhor nota a ser atribuída. Por exemplo, na versão otimista, a nota de RELY poderia ser “baixa” e na pessimista poderia ser “média”. Deve-se observar, porém, que a atribuição de notas diferentes para muitos fatores poderá levar a resultados bem diferentes entre si, o que aumentaria bastante o fator de incerteza quanto à duração do projeto.

Por exemplo, no cenário otimista da Tabela 7.6, o valor de EAF será 1,3267. Já no cenário pessimista o EAF será 2,8497.

Note que a interpretação de otimismo ou pessimismo depende do fator: nos primeiros sete fatores, quanto mais alta a nota, mais pessimismo; nos oito fatores seguintes, quanto mais alta a nota, mais otimismo. Em caso de dúvida, deve-se ter em mente que o valor numérico da opção pessimista sempre deve ser igual ao valor numérico da opção otimista ou maior do que ele.

Assim, no cenário otimista, o esforço previsto para um projeto orgânico com 20 KSLOC seria:

$$E = 2,8 * 20^{1,05} * 1,33 = 86 \text{ desenvolvedores-mês}$$

Já no cenário pessimista o esforço seria:

$$E = 2,8 * 20^{1,05} * 3,06 = 199 \text{ desenvolvedores-mês}$$

Essa discrepância mostra como esses fatores podem afetar a produtividade da equipe em função do projeto.

7.2.3 MODELO AVANÇADO

A implementação avançada ou completa do modelo COCOMO introduz facetas como a decomposição do projeto em subprojetos, bem como estimativas individualizadas para as fases do projeto.

Como o modelo é, porém, complexo e desatualizado, sua apresentação será omitida em função da apresentação de seu sucessor COCOMO II na seção seguinte.

7.3 COCOMO II

COCOMO II³ ou CII é uma evolução do antigo modelo COCOMO 81 e, ao contrário de seu antecessor, funciona bem com ciclos de vida iterativos, sendo fortemente adaptado para uso com o Processo Unificado (Boehm B., 2000), embora também seja definido para os modelos Cascata e Espiral.

O CII foi projetado para mensurar o esforço e o tamanho médio de equipe para as fases de elaboração e construção do Processo Unificado. Então, o esforço e a equipe para as fases de concepção e transição podem ser calculados como uma fração dos valores obtidos para as duas outras fases. A Figura 7.1 apresenta esquematicamente a região de estimativa de esforço abrangida pelo método CII.

Como se pode ver na figura, CII é aplicado para determinar o esforço necessário para desenvolver as fases de elaboração e construção de um projeto. A duração das fases de concepção (caso ainda não tenha terminado) e transição deve ser calculada pela aplicação de um percentual sobre o esforço obtido para as fases de elaboração e construção (Seção 7.3.3).

O método CII define o esforço total, em desenvolvedor-mês, a partir do número de KSLOC, de uma constante ajustável por dados históricos A , de um valor que pode ser calculado para cada projeto, chamado de *expoente de esforço S*, e por um conjunto de fatores multiplicadores de esforço M_i , os quais também são individualmente calculados para cada projeto a partir de notas dadas. A equação geral que define o esforço total nominal de um projeto é a seguinte:

$$E = A * KSLOC^S * \prod_{i=1}^n M_i$$

Em que:

- a) E é o esforço total nominal que se deseja calcular para o projeto (fases de elaboração e construção).
- b) A é uma constante que deve ser calibrada a partir de dados históricos. CII sugere um valor inicial de 2,94.
- c) $KSLOC$ é o número estimado de milhares de linhas de código que deverão ser desenvolvidas.

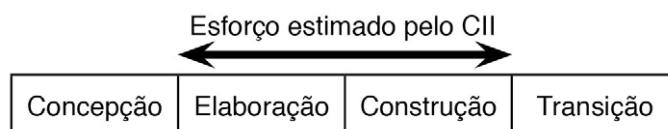


Figura 7.1 Região de estimação de esforço de COCOMO II.

³Disponível em: <sunset.usc.edu/csse/research/COCOMOII/cocomo_main.html>. Acesso em: 21 jan. 2013.

- d) S é o expoente de esforço, cujo cálculo é mostrado a seguir.
- e) M_i são os multiplicadores de esforço, que são explicados na Seção 7.3.2.

O expoente de esforço S é calculado a partir de uma constante B , que deve ser ajustada a partir de dados históricos e de um conjunto de cinco fatores de escala F_i . O cálculo é feito da seguinte forma:

$$S = B + 0,001 * \sum_{j=1}^5 F_j$$

Em que:

- a) S é o expoente de esforço que se deseja calcular.
- b) B é uma constante que deve ser calibrada de acordo com valores históricos. CII sugere um valor inicial de 0,91.
- c) F_j são cinco fatores de escala (*scale factors*) que devem ser atribuídos para cada projeto específico, tomando-se sempre muito cuidado, pois sua influência no cálculo do esforço total do projeto é exponencial.

Ainda em relação aos fatores multiplicadores de esforço M_i , sua quantidade varia em função de se estar calculando o esforço nas fases iniciais ou intermediárias do projeto.

Durante as fases de concepção e início da elaboração usa-se o *Early Design Model* (Seção 7.3.2.2), com 6 fatores ($n = 6$). Mais tarde, pode-se usar o *Post-Architecture Model* (Seção 7.3.2.1), com 16 fatores ($n=16$). A Figura 7.2 resume o momento em que cada um dos modelos deve ser usado, sendo que na fase de elaboração a decisão por um modelo ou outro vai depender de quanto já se tenha avançado.

Nota-se que, se qualquer um dos modelos for aplicado durante a fase de elaboração ou construção, o modelo vai prever o esforço total. Então, o esforço já despendido deverá ser descontado para que a previsão diga respeito ao tempo ainda restante do projeto.

O valor E corresponde, então, ao esforço total em desenvolvedor-mês para estas fases do UP. Mas, normalmente, projetos são desenvolvidos por equipes e não por uma única pessoa. Assim, uma equipe maior pode conseguir desenvolver um projeto mais rápido do que uma equipe menor. Mas existe um limite a partir do qual a equipe será grande demais para o tamanho do projeto e o esforço de gerenciar a equipe não compensará mais o ganho com tempo, havendo inclusive, uma inversão da curva no sentido de que quanto maior a equipe, maior será o tempo do projeto (Figura 7.3).

Pode-se falar, assim, em um tempo ideal e um tamanho de equipe ideal para desenvolver um projeto. O CII sugere que o tempo linear ideal para desenvolver um projeto cujo esforço total E já é conhecido seja calculado a partir da seguinte fórmula:

$$T = C * (E)^{D+0,2*(S-B)}$$

Em que:

- a) T é o tempo linear ideal de desenvolvimento.
- b) B, C e D são constantes que devem ser calibradas a partir de dados históricos (ver abaixo).
- c) E é o esforço total para o projeto, conforme calculado anteriormente.
- d) S é o expoente de esforço que já foi mencionado.

Um desenvolvedor-mês equivale a aproximadamente 152 horas de trabalho, uma conta que já exclui valores médios de fins de semana, feriados e férias.

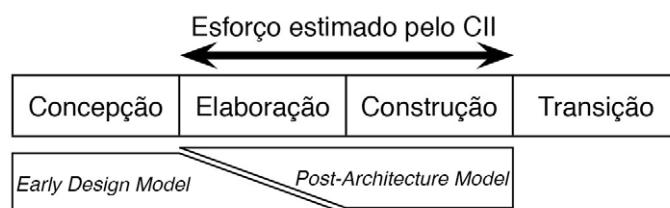


Figura 7.2 Momento de aplicação dos modelos *early design* e *post-architecture*.

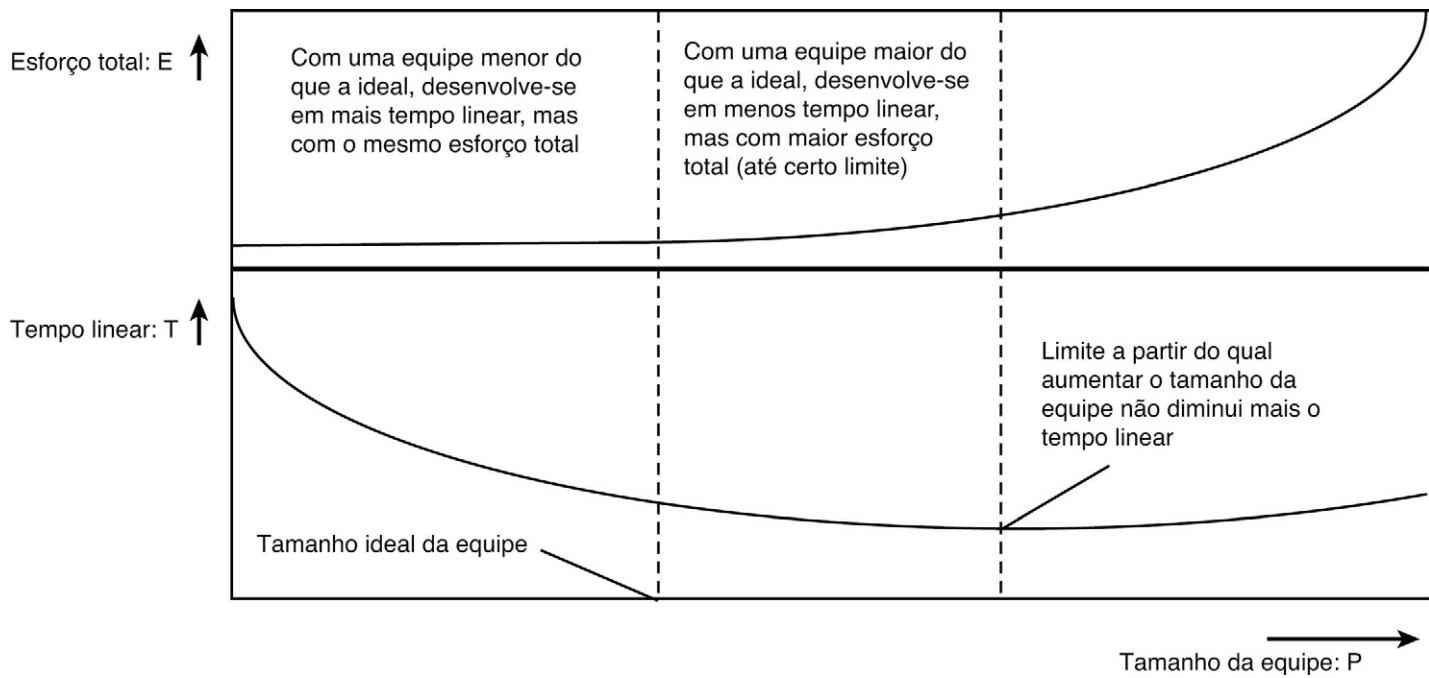


Figura 7.3 Relação entre o tamanho da equipe e o tempo linear de desenvolvimento de um projeto.

Finalmente, o *tamanho médio da equipe* P é obtido pela simples divisão do esforço total pelo tempo linear:

$$P = E / T$$

Essa fórmula, porém, só se aplica quando o tempo T for efetivamente o tempo linear ideal. De acordo com a Figura 7.3, pode-se perceber que uma redução no tempo linear não implica em um aumento de equipe proporcional, ou seja, para fazer o projeto na metade do tempo, não adianta dobrar o tamanho da equipe.

O método COCOMO II possui um multiplicador de esforço especificamente para indicar essa relação. O multiplicador SCED (cronograma de desenvolvimento requerido), apresentado na Seção 7.3.2.1, tem valor nominal igual a 1,0 se o projeto deve ser desenvolvido no seu tempo ideal. Caso se queira forçar um desenvolvimento mais rápido, em 85% do tempo ideal, SCED vai valer 1,14. Caso se queira forçar um desenvolvimento ainda mais rápido, em 75% do tempo ideal, SCED passa a valer 1,43, ou seja, indicando um esforço 43% maior para obter uma redução menor do que 25% no tempo linear. Isso porque o novo tempo linear será maior do que 25% do tempo originalmente calculado, já que o esforço total também será maior.

Os valores das constantes A , B , C e D foram obtidos pela equipe da USC (University of Southern California) a partir da análise de 161 projetos. Tais valores são os seguintes:

- a) $A = 2,94$
- b) $B = 0,91$
- c) $C = 3,67$
- d) $D = 0,28$

Porém, recomenda-se que pelo menos o valor de A seja calibrado a partir de dados obtidos na organização local que estiver usando a técnica (Seção 7.3.4).

7.3.1 FATORES DE ESCALA

Os cinco fatores de escala mencionados anteriormente receberão cada um uma nota que varia de “muito baixo” até “extremamente alto”. Os fatores de escala terão impacto exponencial no tempo de desenvolvimento. Se os fatores de escala forem nominais ($= 1,0$), então estima-se que um projeto com 200 KSLOC terá um esforço duas vezes maior do que um projeto com 100 KSLOC, ou seja, o crescimento do esforço é proporcional ao tamanho do projeto. Em outras palavras, mesmo que o projeto aumente de tamanho, o custo de cada linha de código permanece inalterado.

Se os fatores de escala ficarem acima do nominal ($> 1,0$), então um projeto com 200 KSLOC vai usar *mais do que o dobro* do esforço do que um projeto com 100 KSLOC, ou seja, quanto maior o projeto, mais cara se torna cada linha de código.

Por outro lado, se os fatores de escala ficarem abaixo do nominal ($< 1,0$), então um aumento do tamanho do projeto vai proporcionar um ganho em escala, ou seja, quanto maior o projeto, mais barata se torna cada linha de código.

Os fatores de escala são os seguintes:

- a) *Precedentes (PREC)*: Se o produto é similar a vários projetos desenvolvidos anteriormente, então PREC é alto.
- b) *Flexibilidade no Desenvolvimento (FLEX)*: Se o produto deve ser desenvolvido estritamente dentro dos requisitos, é preso a definições de interfaces externas, então FLEX é baixo.
- c) *Arquitetura/Resolução de Riscos (RESL)*: Se existe bom suporte para resolver riscos e para definir a arquitetura, então RESL é alto.
- d) *Coesão da Equipe (TEAM)*: Se a equipe é bem formada e coesa, então TEAM é alto.
- e) *Maturidade de Processo (PMAT)*: Esse fator pode estar diretamente associado com o nível de maturidade CMMI (Seção 12.3.3). Quanto mais alto o nível de maturidade, maior será o PMAT.

As tabelas abaixo são sugestões de Boehm sobre como definir a nota de cada um dos fatores de escala considerados. Para cada fator, toma-se a primeira tabela e decide-se qual a avaliação para cada uma das características na primeira coluna. Após obter todas as avaliações (uma para cada linha), determina-se uma nota para o fator como um todo, escolhendo a coluna mais representativa (muito baixo, baixo, nominal, alto, muito alto ou extremamente alto). A escolha da coluna mais representativa não é feita de maneira formal, ou seja, a ponderação dos requisitos será feita pelo planejador de projeto de acordo com sua percepção e experiência. Mas a princípio, deve-se considerar a coluna que represente a média das notas dadas.

Uma vez selecionada a avaliação (coluna) para o fator de escala, usa-se a segunda tabela para obter sua equivalente numérica, a qual será usada na fórmula do cálculo do esforço E , conforme mostrado no início deste capítulo.

Para PREC (o produto é similar a produtos anteriormente desenvolvidos pela mesma equipe), a Tabela 7.7, em duas partes, apresenta os padrões para atribuição do equivalente numérico.

Assim, por exemplo, imaginando que um projeto tenha a seguinte avaliação:

- a) Compreensão organizacional dos objetivos do produto: *considerável*.
- b) Experiência no trabalho com sistemas de software relacionados: *moderada*.

TABELA 7.7 Forma de obtenção do equivalente numérico para PREC

| Característica | Muito baixo Baixo | Nominal Alto | Muito alto Extra-alto |
|--|----------------------------|----------------------------|--------------------------|
| Compreensão organizacional dos objetivos do produto | Geral | Considerável | Total |
| Experiência no trabalho com sistemas de software relacionados | Moderada | Considerável | Extensiva |
| Desenvolvimento concorrente de novo hardware e procedimentos operacionais associados | Extensivo | Moderado | Algum |
| Necessidade de arquiteturas e algoritmos de processamento de dados inovadores | Considerável | Algum | Mínimo |
| Nota média | Muito baixo | Baixo | Nominal |
| Interpretação | Totalmente sem precedentes | Largamente sem precedentes | Um tanto sem precedentes |
| Fator numérico | 6,20 | 4,96 | 3,72 |
| | | | Alto |
| | | | Muito alto |
| | | | Extra-alto |
| | | | Totalmente familiar |
| | | | Altamente familiar |
| | | | Genericamente familiar |

TABELA 7.8 Forma de obtenção do equivalente numérico para FLEX

| Característica | | | Muito baixo Baixo | Nominal Alto | Muito alto Extra-alto | |
|---|-------------|-----------------------|----------------------|--------------------|--------------------------|--------------|
| Necessidade de conformação do software a requisitos preestabelecidos | | | Total | Considerável | Básica | |
| Necessidade de conformação do software a especificações de interfaces com sistemas externos | | | Total | Considerável | Básica | |
| Combinação das inflexibilidades acima com prêmio por término antecipado do projeto | | | Alto | Médio | Baixo | |
| Nota média | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra-alto |
| Interpretação | Rigoroso | Relaxamento ocasional | Algum relaxamento | Conformidade geral | Alguma conformidade | Metas gerais |
| Fator numérico | 5,07 | 4,05 | 3,04 | 2,03 | 1,01 | 0,00 |

c) Desenvolvimento concorrente de novo hardware e procedimentos operacionais associados: *moderado*.

d) Necessidade de arquiteturas e algoritmos de processamento de dados inovadores: *algum*.

Neste caso, tem-se três notas na coluna Nominal/Alto e uma nota na coluna Muito baixo/Baixo. Não faria sentido atribuir notas Muito baixo (porque apenas uma nota contra três está nessa coluna), nem Muito alto ou Extra-alto (porque nenhuma nota está nessas colunas). Como são três notas na coluna Nominal/Alto e uma na coluna Muito baixo/Baixo, pode-se atribuir nota “Nominal” a PREC (o limite inferior da segunda coluna). Assim, a interpretação de PREC vai considerar que o projeto é “um tanto sem precedentes” e o fator de escala numérico será 3,72.

A Tabela 7.8 apresenta a forma de cálculo do fator de escala *FLEX*, ou seja, qual a *flexibilidade no desenvolvimento* em relação aos requisitos.

Interpretando as tabelas anteriores, depreende-se que quanto maior o rigor em relação à conformidade com os requisitos, ou seja, quanto menor a flexibilidade do projeto, mais tempo ele vai levar para ser desenvolvido. Isso é natural, uma vez que as atividades de projeto incluem não apenas a escrita de código, mas todas as atividades, inclusive as inspeções de conformidade e testes exaustivos.

O fator RESL, ou seja, a existência de arquitetura ou sistema de suporte para *resolução de riscos* é calculado de acordo com os dados da Tabela 7.9.

Um risco crítico, ou de alta importância, deve ser considerado um risco com uma combinação de impacto e probabilidade alta, conforme mostrado na Seção 8.4. Riscos não críticos podem ser os de importância média. Riscos de baixa importância não precisam ser contabilizados na última linha da tabela de RESL.

No Processo Unificado, o porcentual do cronograma dedicado a estabelecer a arquitetura pode ser compreendido como o porcentual de duração da fase de elaboração em relação ao projeto como um todo, já que essa fase específica tem como objetivo estabilizar a arquitetura.

O fator de escala TEAM, ou seja, a *coesão da equipe de desenvolvimento*, pode ser calculado como mostrado na Tabela 7.10.

É possível observar que os fatores de escala do CII não são apenas itens para estimativa de esforço, mas também recomendações de boas práticas, ou seja, objetivos a serem buscados. Se cada uma das características listadas obtiver notas positivas, o tempo de desenvolvimento tenderá a ser muito mais baixo do que com notas mais negativas.

Finalmente, o fator PMAT, ou *maturidade do processo*, pode ser calculado a partir do nível de maturidade obtido pela empresa, de acordo com o modelo CMM (a Seção 12.3 apresenta o modelo CMMI, sucessor de CMM) ou SPICE (Seção 12.2). O equivalente numérico pode ser obtido como mostrado na Tabela 7.11.

Na falta de uma avaliação por CMMI ou SPICE, pode-se aplicar um questionário de avaliação (Boehm B., 2000) para obter o nível correspondente. Esse valor corresponde à linha “Nível EPML”, na primeira tabela da figura, onde EPML significa *Estimated Process Maturity Level*. O questionário e as formas de cálculo são apresentados na Seção 7.3.5.

TABELA 7.9 Forma de obtenção do equivalente numérico para RESL

| Característica | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra-alto |
|--|---------------------|-----------------|-----------------|------------------|------------------------|-------------------------|
| O plano de gerenciamento de risco identifica todos os itens de risco críticos e estabelece marcos para resolvê-los | Nada | Um pouco | Alguma coisa | Geralmente | Largamente | Totalmente |
| Cronograma, orçamento e marcos internos são compatíveis com o plano de gerenciamento de risco | Nada | Um pouco | Alguma coisa | Geralmente | Largamente | Totalmente |
| Percentual do cronograma de desenvolvimento dedicado a estabelecer a arquitetura, uma vez definidos os objetivos gerais do produto | 5 | 10 | 17 | 25 | 33 | 40 |
| Percentual de arquitetos de software experientes (<i>top</i>) disponíveis para o projeto em relação ao considerado necessário | 20 | 40 | 60 | 80 | 100 | 120 |
| Supporte de ferramentas disponível para resolver itens de risco, desenvolver e verificar especificações arquiteturais | Nenhum | Pouco | Algum | Bom | Forte | Total |
| Nível de incerteza nos determinantes-chave da arquitetura: missão, interface com usuário, COTS, hardware, tecnologia, desempenho | Extremo | Significativo | Considerável | Algum | Pouco | Muito pouco |
| Número de itens de risco e sua importância | Mais de 10 críticos | 5 a 10 críticos | 2 a 4 críticos | 1 crítico | Mais de 5 não críticos | Menos de 5 não críticos |
| Nota média | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra-alto |
| Interpretação | Pouco (20%) | Algum (40%) | Frequente (60%) | Geralmente (75%) | Largamente (90%) | Totalmente (100%) |
| Fator numérico | 7,07 | 5,65 | 4,24 | 2,83 | 1,41 | 0,00 |

Ao contrário das tabelas anteriores, nesta, apenas uma opção de avaliação entre CMM/CMMI e EPML/SPICE é necessária, e não uma combinação das duas.

7.3.2 MULTIPLICADORES DE ESFORÇO

Os multiplicadores de esforço M_i são usados para ajustar a estimativa de esforço para o desenvolvimento de um sistema baseando-se em características próprias do projeto e da equipe que podem onerar esse tempo.

CII apresenta dois grupos de multiplicadores de esforço: um para ser aplicado até a fase de elaboração (*Early Design Model*) e outro para ser aplicado depois (*Post-Architecture*), conforme já mostrado na Figura 7.2.

7.3.2.1 Multiplicadores de Esforço do Post-Architecture Model

Os multiplicadores de esforço do *Post-Architecture Model* estão divididos nos seguintes grupos:

- a) Fatores do produto.
- b) Fatores da plataforma.
- c) Fatores humanos.
- d) Fatores de projeto.

TABELA 7.10 Forma de obtenção do equivalente numérico para TEAM

| Característica | | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra-alto |
|---|---------------------------|-----------------------------|-------------------------------------|---|-----------------------------------|----------------------|------------|
| Consistência dos objetivos e cultura dos interessados | Pouca | Alguma | Básica | Considerável | Forte | Total | |
| Habilidade e vontade dos interessados em acomodar os objetivos de outros interessados | Pouca | Alguma | Básica | Considerável | Forte | Total | |
| Experiência dos interessados em trabalhar como uma equipe | Nenhuma | Pouca | Pouca | Básica | Considerável | Extensiva | |
| Construção de equipes com os interessados para obter visão compartilhada e compromissos | Nenhuma | Pouca | Pouca | Básica | Considerável | Extensiva | |
| Nota média | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra-alto | |
| Interpretação | Interações muito difíceis | Algumas interações difíceis | Interações basicamente cooperativas | Interações predominantemente cooperativas | Interações altamente cooperativas | Interações perfeitas | |
| Fator numérico | 5,48 | 4,38 | 3,29 | 2,19 | 1,10 | 0,00 | |

TABELA 7.11 Forma de obtenção do equivalente numérico para PMAT

| Característica | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra-alto |
|-----------------------|-----------------------|---------------------|-------------------|---------------------|---|----------------------------------|
| Nível CMM (ou CMMI) | 1 – inferior | 1 – superior | 2 | 3 | 4 | 5 |
| Nível EPML (ou SPICE) | 0 | 1 | 2 | 3 | 4 | 5 |
| Nota média | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra-alto |
| Interpretação | Sem processo definido | Processo incipiente | Processo definido | Processo gerenciado | Processo padronizado gerenciado quantitativamente | Processo em otimização constante |
| Fator numérico | 7,80 | 6,24 | 4,68 | 3,12 | 1,56 | 0,00 |

Os fatores do produto avaliam características do produto que podem afetar o esforço de desenvolvimento. Esses fatores são os seguintes:

- a) Software com Confiabilidade Requerida (RELY).
- b) Tamanho da Base de Dados (DATA).
- c) Complexidade do Produto (CPLX).
- d) Desenvolvimento Visando Reusabilidade (RUSE).
- e) Documentação Necessária para o Ciclo de Desenvolvimento (DOCU).

O multiplicador de esforço RELY (*Software com Confiabilidade Requerida*) avalia o tipo de consequências caso o software tenha alguma falha. Para aplicar a tabela, deve-se encontrar o descritor (primeira linha) que melhor descreve o fator de confiabilidade requerida. A partir dele encontram-se a avaliação e seu equivalente numérico na coluna correspondente (Tabela 7.12). Por exemplo, quando o efeito de uma falha do software é apenas inconveniente, então RELY é muito baixo e o equivalente numérico é 0,82.

No caso de RELY, a avaliação “Extra-alto” é inexistente (n/a). Isso também acontecerá com outros multiplicadores, como será visto mais adiante.

O multiplicador de esforço DATA (*Tamanho da Base de Dados*) avalia o tamanho relativo da base de dados usada para testes do programa (não a base de dados final). A razão D/P é o número de Kbytes na base de dados de teste

TABELA 7.12 Forma de obtenção do equivalente numérico para RELY

| Descriptor | Pequena inconveniência | Perdas pequenas, facilmente recuperáveis | Perdas moderadas, facilmente recuperáveis | Alta perda financeira | Risco à vida humana | |
|----------------------|------------------------|--|---|-----------------------|---------------------|------------|
| Avaliação | Muito baixo | Baixo | Nominal | Alto | Muito Alto | Extra-alto |
| Equivalente numérico | 0,82 | 0,92 | 1,00 | 1,10 | 1,26 | n/a |

TABELA 7.13 Forma de obtenção do equivalente numérico para DATA

| Descriptor | | D/P < 10 | 10 ≤ D/P ≤ 100 | 100 ≤ D/P ≤ 1.000 | D/P ≥ 1.000 | |
|----------------------|-------------|----------|----------------|-------------------|-------------|------------|
| Avaliação | Muito baixo | Baixo | Nominal | Alto | Muito Alto | Extra-alto |
| Equivalente numérico | n/a | 0,90 | 1,00 | 1,14 | 1,28 | n/a |

(D) dividido pelo número de milhares de linhas (P) estimado do programa (em KSLOC). A Tabela 7.13 apresenta os parâmetros de cálculo para DATA.

Assim, por exemplo, se o projetista estima que a base de dados para testes do sistema deverá ter 500 registros cada um com 2 Kbytes em média, D será igual a 1.000. Se o número de milhares de linhas de código P, for, digamos, 12 KSLOC, então $D/P = 1.000/12 = 83,333\dots$, e assim, o valor de DATA para esse projeto será Nominal.

O multiplicador de esforço CPLX (*Complexidade do Produto*) avalia a complexidade em cinco áreas: operações de controle (estruturas de controle, recursão, concorrência e distribuição), operações computacionais (cálculos matemáticos), operações dependentes de dispositivos (entrada e saída de dados), operações de gerenciamento de dados (desde simples dados em memória até bancos de dados distribuídos) e operações de gerenciamento de interface (simples entrada de texto num extremo e realidade virtual no outro). Então, ao contrário dos multiplicadores anteriores, que são obtidos a partir de um único descriptor, CPLX terá cinco descritores (um para cada área). A complexidade do produto é dada pela média subjetivamente ponderada dessas cinco áreas, conforme mostrado na Tabela 7.14.

O multiplicador de esforço RUSE (*Desenvolvimento Visando Reusabilidade*) avalia o quanto o projeto é feito pensando em gerar componentes que depois possam ser reusados. O desenvolvimento baseado em SPL (Linhas de Produto de Software – Seção 3.14), por exemplo, leva a um valor alto de RUSE. A Tabela 7.15 apresenta o padrão de atribuição de notas a esse multiplicador.

O maior esforço de desenvolvimento quando se usa linhas de produto de software é justamente decorrente da necessidade de maior esforço para identificar e estruturar uma família de produtos, o que é mais difícil do que quando se desenvolve um produto isolado. A vantagem das linhas de produto acaba aparecendo justamente quando um número significativo de produtos é desenvolvido e a reusabilidade reduz o esforço necessário para produzir cada produto individual derivado da linha.

O multiplicador DOCU (*Documentação Necessária para o Ciclo de Desenvolvimento*) mede o quanto a documentação necessária para o desenvolvimento realmente é produzida. Se não há compromisso com a documentação, a DOCU é baixa; se há documentação em excesso, além das necessidades reais, então o índice é alto. A Tabela 7.16 mostra como chegar aos valores numéricos desse multiplicador.

O fato de o processo de desenvolvimento deixar muitas necessidades de documentação não cobertas faz com que o desenvolvimento seja mais rápido, pois documentar leva tempo. Porém, isso usualmente não é uma boa prática, pois poderá gerar problemas mais adiante, especialmente na fase de operação do sistema, quando ele precisar sofrer manutenção.

Entretanto, documentação excessivamente burocrática exigirá muito esforço sem necessariamente produzir algum ganho depois.

Os multiplicadores de esforço referentes à *plataforma* se referem à complexidade da plataforma-alvo de implementação (hardware e software básico). Esses fatores são os seguintes:

TABELA 7.14 Forma de obtenção do equivalente numérico para CPLX

| | | | | | | |
|--|---|--|--|---|---|---|
| Operações de controle | Código sequencial com poucas estruturas não aninhadas. Composição simples de módulos via chamada de procedimentos ou <i>scripts</i> . | Aninhamento simples de estruturas de controle. Basicamente predicados simples. | Basicamente aninhamento simples. Algum controle intermódulos. Tabelas de decisão. Chamadas ou passagem de mensagens, incluindo processamento distribuído suportado por <i>middleware</i> . | Estruturas altamente aninhadas com vários predicados compostos. Controle de fila e pilha. Processamento distribuído homogêneo. Controle de tempo real simples em processador único. | Código reentrante e recursivo. Gerenciamento de interrupção com prioridade fixa. Sincronização de tarefas. Chamadas complexas. Processamento distribuído heterogêneo. Controle de tempo real complexo em processador único. | Escalonamento de múltiplos recursos com mudança dinâmica de prioridades. Controle em nível de microcódigo. Controle complexo de tempo real distribuído. |
| Operações computacionais | Avaliação de expressões simples como $A := B + C * (D - E)$. | Avaliação de expressões de nível moderado como $D := \text{SQRT}(B^{**2} - 4 * A * C)$. | Uso de rotinas matemáticas e estatísticas-padrão. Operações básicas sobre matrizes e vetores. | ANALISE NUMÉRICA BÁSICA: interpolação multivariada e equações diferenciais ordinárias. Arredondamento e truncamento básicos. | ANALISE NUMÉRICA COMPLEXA, mas estruturada: equações de matrizes, equações diferenciais parciais. Paralelização simples. | ANALISE NUMÉRICA COMPLEXA E NÃO ESTRUTURADA: análise de ruído altamente precisa, dados estocásticos. Paralelização complexa. |
| Operações dependentes de dispositivo | Comandos simples de leitura e escrita com formatação simples. | Sem necessidade de conhecimento de características particulares de processador ou dispositivo de E/S. E/S feita por Get e Put. | Processamento de E/S inclui seleção de dispositivo, checagem de <i>status</i> e processamento de erros. | OPERAÇÕES DE E/S EM NÍVEL FÍSICO (TRADUÇÕES DE ENDEREÇOS DE ARMAZENAMENTO FÍSICOS; BUSCAS E LEITURAS ETC.). Overlap de E/S otimizado. | ROTINAS PARA DIAGNÓSTICO DE INTERRUPÇÃO. GERENCIAMENTO DE LINHA DE COMUNICAÇÃO. SISTEMAS EMBARCADOS COM CONSIDERAÇÃO INTENSIVA DE PERFORMANCE. | CODIFICAÇÃO DE DISPOSITIVOS DEPENDENTES DE TEMPO. OPERAÇÕES MICROPGRAMADAS. SISTEMAS EMBUTIDOS COM PERFORMANCE CRÍTICA. |
| Operações de gerenciamento de dados | Arrays simples em memória. Simples consultas e atualizações em COTS ou banco de dados. | Arquivos simples sem edição nem buffers. Consultas e atualizações em bancos de dados ou COTS moderadamente complexas. | Entrada de múltiplos arquivos e saída em arquivo único. Mudanças estruturais simples. Edição simples. Consultas e atualizações complexas em COTS ou banco de dados. | GATILHOS SIMPLES ATIVADOS PELO CONTEÚDO DE SEQUÊNCIAS DE DADOS. REESTRUTURAÇÃO DE DADOS COMPLEXA. | COORDENAÇÃO DE BANCOS DE DADOS DISTRIBUÍDOS. GATILHOS COMPLEXOS. OTIMIZAÇÃO. | ESTRUTURAS RELACIONAIS E DE OBJETOS DINÂMICAS E ALTAMENTE ACOPLADAS. GERENCIAMENTO DE DADOS EM LINGUAGEM NATURAL. |
| Operações de gerenciamento de interface com usuário | Formulários de entrada simples e geradores de relatórios. | Uso de construtores de interface com usuário (GUI) simples. | Simples uso de um conjunto de widgets. | DESENVOLVIMENTO E EXTENSÃO DE CONJUNTO DE Widgets. E/S POR VOZ. MULTIMÍDIA. | GRÁFICOS DINÂMICOS 2D E 3D MODERADAMENTE COMPLEXOS. MULTIMÍDIA. | MULTIMÍDIA COMPLEXA. REALIDADE VIRTUAL. INTERFACE EM LINGUAGEM NATURAL. |
| Avaliação | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra-alto |
| Equivalente numérico | 0,73 | 0,87 | 1,00 | 1,17 | 1,34 | 1,74 |

TABELA 7.15 Forma de obtenção do equivalente numérico para RUSE

| Descriptor | | Nenhum reuso | Dentro do projeto | Dentro de um programa | Dentro de uma SPL | Entre múltiplas SPLs |
|----------------------|-------------|--------------|-------------------|-----------------------|-------------------|----------------------|
| Avaliação | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra-alto |
| Equivalente numérico | n/a | 0,95 | 1,00 | 1,07 | 1,15 | 1,24 |

TABELA 7.16 Forma de obtenção do equivalente numérico para DOCU

| Muitas necessidades de ciclo de vida não cobertas | Algumas necessidades de ciclo de vida não cobertas | Exatamente dimensionada para as necessidades do ciclo de vida | Excessiva para as necessidades do ciclo de vida | Muito excessiva para as necessidades do ciclo de vida | | |
|---|--|---|---|---|------------|------------|
| Descriptor | | | | | | |
| Avaliação | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra-alto |
| Equivalente numérico | 0,81 | 0,91 | 1,00 | 1,11 | 1,23 | n/a |

TABELA 7.17 Forma de obtenção do equivalente numérico para TIME

| | | | Menos de 50% de uso do tempo de execução disponível | 70% de uso do tempo de execução disponível | 85% de uso do tempo de execução disponível | 95% de uso do tempo de execução disponível |
|----------------------|-------------|-------|---|--|--|--|
| Descriptor | | | | | | |
| Avaliação | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra-alto |
| Equivalente numérico | n/a | n/a | 1,00 | 1,11 | 1,29 | 1,63 |

TABELA 7.18 Forma de obtenção do equivalente numérico para STOR

| | | | Menos de 50% de uso da memória principal | 70% de uso da memória principal | 85% de uso da memória principal | 95% de uso da memória principal |
|----------------------|-------------|-------|--|---------------------------------|---------------------------------|---------------------------------|
| Descriptor | | | | | | |
| Avaliação | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra-alto |
| Equivalente numérico | n/a | n/a | 1,00 | 1,05 | 1,17 | 1,46 |

- a) Restrição de Tempo de Execução (TIME).
- b) Restrição de Memória Principal (STOR).
- c) Volatilidade da Plataforma (PVOL).

O multiplicador TIME (*Restrição de Tempo de Execução*) avalia a porcentagem esperada de uso dos processadores disponíveis pela aplicação (Tabela 7.17).

O multiplicador STOR (*Restrição de Memória Principal*) avalia a porcentagem esperada de uso da memória principal pela aplicação. A Tabela 7.18 apresenta os equivalentes numéricos para esse multiplicador.

O uso de técnicas como *swap* nos modernos sistemas operacionais pode fazer que a preocupação com o uso da memória principal seja sempre nominal no caso de sistemas de informação, porque a memória sempre pode ser aumentada virtualmente. Porém, em aplicações embarcadas, esse indicador pode ser avaliado de forma mais crítica, justamente pela falta desse tipo de mecanismo.

O multiplicador PVOL (*Volatilidade da Plataforma*) avalia a plataforma de desenvolvimento, a qual inclui hardware e software básicos, sobre os quais a aplicação é construída. O fator é avaliado como “Baixo” quando

ocorrem mudanças de plataforma em períodos superiores a um ano e como “Alto” quando as mudanças ocorrem em média a cada duas semanas. A Tabela 7.19 mostra como obter os valores numéricos.

Os fatores humanos considerados multiplicadores de esforço são os seguintes:

- a) Capacidade dos Analistas (ACAP).
- b) Capacidade dos Programadores (PCAP).
- c) Continuidade de Pessoal (PCON).
- d) Experiência em Aplicações Semelhantes (APEX).
- e) Experiência na Plataforma (PLEX).
- f) Experiência na Linguagem e Ferramentas (LTEX).

O multiplicador ACAP (*Capacidade dos Analistas*) avalia a capacidade dos analistas de analisar e modelar aplicações, eficiência e eficácia, e as habilidades de cooperar e comunicar. Quanto maior a capacidade, menor o valor de ACAP. A Tabela 7.20 mostra como obter o valor numérico para a capacidade dos analistas, avaliada em termos de percentis. Por exemplo, se os analistas estão no percentil 15% mais baixo, então o multiplicador é avaliado como “Muito baixo”.

O multiplicador PCAP (*Capacidade dos Programadores*) avalia os programadores de forma semelhante à ACAP (Tabela 7.21).

O multiplicador PCON (*Continuidade de Pessoal*) avalia a porcentagem de trocas de desenvolvedores no período de um ano. Quanto menos trocas, menor o valor de PCON. A Tabela 7.22 mostra como obter o valor numérico de PCON a partir da porcentagem de troca de desenvolvedores no período de um ano.

TABELA 7.19 Forma de obtenção do equivalente numérico para PVOL

| | | | | | | |
|-----------------------------|-------------|---|---------------------------------------|--------------------------------------|--------------------------------------|------------|
| Descriptor | | Mudanças grandes a cada 12 meses, pequenas a cada mês | Grandes: 6 meses; pequenas: 2 semanas | Grandes: 2 meses; pequenas: 1 semana | Grandes: 2 semanas; pequenas: 2 dias | |
| Avaliação | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra-alto |
| Equivalente numérico | n/a | 0,87 | 1,00 | 1,15 | 1,30 | n/a |

TABELA 7.20 Forma de obtenção do equivalente numérico para ACAP

| | | | | | | |
|-----------------------------|--------------|-------|---------|------|------------|------------|
| Descriptor | Percentil 15 | 35 | 55 | 75 | 90 | |
| Avaliação | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra-alto |
| Equivalente numérico | 1,42 | 1,19 | 1,00 | 0,85 | 0,71 | n/a |

TABELA 7.21 Forma de obtenção do equivalente numérico para PCAP

| | | | | | | |
|-----------------------------|--------------|-------|---------|------|------------|------------|
| Descriptor | Percentil 15 | 35 | 55 | 75 | 90 | |
| Avaliação | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra-alto |
| Equivalente numérico | 1,34 | 1,15 | 1,00 | 0,88 | 0,76 | n/a |

TABELA 7.22 Forma de obtenção do equivalente numérico para PCON

| | | | | | | |
|-----------------------------|-------------|---------|---------|--------|------------|------------|
| Descriptor | 48%/ano | 24%/ano | 12%/ano | 6%/ano | 3%/ano | |
| Avaliação | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra-alto |
| Equivalente numérico | 1,29 | 1,12 | 1,00 | 0,90 | 0,81 | n/a |

O multiplicador APEX (*Experiência em Aplicações Semelhantes*) avalia o tempo médio (em anos) de experiência da equipe em aplicações semelhantes à que vai ser desenvolvida. Quanto maior o tempo, menor o valor de APEX. A Tabela 7.23 mostra como se obter os valores para APEX.

O multiplicador PLEX (*Experiência na Plataforma*) avalia a experiência da equipe na plataforma de desenvolvimento, incluindo bibliotecas, hardware, sistema operacional, banco de dados, middleware e outros itens relacionados. A Tabela 7.24 apresenta a forma de se obter os equivalentes numéricos para PLEX.

O multiplicador LTEX (*Experiência na Linguagem e Ferramentas*) avalia o tempo médio de experiência da equipe nas ferramentas CASE e linguagens usadas para o desenvolvimento (Tabela 7.25).

Os fatores de projeto avaliam a influência do uso de ferramentas modernas de desenvolvimento, ambiente de trabalho e aperto do cronograma. Esses fatores são os seguintes:

- a) Uso de Ferramentas de Software (TOOL).
- b) Equipe de Desenvolvimento Distribuída (SITE).
- c) Cronograma de Desenvolvimento Requerido (SCED).

O multiplicador TOOL (*Uso de Ferramentas de Software*) avalia a qualidade do suporte computacional ao ambiente de desenvolvimento. O uso de simples compiladores implica um índice ruim (alto) para esse fator, enquanto o uso de ferramentas CASE e de gerenciamento de projeto que integram todas as atividades de desenvolvimento implicam uma boa avaliação (índice baixo). A Tabela 7.26 mostra como obter os valores numéricos para TOOL.

O multiplicador SITE (*Equipe de Desenvolvimento Distribuída*) avalia a influência da distribuição da equipe de desenvolvimento. Equipes distribuídas internacionalmente apontam para um aumento de carga em relação a esse fator, enquanto uma equipe que trabalha toda na mesma sala implica uma carga menor. Ao contrário da maioria dos

TABELA 7.23 Forma de obtenção do equivalente numérico para APEX

| | | | | | | |
|-----------------------------|------------------|---------|---------|--------|------------|------------|
| Descriptor | Menos de 2 meses | 6 meses | 1 ano | 3 anos | 6 anos | |
| Avaliação | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra-alto |
| Equivalente numérico | 1,22 | 1,10 | 1,00 | 0,88 | 0,81 | n/a |

TABELA 7.24 Forma de obtenção do equivalente numérico para PLEX

| | | | | | | |
|-----------------------------|------------------|---------|---------|--------|------------|------------|
| Descriptor | Menos de 2 meses | 6 meses | 1 ano | 3 anos | 6 anos | |
| Avaliação | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra-alto |
| Equivalente numérico | 1,19 | 1,09 | 1,00 | 0,91 | 0,85 | n/a |

TABELA 7.25 Forma de obtenção do equivalente numérico para LTEX

| | | | | | | |
|-----------------------------|------------------|---------|---------|--------|------------|------------|
| Descriptor | Menos de 2 meses | 6 meses | 1 ano | 3 anos | 6 anos | |
| Avaliação | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra-alto |
| Equivalente numérico | 1,20 | 1,09 | 1,00 | 0,91 | 0,84 | n/a |

TABELA 7.26 Forma de obtenção do equivalente numérico para TOOL

| | | | | | | |
|-----------------------------|-----------------------------|---------------------------------|--|--|---|------------|
| Descriptor | Editar, codificar, debugar. | CASE simples. Pouca integração. | Ferramentas básicas de ciclo de vida moderadamente integradas. | Ferramentas de ciclo de vida fortes e maduras, moderadamente integradas. | Ferramentas de ciclo de vida fortes, maduras e bem integradas com processos, métodos e reúso. | |
| Avaliação | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra-alto |
| Equivalente numérico | 1,17 | 1,09 | 1,00 | 0,90 | 0,78 | n/a |

TABELA 7.27 Forma de obtenção do equivalente numérico para SITE

| Descriptor de co-locação | Internacional | Multicidade e multiempresa | Multicidade ou multiempresa | Mesma cidade ou área metropolitana | Mesmo edifício ou complexo | Totalmente co-locada |
|---------------------------|---------------------------|----------------------------|-----------------------------|---------------------------------------|----------------------------|-----------------------|
| Descriptor de comunicação | Alguns telefones, correio | Telefones individuais, fax | E-mail | Comunicação eletrônica de banda larga | Videoconferência | Multimídia interativa |
| Avaliação | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra-alto |
| Equivalente numérico | 1,22 | 1,09 | 1,00 | 0,93 | 0,86 | 0,80 |

TABELA 7.28 Forma de obtenção do equivalente numérico para SCED

| Descriptor | 75% do tempo nominal | 85% | 100% | 130% | 160% | |
|----------------------|----------------------|-------|---------|------|------------|------------|
| Avaliação | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra-alto |
| Equivalente numérico | 1,43 | 1,14 | 1,00 | 1,00 | 1,00 | n/a |

multiplicadores, SITE tem dois descritores. Deve ser feita a média subjetiva entre os dois para determinar a nota do multiplicador. A Tabela 7.27 mostra como obter o valor numérico para SITE.

Assim, uma equipe internacional (muito baixo) que utilize videoconferência (muito alto), por exemplo, pode ser avaliada na média desses dois descritores: nominal.

O multiplicador SCED (*Cronograma de Desenvolvimento Requerido*) reflete o grau requerido de aceleração forçada a um cronograma nominal ideal ou seu relaxamento (Tabela 7.28). Um cronograma forçadamente mais rápido do que o usual implica um fator com valor mais alto, ou seja, maior esforço de desenvolvimento.

7.3.2.2 Multiplicadores de Esforço do Early Design Model

Os multiplicadores de esforço do *Early Design Model* são o resultado de combinações dos fatores do *Post-Architecture Model*, além de outras informações. Parte-se do princípio de que, quando esse modelo é aplicado, ainda não se tem muita informação sobre o real ambiente de desenvolvimento e as características do projeto. Dessa forma, as avaliações dos 17 multiplicadores de esforço do *Post-Architecture Model* são meras estimativas. Assim, o *Early Design Model* cria seus multiplicadores de esforço a partir de uma combinação dessas estimativas com outras informações mais passíveis de serem conhecidas nessa fase de um projeto.

Os fatores multiplicadores de esforço do *Early Design Model* são os seguintes:

- a) Capacidade de Pessoal (PERS).
- b) Confiabilidade e Complexidade do Produto (RCPX).
- c) Desenvolvimento para Reúso (RUSE).
- d) Dificuldade com a Plataforma (PDIF).
- e) Experiência do Pessoal (PREX).
- f) Instalações (FCIL).
- g) Cronograma de Desenvolvimento Requerido (SCED).

O multiplicador PERS (*Capacidade de Pessoal*) é obtido a partir da média subjetiva de três descritores. O primeiro deles é a soma dos multiplicadores ACAP, PCAP e PCON, definidos para a fase de construção, mas considerando a seguinte equivalência numérica:

- a) Muito baixo=1
- b) Baixo=2
- c) Nominal=3
- d) Alto=4
- e) Muito alto=5
- f) Extra-alto=6

TABELA 7.29 Forma de obtenção do equivalente numérico para PERS

| Soma de ACAP, PCAP e PCON | 3 a 4 | 5 a 6 | 7 a 8 | 9 | 10 a 11 | 12 a 13 | 14 a 15 |
|---------------------------------|-------------|-------------|-------|---------|---------|------------|------------|
| Média dos percentis ACAP e PCAP | 20% | 35% | 45% | 55% | 65% | 75% | 85% |
| Taxa de troca de pessoal anual | 45% | 30% | 20% | 12% | 9% | 6% | 4% |
| Avaliação | Extra baixo | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra-alto |
| Equivalente numérico | 2,12 | 1,62 | 1,26 | 1,00 | 0,83 | 0,63 | 0,50 |

TABELA 7.30 Forma de obtenção do equivalente numérico para RCPX

| Soma de RELY, DATA, CPLX e DOCU | 5 a 6 | 7 a 8 | 9 a 11 | 12 | 13 a 15 | 16 a 18 | 19 a 21 |
|---|---------------|-------------|---------|----------|----------|----------------|-----------------------|
| Ênfase em confiabilidade e documentação | Muito pouca | Pouca | Alguma | Básica | Forte | Muito forte | Extrema |
| Complexidade do produto | Muito simples | Simples | Alguma | Moderada | Complexa | Muito complexa | Extremamente complexa |
| Tamanho do banco de dados | Pequeno | Pequeno | Pequeno | Moderado | Grande | Muito grande | Muito grande |
| Avaliação | Extra baixo | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra alto |
| Equivalente numérico | 0,49 | 0,60 | 0,83 | 1,00 | 1,33 | 1,91 | 2,72 |

TABELA 7.31 Forma de obtenção do equivalente numérico para PDIF

| Soma de TIME, STOR e PVOL | | | 8 | 9 | 10 a 12 | 13 a 15 | 16 a 17 |
|--|-------------|-------------|---------------|--------------|------------------|------------|-------------------|
| Restrição de tempo e memória combinadas de TIME e STOR | | | Menos de 50% | Menos de 50% | 65% | 80% | 90% |
| Volatilidade da plataforma | | | Muito estável | Estável | Um tanto volátil | Volátil | Altamente volátil |
| Avaliação | Extra baixo | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra alto |
| Equivalente numérico | n/a | n/a | 0,87 | 1,00 | 1,29 | 1,81 | 2,61 |

Assim, o valor da soma dos três multiplicadores, que nesse caso variam de “Muito baixo” (1) a “Muito alto” (5) dará um resultado entre 3 e 15. O segundo descritor é o percentil combinado de ACAP e PCAP. O terceiro é o percentual anual de troca de pessoal. Observe que passa a existir a nota “Extra baixo”, que não aparecia nos multiplicadores *Post-Architecture*. A Tabela 7.29 mostra como obter os valores para PERS.

O multiplicador RCPX (*Confiabilidade e Complexidade do Produto*) é uma combinação dos fatores RELY, DATA, CPLX e DOCU. Ele é obtido a partir da média subjetiva de quatro descritores. A Tabela 7.30 mostra como obter os equivalentes numéricos para esse multiplicador.

O multiplicador RUSE (*Desenvolvimento para Reuso*) nessa fase é exatamente o mesmo que foi calculado para *Post-Architecture*.

O multiplicador PDIF (*Dificuldade com a Plataforma*) é uma combinação de TIME, STOR e PVOL e mais dois descritores. A Tabela 7.31 mostra como obter os valores para PDIF.

O multiplicador PREX (*Experiência do Pessoal*) é uma combinação de APEX, LTEX e PLEX com o descritor que se refere à experiência média da equipe. A Tabela 7.32 mostra como obter os valores para PREX.

O multiplicador FCIL (*Instalações*) é uma combinação de TOOL e SITE, além de mais dois descritores. A Tabela 7.33 mostra como obter os valores para FCIL.

O multiplicador SCED (*Cronograma de Desenvolvimento Requerido*) é o mesmo SCED calculado para *Post-Architecture*.

TABELA 7.32 Forma de obtenção do equivalente numérico para PREX

| | | | | | | | |
|---|------------------|-------------|---------|---------|---------|------------|------------|
| Soma de APEX, PLEX e LTEX | 3 a 4 | 5 a 6 | 7 a 8 | 9 | 10 a 11 | 12 a 13 | 14 a 15 |
| Experiência média nas aplicações, plataforma, linguagem e ferramentas | Menos de 3 meses | 5 meses | 9 meses | 1 ano | 2 anos | 4 anos | 6 anos |
| Avaliação | Extrabaixo | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra-alto |
| Equivalente numérico | 1,59 | 1,33 | 1,22 | 1,00 | 0,87 | 0,74 | 0,62 |

TABELA 7.33 Forma de obtenção do equivalente numérico para FCIL

| | | | | | | | |
|-------------------------|--|---|---|---|--|---|---|
| Soma de TOOL e SITE | 2 | 3 | 4 a 5 | 6 | 7 a 8 | 9 a 10 | 11 |
| Suporte por ferramentas | Mínimo | Algum | Coleção simples de ferramentas CASE | Ferramentas básicas de ciclo de vida | Bom, moderadamente integrado | Forte, moderadamente integrado | Forte, bem integrado |
| Condições multisite | Suporte fraco em ambiente multisite complexo | Algum suporte para desenvolvimento multisite complexo | Algum suporte para desenvolvimento multisite moderadamente complexo | Suporte básico para ambiente multisite moderadamente complexo | Suporte forte para ambiente multisite moderadamente complexo | Suporte forte para ambiente multisite simples | Suporte muito forte para equipe co-locada ou ambiente multisite simples |
| Avaliação | Extrabaixo | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra-alto |
| Equivalente numérico | 1,43 | 1,30 | 1,10 | 1,00 | 0,87 | 0,73 | 0,62 |

TABELA 7.34 Aplicação de esforço e tempo linear às fases do UP

| Fase | Esforço nominal | Intervalo | Tempo linear nominal | Intervalo |
|---------------|-----------------|-------------|----------------------|-------------|
| Concepção | 0,06E | 0,02 a 0,15 | 0,125T | 0,02 a 0,30 |
| Elaboração | 0,24E | 0,20 a 0,28 | 0,375T | 0,33 a 0,42 |
| Construção | 0,76E | 0,72 a 0,80 | 0,625T | 0,58 a 0,67 |
| Transição | 0,12E | 0,00 a 0,20 | 0,125T | 0,00 a 0,20 |
| Totais | 1,18E | | 1,25T | |

7.3.3 APPLICANDO COCOMO II PARA AS FASES DO UP

Como mostrado na Tabela 7.34, CII estima o esforço E a ser despendido nas fases de elaboração e construção. Considera-se, assim, que 100% do valor E estará contido nessas fases. O esforço despendido nas fases de concepção e transição será *somado* a esse valor, passando de 100%. O mesmo raciocínio aplica-se ao tempo linear T . As colunas “Intervalo”, na Tabela 7.34, indicam os limites dentro dos quais se espera que o esforço e o tempo linear possam variar.

Essa tabela é compatível com a Figura 6.1, em que os valores de esforço e tempo linear são convertidos em porcentagens do esforço total das quatro fases. Assim, por exemplo, um projeto com $E=56$ desenvolvedor-mês e $T=11,5$ meses terá, em média, os valores de esforço (em desenvolvedor-mês) e duração por fase definidos como na Tabela 7.35.

Entretanto, os tempos e esforços por fase variam dentro de faixas, conforme mostrado na Tabela 7.34. O esforço e o tempo das fases de elaboração e construção devem variar inversamente, porque ambos, somados, devem resultar sempre no valor E e T , respectivamente.

TABELA 7.35 Exemplo de cálculo de tempo e esforço para as fases do UP de um projeto com $E=56$ e $T=11,5$

| Fase | Esforço (desenvolvedor-mês) | Tempo (meses) |
|---------------|-----------------------------|---------------|
| Concepção | 3,4 | 1,4 |
| Elaboração | 13,4 | 4,3 |
| Construção | 42,6 | 7,2 |
| Transição | 6,7 | 1,4 |
| Totais | 66,1 | 14,3 |

TABELA 7.36 Resumo do esforço relativo às disciplinas UP nas diferentes fases

| Disciplina | Concepção (%) | Elaboração (%) | Construção (%) | Transição (%) |
|-----------------------|---------------|----------------|----------------|---------------|
| Gerenciamento | 14 | 12 | 10 | 14 |
| Ambiente/Configuração | 10 | 8 | 5 | 5 |
| Requisitos | 38 | 18 | 8 | 4 |
| <i>Design</i> | 19 | 36 | 16 | 4 |
| Implementação | 8 | 13 | 34 | 19 |
| Avaliação/Teste | 8 | 10 | 24 | 24 |
| Implantação | 3 | 3 | 3 | 30 |
| Total | 100 | 100 | 100 | 100 |

As fases de concepção e transição, porém, podem variar livremente dentro dos intervalos definidos. Seguem alguns exemplos de fatores que podem aumentar ou diminuir a duração dessas fases:

- a) Requisitos bem definidos logo no início do projeto (por exemplo, criar um substituto para um sistema que já existe) fazem que a fase de concepção seja bem menor (em tempo e esforço) do que seu valor nominal.
- b) Se o sistema, depois de pronto, vai mudar a maneira como as pessoas trabalham, então a fase de transição deverá ser bem maior.
- c) Se existem importantes riscos técnicos, a fase de concepção será maior.
- d) Se a comunidade de usuários é grande e heterogênea, a fase de concepção deverá ser maior.
- e) Se houver necessidade de integração com hardware e sistemas legados existentes, então a fase de transição será maior.

Em resumo, mais incerteza em relação aos requisitos implica uma fase de concepção maior do que o valor nominal. E implantações complexas do sistema implicam uma fase de transição maior.

O manual de CII (Boehm B., 2000)⁴ apresenta ainda outras informações, como, por exemplo, a forma de estimar a duração relativa de cada disciplina e atividade do UP nas diferentes fases, o que é resumido na Tabela 7.36.

7.3.4 CALIBRAGEM DO MODELO

Os valores das constantes de CII foram definidos a partir de um conjunto de projetos-base. Para obter melhores resultados em uma empresa específica é necessário que essas constantes sejam calibradas para os parâmetros específicos da empresa. Além disso, com o passar do tempo, esses valores também poderão mudar, exigindo novas calibragens.

⁴Disponível em: <csse.usc.edu/csse/research/COCOMOII/cocomo2000.0_CII_modelman2000.0.pdf> . Acesso em: 21 jan. 2013.

TABELA 7.37 Exemplo de calibragem para a constante A

| Real | $KSLOC^S * \prod_{i=1}^n M_i$ | $\ln(\text{Real})$ | $\ln(KSLOC^S * \prod_{i=1}^n M_i)$ | $\ln(\text{Real}) - \ln(KSLOC^S * \prod_{i=1}^n M_i)$ |
|--------|-------------------------------|--------------------|------------------------------------|---|
| 1854,6 | 686,7 | 7,53 | 6,53 | 0,99 |
| 258,5 | 94,3 | 5,55 | 4,55 | 1,01 |
| 201,0 | 77,7 | 5,30 | 4,35 | 0,95 |
| 58,9 | 20,3 | 4,08 | 3,01 | 1,07 |
| 9661,0 | 3338,8 | 9,18 | 8,11 | 1,06 |
| 7021,3 | 2753,5 | 8,86 | 7,92 | 0,94 |
| 91,7 | 38,9 | 4,52 | 3,66 | 0,86 |
| 689,7 | 301,1 | 6,54 | 5,71 | 0,83 |

X=0,96

A=2,62

A constante A , por exemplo, na equação geral de estimação de esforço, tem seu valor definido em 2,94:

$$E = A * KSLOC^S * \prod_{i=1}^n M_i$$

Para obter-se um valor mais adequado ao ambiente local de trabalho, recomenda-se ter realizado pelo menos cinco projetos para os quais haja a estimativa e o valor real de esforço realizado. A Tabela 7.37, adaptada de Boehm (2000), mostra os dados obtidos para oito projetos.

A primeira coluna (*Real*) mostra o esforço real de cada projeto em desenvolvedor-mês (fases de elaboração e construção). A segunda coluna apresenta o cálculo da estimativa não ajustada, ou seja, a fórmula do esforço total sem a constante A :

$$KSLOC^S * \prod_{i=1}^n M_i$$

Esses são os valores básicos para o cálculo.

As duas colunas seguintes apresentam o logaritmo natural (*ln*) do valor real do esforço e da estimativa não ajustada. Por fim, a última coluna apresenta a diferença entre essas duas colunas.

No final da tabela, o valor X corresponde à média das diferenças (última coluna). A constante A é calculada como o antilogaritmo dessa média, ou seja, $A = e^x$. Esse exemplo mostra que nesse ambiente local a constante A deveria ser 2,62, e não 2,94.

Boehm (2000) também mostra como calibrar a distribuição das estimativas por atividade e fase, para o ambiente local, o que pode ser particularmente interessante quando essas estimativas são efetivamente usadas para calcular o esforço dentro dos ciclos iterativos.

7.3.5 QUESTIONÁRIO EPML

O questionário EPML (*Estimated Process Maturity Level*) é uma interessante ferramenta não só para avaliar o fator PMAT, mas também como autoavaliação da empresa em relação às boas práticas de processo. Principalmente por essa segunda razão, ele é reproduzido nesta seção.

O questionário subdivide-se em 18 áreas-chave, correspondentes às áreas de avaliação de processo do CMM. Cada pergunta deve ser respondida com uma das opções a seguir:

- a) *Quase sempre*: quando os objetivos são consistentemente obtidos e bem estabelecidos em procedimentos operacionais padrão (mais de 90% das vezes).
- b) *Frequentemente*: quando os objetivos são obtidos com relativa frequência, mas algumas vezes são omitidos por conta de circunstâncias difíceis (entre 60 e 90% das vezes).
- c) *Metade do tempo*: quando os objetivos são obtidos em cerca de metade das vezes (entre 40 e 60% das vezes).
- d) *Ocasionalmente*: quando os objetivos são obtidos algumas vezes, mas com pouca frequência (entre 10 e 40% das vezes).
- e) *Raramente*: quando os objetivos raramente ou nunca são obtidos (menos de 10% das vezes).
- f) *Não se aplica*: quando se tem o conhecimento necessário sobre a organização, o projeto e a área-chave, mas entende-se que área-chave não se aplica às circunstâncias.
- g) *Não sabe*: quando não se sabe o que responder à área-chave.

Deve ser dada uma resposta para cada uma das 18 áreas-chave:

- a) *Gerenciamento de requisitos*: Os requisitos do software são controlados a ponto de se estabelecer uma *baseline* para uso da gerência de engenharia de software? Os planos, produtos e atividades relacionados ao software são mantidos consistentes com os requisitos do software?
- b) *Planejamento de projeto de software*: Estimativas de esforço são documentadas para uso em planejamento e rastreamento de projeto de software? As atividades são planejadas e documentadas? Grupos afetados e indivíduos concordam formalmente com seus entendimentos (*commitments*) relacionados ao projeto de software?
- c) *Rastreamento e supervisão de projeto de software*: Resultados de desempenho reais são rastreados em relação aos planos de software? Ações corretivas são tomadas e gerenciadas até o final, quando os resultados e desempenho reais se desviam significativamente dos planos de software? Mudanças nos entendimentos do software são acordadas formalmente pelos grupos ou indivíduos afetados?
- d) *Gerenciamento de subcontratos de software*: O contratador seleciona subcontratados qualificados? O contratador e o subcontratado concordam formalmente com seus entendimentos um com outro? O contratador e o subcontratado mantêm comunicações constantes? O contratador rastreia os resultados e desempenho reais do subcontratado em relação aos seus empreendimentos?
- e) *Garantia de qualidade de software*: As atividades de garantia de qualidade de software são planejadas? A aderência de produtos e atividades de software aos padrões, procedimentos e requisitos aplicáveis é verificada objetivamente? Grupos e indivíduos afetados são informados das atividades e resultados de garantia de qualidade de software? Assuntos de não conformação que não podem ser resolvidos dentro do projeto de software são levados à gerência superior?
- f) *Gerenciamento de configuração de software*: As atividades de gerenciamento de configuração de software são planejadas? Os produtos de trabalho selecionados são identificados, controlados e disponibilizados? Mudanças nos produtos de trabalho identificadas são controladas? Os grupos e indivíduos afetados são informados do *status* e do conteúdo das *baselines* de software?
- g) *Foco do processo organizacional*: Atividades de desenvolvimento e melhoria de processo de software são coordenadas entre as diferentes partes da organização? Os pontos fortes e fracos do processo de software usado são identificados em relação a um processo-padrão? As atividades de desenvolvimento e melhoria de processo em nível organizacional são planejadas?
- h) *Definição de processo organizacional*: É desenvolvido e mantido um processo de software padrão para a organização? Informação relacionada ao uso do processo de software padrão pelos projetos é coletada, revisada e disponibilizada?
- i) *Programa de treinamento*: As atividades de treinamento são planejadas? É fornecido treinamento para o desenvolvimento de habilidades e conhecimentos necessários para realizar a gerência e os papéis técnicos na área de software? Os indivíduos do grupo de engenharia de software e dos grupos relacionados recebem o treinamento necessário para desempenhar seus papéis?
- j) *Gerenciamento integrado de software*: O processo de software definido para o projeto é uma versão especializada do processo de software padrão da organização? O projeto é planejado e gerenciado de acordo com o processo de software definido para ele?

- k) Engenharia de produto de software:** As atividades de engenharia de software são definidas, integradas e consistentemente realizadas para produzir o software? Os produtos de trabalho são consistentes uns com os outros?
- l) Coordenação intergrupos:** Os requisitos do usuário são acordados por todos os grupos afetados? Os entendimentos entre os grupos de engenharia são acordados pelos outros grupos afetados? Os grupos de engenharia identificam, rastreiam e resolvem assuntos intergrupos?
- m) Revisões:** Atividades de revisão são planejadas? Defeitos nos produtos de trabalho são identificados e removidos?
- n) Gerenciamento quantitativo de processo:** As atividades de gerenciamento quantitativo de processo são planejadas? O desempenho de processo para o projeto definido é controlado quantitativamente? A capacidade do processo de software padrão da organização é conhecida em termos quantitativos?
- o) Gerenciamento da qualidade de software:** As atividades de gerenciamento de qualidade do projeto de software são planejadas? Objetivos mensuráveis para a qualidade de produto de software e suas prioridades são definidos? O progresso real rumo a obter as metas de qualidade para os produtos de software é quantificado e gerenciado?
- p) Prevenção de defeitos:** Atividades de prevenção de defeitos são planejadas? Causas comuns de efeitos são detectadas e identificadas? Causas comuns de defeitos são priorizadas e sistematicamente eliminadas?
- q) Gerenciamento de mudança tecnológica:** A incorporação de mudanças na tecnologia é planejada? Novas tecnologias são avaliadas para determinar seu efeito na qualidade e produtividade? Novas tecnologias apropriadas são transferidas e praticadas normalmente por toda a organização?
- r) Gerenciamento de mudança de processo:** A melhoria de processo contínua é planejada? Toda a organização participa das atividades de melhoria de processo de software? O processo de software padrão da organização e os projetos definidos são continuamente melhorados?

Para obter o valor de EPML (entre 0 e 5) que vai permitir avaliar o multiplicador PMAT é necessário transformar as respostas dadas anteriormente em um número. Inicialmente, eliminam-se áreas-chave para as quais a resposta foi “Não se aplica” ou “Não sei”. Fica-se, então, com n respostas válidas. Mas deve-se tomar cuidado, porque quanto menor o n , menos confiança se poderá ter no resultado da avaliação. A princípio, as 18 áreas deveriam ser avaliadas.

As respostas dadas são convertidas em números K_i (para $1 \leq i \leq n$) da seguinte maneira:

- a) Quase sempre:** 1,00.
- b) Frequentemente:** 0,75.
- c) Metade do tempo:** 0,50.
- d) Ocasionalmente:** 0,25.
- e) Raramente:** 0,01.

Então, EPML é calculado através da seguinte fórmula:

$$EPML = 5 * \frac{\left(\sum_{i=1}^n K_i \right)}{n}$$

Finalmente, o valor é arredondado para o valor inteiro mais próximo e aplicado na Tabela 7.11 para obter-se a avaliação de PMAT.

7.4 Pontos de Função

A técnica de *Análise por Pontos de Função* (APF), ou *Function Point Analysis* (Albrecht & Gaffney Jr., 1983)⁵, é também uma técnica paramétrica para estimativa de esforço para desenvolvimento de software. Porém, ao contrário de COCOMO, ela não se baseia em linhas de código, mas em requisitos.

A análise de pontos de função é aplicável, portanto, a partir do momento em que os requisitos funcionais do software tenham sido definidos. Esses requisitos serão convertidos em valores numéricos, que, depois de ajustados à

⁵Disponível em: <ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1703110>. Acesso em: 21 jan. 2013.

capacidade da empresa desenvolvedora, representarão o esforço necessário para desenvolver o sistema. Assim, a medida obtida pela técnica é, a princípio, independente da linguagem de programação e de tecnologia empregada.

A APF pode ser aplicada para medir o tamanho de um sistema antes de desenvolvê-lo, de forma que seu custo seja previsto mais adequadamente. Além disso, pode ser aplicada também a processos de manutenção, permitindo se estimar o esforço necessário para implantar determinada alteração no software, especialmente se for a adição de uma nova funcionalidade.

A técnica também pode ser usada para calcular o custo-benefício de software ou componentes de software comprados, uma vez que a divisão do preço do produto pelo número de pontos de função que ele implementa dá uma ideia de seu custo relativo. Então, o custo por ponto de função pode ser visto como um fator de normalização para a comparação de produtos de software.

Como é baseada em requisitos implementados, e não no número de linhas de código produzidas, a técnica é mais adequada para medir a produtividade de uma equipe de desenvolvimento, pois ferramentas de produtividade e reusabilidade permitirão implementar mais funcionalidades perceptíveis para o cliente com menos linhas de código.

Existem três contagens específicas de pontos de função:

- a) *Contagem para desenvolvimento de projeto*: é usada para estimar o esforço para o desenvolvimento de um novo projeto.
- b) *Contagem para melhoria de projeto* (Seção 14.5.8): é usada para a evolução de software, em que se contam as funcionalidades adicionadas, alteradas e removidas. A técnica é aplicável apenas para a manutenção adaptativa, já que a manutenção corretiva e a manutenção perfectiva são muito imprevisíveis (Seção 14.2).
- c) *Contagem de aplicação*: é usada para contar pontos de função de aplicações existentes. Pode ter vários objetivos, dentre eles estimar o tamanho funcional da aplicação de forma a relativizar outras métricas (Seção 9.5). Pode, por exemplo, ser mais realista conhecer o número de defeitos por ponto de função do que simplesmente o número de defeitos do software.

A contagem de pontos de função segue um método composto por seis passos, que serão explicados detalhadamente a seguir:

- a) Determinar o tipo de contagem (desenvolvimento, melhoria ou aplicação existente).
- b) Determinar os limites da aplicação (escopo do sistema).
- c) Identificar e atribuir valor em pontos de função não ajustados para as transações sobre dados (entradas, consultas e saídas externas).
- d) Identificar e atribuir valor em pontos de função não ajustados (UFP) para os dados estáticos (arquivos internos e externos).
- e) Determinar o fator de ajuste técnico (VAF).
- f) Calcular o número de pontos de função ajustados (AFP).

Após esse passo, ainda é possível obter o esforço e o custo total do projeto, sua duração linear ideal e o tamanho médio de equipe.

A técnica é divulgada e normatizada internacionalmente pelo IFPUG (*International Function Point Users Group*)⁶ e no Brasil pelo BFPUG (*Brazilian Function Point Users Group*)⁷. É reconhecida como métrica de software pela ISO na norma ISO/IEC 20926 – *Software Engineering – Function Point Counting Practices Manual*⁸.

Além do método de contagem do IFPUG, apresentado nas seções seguintes, existem outros dois métodos internacionalmente relevantes: NESMA⁹, da associação holandesa de métricas, e Mark II (Symons, 1988)¹⁰, ou MK II, mantido pela associação inglesa de métricas. Ao contrário do manual de contagem do IFPUG, que deve ser adquirido, os manuais dessas duas técnicas podem ser obtidos gratuitamente em seus sites, bastando, para isso, fazer registro gratuito na respectiva associação.

⁶Disponível em: <www.ifpug.org/> . Acesso em: 21 jan. 2013.

⁷Disponível em: <www.bfpug.com.br/> . Acesso em: 21 jan. 2013.

⁸Disponível em: <www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=35582> . Acesso em: 21 jan. 2013.

⁹Disponível em: <www.nesma.nl/section/home/> . Acesso em: 21 jan. 2013.

¹⁰Disponível em: <www.ceng.metu.edu.tr/~e120353/HoughTransform/FP_analysis.pdf> . Acesso em: 21 jan. 2013.

7.4.1 INTERPRETAÇÃO E CLASSIFICAÇÃO DOS REQUISITOS COMO FUNÇÕES

A APF é baseada na contagem de pontos para cada uma das funções do sistema. Primeiramente deve-se ter em mente que apenas funcionalidades visíveis para o usuário devem ser contadas. Então, não é todo e qualquer requisito que conta. Se um requisito menciona apenas algum cálculo interno, não deve ser contado como função, embora possivelmente vá aparecer como parte de outra função.

Apenas transferências de informação para dentro e para fora do escopo do sistema (e arquivos de dados acessíveis pelo usuário) são considerados funções.

Um requisito pode ter mais de uma função representada nele. Se o requisito fala, por exemplo, em manter um cadastro de clientes e de produtos, então, na verdade, são dois cadastros, que devem ser interpretados como dois arquivos individuais, cada qual com sua contagem de pontos.

Assim, um primeiro passo para o uso da técnica é tomar os requisitos, eliminar aqueles que são meramente funções internas (ou incorporá-los às funções a que eventualmente pertençam) e subdividir aqueles que representam mais de uma função, até obter uma lista de funções individuais visíveis para o usuário.

Embora trabalhos como o de Longstreet (2012)¹¹ procurem interpretar essas funções em termos de interface com o usuário, o ideal é que sejam identificadas nos requisitos funcionais, ou seja, na sua forma mais essencial e independente de tecnologia.

A técnica APF avalia as duas naturezas dos dados:

- a) *Dados estáticos*, ou seja, a representação estrutural dos dados, na forma de arquivos internos ou externos.
- b) *Dados dinâmicos*, ou seja, a representação das transações sobre os dados, na forma de entradas, saídas e consultas externas.

As transações (entradas, saídas e consultas externas) devem ser processos elementares, isso é, de único passo. Uma transação é a menor unidade de atividade que faça sentido do ponto de vista do usuário e, o que é mais importante, deixe o sistema em um estado consistente.

De outro lado, os arquivos internos ou externos devem ser elementos complexos de informação reconhecíveis pelo usuário (ou seja, não são necessariamente representações físicas internas). Podem ser considerados arquivos internos e externos os elementos de informação que possam ser representados por classes do modelo conceitual, em orientação a objetos, ou tabelas, no modelo relacional, por exemplo. Arquivos internos são aqueles mantidos pela própria aplicação, enquanto arquivos externos são aqueles usados pela aplicação, mas mantidos externamente.

O que vai determinar se um arquivo é interno ou externo, portanto, é a definição da *borda do sistema*, ou seja, os limites daquilo que é considerado interno ao sistema que está sendo desenvolvido ou analisado e de outros sistemas. No caso de aplicações cliente/servidor, deve-se considerar que a borda do sistema passa por ambos os módulos, porque nenhum deles, isoladamente, se constitui em uma aplicação completa com significado para o usuário.

Aqui convém fazer uma ressalva, já que o método APF original diferencia os *Record Element Types* (RET), que correspondem a um subconjunto de dados reconhecível pelo usuário dentro de um arquivo interno ou externo, dos *File Types Referenced* (FTR), que podem conter os RET. Fazendo-se um paralelo com orientação a objetos, os RET seriam classes quaisquer, enquanto os FTR seriam classes que não são componentes nem agregados de outras classes. Por exemplo, se um CD é composto por músicas, então apenas o CD poderia ser um FTR, mas tanto o CD quanto a música poderiam ser RET.

Assim, um FTR é uma classe que, caso participe de uma composição ou agregação, ocupa o lugar mais alto da hierarquia. Um FTR pode *ter* componentes, mas não pode *ser* componente. Essa distinção é importante, porque o cálculo da complexidade das transações (entrada, saída e consulta) é baseado em FTR, enquanto o cálculo da complexidade dos arquivos internos e externos é baseado em RET. Resumindo, a complexidade das transações contabiliza apenas as classes no mais alto nível de uma hierarquia de composição, enquanto a complexidade de arquivos contabiliza quaisquer classes, inclusive as componentes.

As funções identificadas nos requisitos devem, então, ser classificadas de acordo com a sua natureza:

- a) *Entradas externas*: entradas de dados ou controle que têm como consequência a alteração do estado interno das informações do sistema.

¹¹Disponível em: <www.softwaremetrics.com/Function%20Point%20Training%20Booklet%20New.pdf> . Acesso em: 21 jan. 2013.

- b) *Saídas externas*: saídas de dados que podem ser precedidas ou não da entrada de parâmetros. Pelo menos um dos dados de saída deve ser derivado, ou seja, calculado.
- c) *Consultas externas*: saídas de dados que podem ser precedidas ou não da entrada de parâmetros. Os dados devem sair da mesma forma como estavam armazenados, sem transformações ou cálculos.
- d) *Arquivo interno*: elemento do modelo conceitual percebido pelo usuário e mantido internamente pelo sistema.
- e) *Arquivo externo*: elemento do modelo conceitual percebido pelo usuário e mantido externamente por outras aplicações.

Entradas externas são, portanto, funções que pegam dados ou controle do usuário ou de outras aplicações e levam para dentro do sistema. A função deve ter como objetivo armazenar, alterar ou remover dados de forma direta (alterando os dados no sistema) ou indireta (solicitando a outra aplicação que faça a alteração nos dados). Ou seja, dados passados pelo usuário com o único fim de servir de *parâmetro* para uma consulta ou saída não devem ser considerados entradas.

De outro lado, um comando que passa o identificador de um registro (por exemplo, o CPF de um cliente) para deletar um conjunto de dados (por exemplo, deletar o cliente) pode ser considerado entrada, pois é um controle que remove dados, ou seja, faz uma mudança no estado interno das informações do sistema.

Assim, entradas são funções que adicionam, alteram ou deletam informação no sistema. Seus argumentos são os dados passados como parâmetro para localizar os objetos a serem alterados e os parâmetros que correspondem aos novos valores, quando for o caso.

Algumas vezes, as operações de alteração e exclusão ocorrem em dois passos, ou seja, são precedidas de uma consulta. Essa consulta é contabilizada à parte, como uma consulta externa. Por exemplo, um usuário que deseja fazer a alteração do cadastro de um cliente primeiramente entra com o CPF desse cliente e visualiza seus dados na tela (consulta externa); em seguida, ele altera os dados que deseja e salva as novas informações (entrada externa).

Além disso, se uma entrada externa puder produzir um conjunto de mensagens de erro, por conta de possíveis exceções, cada mensagem vai contar como um argumento da entrada (conforme será visto mais adiante). Essas mensagens não devem ser contabilizadas como saídas independentes.

Saídas externas são funções que pegam dados de dentro do sistema e apresentam ao cliente ou enviam a outras aplicações em sua forma original ou transformada, sendo que pelo menos um valor derivado (calculado) deve existir para que seja uma saída externa.

As saídas, portanto, são os fluxos de informação para fora do sistema. Não se conta como saída externa uma função que simplesmente pega dados do sistema e os apresenta da forma como estão (isso será contabilizado como uma *consulta externa*). Mas se houver qualquer tipo de operação matemática ou lógica sobre esses dados, então a função poderá ser considerada uma saída.

Uma saída externa também pode ter parâmetros. Por exemplo, passam-se o CPF de um cliente e uma data e, a partir desses parâmetros, se obtém um relatório de todas as vendas realizadas para esse cliente a partir da data definida e o seu total. Como a data e o CPF são usados para filtrar as vendas que devem aparecer no relatório e fazer a totalização, essa função deve ser considerada saída externa, e não mera consulta.

A *consulta externa* consiste da apresentação de dados da mesma forma como foram armazenados, sem cálculos ou transformações. Normalmente, a consulta inclui parâmetros de entrada, como passar o CPF de uma pessoa para obter seus dados de cadastro (isso é uma simples consulta).

Um *arquivo interno* é uma informação complexa (uma classe, em orientação a objetos) do tipo FTR, ou seja, uma classe que não é componente de outras classes, embora possa ter seus próprios componentes.

Um *arquivo externo* é uma informação complexa (uma classe, em orientação a objetos) do tipo FTR que é mantida em outras aplicações, ou seja, não é gerenciada pela aplicação que se vai desenvolver.

Como os arquivos internos e externos devem ser do tipo FTR, então, por exemplo, se o modelo conceitual possui uma classe CD que agrupa a classe Música, apenas a classe CD será considerada arquivo interno ou externo, pois Música faz parte de CD (por agregação ou composição), não sendo considerada um arquivo interno ou externo à parte.

A maioria dos arquivos internos e externos tem associadas transações de consulta, entrada e saída, que são contabilizadas à parte.

Note que, quando da verificação da complexidade dessas funções, apenas a complexidade visível pelo usuário será considerada. A complexidade algorítmica interna não é aplicada às funções individuais (até porque, a princípio, ela pode não ser conhecida), mas estimada para o projeto como um todo, da mesma forma que os fatores multiplicadores de esforço de CII (Seção 7.3.2).

7.4.2 UFP – PONTOS DE FUNÇÃO NÃO AJUSTADOS

Uma vez determinado o tipo da função (transação ou arquivo), sua complexidade vai ser calculada a partir dos seguintes fatores:

- a) *Registro (RET – Record Element Type)*: corresponde a um subconjunto de dados reconhecível pelo usuário dentro de um arquivo interno ou externo (uma classe qualquer).
- b) *Arquivo (FTR – File Types Referenced)*: corresponde a um arquivo interno ou externo, usado em uma transação (uma classe que não seja componente de outra).
- c) *Argumento (DET – Data Element Type)*: corresponde a uma unidade de informação (um campo), a princípio indivisível e reconhecível pelo usuário; normalmente seria o campo de uma tabela, o atributo de uma classe ou o parâmetro de uma função.

A complexidade das transações é determinada pela quantidade de arquivos FTR e argumentos DET. Já a complexidade dos arquivos internos e externos é determinada pela quantidade de registros RET e argumentos DET.

No caso das entradas externas, os DET podem ser os campos de entrada de informação, mensagens de erro e botões que podem ser pressionados, por exemplo.

No caso das saídas externas, os DET podem ser os campos em um relatório, valores calculados, mensagens de erro e cabeçalhos de colunas que são lidos de um arquivo interno.

No caso das consultas externas, os DET podem ser os campos usados para pesquisa e os campos mostrados como resposta à consulta. Se houver vários botões para fazer a consulta de maneira diferente, cada um deles também deve contar como um DET.

Note que as saídas e consultas podem ter tanto dados de entrada (parâmetros) como dados de saída (resultados). Normalmente, as entradas externas só têm campos de entrada (a informação que se vai armazenar), mas as mensagens de erro potencialmente produzíveis por uma entrada também podem ser contabilizadas como um dado DET.

A complexidade de uma transação é, portanto, calculada a partir da quantidade de FTR e DET. O FTR corresponde ao número de classes (que não são componentes ou agregados de outra classe) do modelo conceitual que contêm as informações de entrada e/ou saída como atributos. Por exemplo, se uma função de entrada passa uma data que será armazenada como atributo da classe X e um valor numérico que será armazenado como atributo de uma classe Y, então existem duas classes envolvidas e o FTR, nesse caso, é igual a 2. Se os dois valores vão ser armazenados como atributos de uma mesma classe, ou como atributos de dois componentes de uma mesma classe, ou ainda como atributos de uma classe e uma de suas componentes, então considera-se que há apenas um FTR. Por exemplo, uma função que grava o nome de um CD e o nome das músicas deste CD tem apenas um FTR (o CD), mesmo que as músicas sejam consideradas em uma classe à parte. Como uma música faz parte do CD, ela não pode ser contabilizada como FTR.

O segundo valor usado para calcular a complexidade de uma transação é o seu número de argumentos, ou DET, ou seja, a quantidade de valores de entrada e/ou saída, independentemente das classes a que ele pertença. Assim, por exemplo, uma função de entrada que envia 5 valores ao sistema terá 5 argumentos, e o DET será igual a 5. Uma consulta que passa 2 parâmetros e recebe 8 valores como retorno tem 10 argumentos DET.

Deve-se ainda saber que o que conta, no caso de DET, é o *tipo* de valor. Caso a função envie ou receba uma lista de valores de um mesmo tipo, a lista conta uma única vez. Por exemplo, uma função que passar um nome de pessoa e uma lista de telefones terá 2 argumentos (mesmo que essa lista tenha 10 telefones, ela conta uma única vez). De outro lado, se algum tipo de cálculo for feito com a lista de valores, então cada cálculo contará como um argumento diferente. Por exemplo, uma saída que apresente uma lista de notas, a média e o desvio-padrão dessas notas deverá ter considerados 3 argumentos DET.

As funções de entrada têm sua complexidade calculada de acordo com a Tabela 7.38.

As saídas e consultas têm sua complexidade calculada de acordo com a Tabela 7.39.

Já os arquivos internos e externos têm sua complexidade calculada em função de classes quaisquer (inclusive componentes de outras classes), ou seja, RET. Além de RET, deve-se usar DET, como no caso de transações para determinar a complexidade dos arquivos. Deve ser, então, aplicada a Tabela 7.40.

Embora a maioria dos arquivos do tipo CRUD se relate com uma única classe (exemplo, um cadastro de clientes), algumas informações, por terem subcomponentes, podem acabar se relacionando com mais de uma classe, apesar de ainda poderem ser consideradas um cadastro único. É o caso, por exemplo, do padrão Mestre-Detalhe,

TABELA 7.38 Complexidade funcional de entradas externas

| Argumentos DET | | | |
|------------------|-------|--------|------------|
| Classes FTR | 1 a 4 | 5 a 15 | 16 ou mais |
| 0 a 1 | Baixa | Baixa | Média |
| 2 | Baixa | Média | Alta |
| 3 ou mais | Média | Alta | Alta |

TABELA 7.39 Complexidade funcional de saídas e consultas

| Argumentos DET | | | |
|------------------|-------|--------|------------|
| Classes FTR | 1 a 5 | 6 a 19 | 20 ou mais |
| 0 a 1 | Baixa | Baixa | Média |
| 2 a 3 | Baixa | Média | Alta |
| 4 ou mais | Média | Alta | Alta |

TABELA 7.40 Complexidade funcional de arquivos internos e externos

| Argumentos DET | | | |
|------------------|--------|---------|------------|
| Classes RET | 1 a 19 | 20 a 50 | 51 ou mais |
| 1 | Baixa | Baixa | Média |
| 2 a 5 | Baixa | Média | Alta |
| 6 ou mais | Média | Alta | Alta |

TABELA 7.41 Pontos de função não ajustados por tipo e complexidade de função

| Complexidade funcional | | | |
|------------------------|-------|-------|------|
| Tipo de função | Baixa | Média | Alta |
| Entrada | 3 | 4 | 6 |
| Saída | 4 | 5 | 7 |
| Consulta | 3 | 4 | 6 |
| Arquivo interno | 7 | 10 | 15 |
| Arquivo externo | 5 | 7 | 10 |

em que duas classes são tratadas juntas como um único cadastro, como livro e capítulo, CD e música, passagem e trecho etc. Assim, um arquivo interno ou externo será a classe de mais alto nível na hierarquia de composição (FTR), e o número de classes RET da Tabela 7.40 será o número de componentes ou subcomponentes dela mais um. Por exemplo, CD poderia ser um arquivo interno (FTR) com duas classes RET: CD e Música.

Depois de determinar a complexidade de cada transação e arquivo, pode-se obter seu número de pontos de função não ajustados UFP aplicando a Tabela 7.41.

O número UFP de pontos de função não ajustados para o sistema como um todo será simplesmente a soma dos pontos de função não ajustados obtidos para cada uma das funções do sistema.

A identificação das transações e arquivos de um sistema é o ponto-chave para determinar sua complexidade. Porém, muitas vezes, requisitos são incompletos ou muito genéricos para que uma boa contagem possa ser feita.

A própria técnica de pontos de função pode ajudar a melhorar a qualidade dos requisitos. Em primeiro lugar, ela precisa dos argumentos DET; então será necessário verificar se os requisitos efetivamente listam todas as informações elementares necessárias para realizar cada função.

Em segundo lugar, a identificação de arquivos e transações caminham juntas. Uma vez identificado um arquivo interno ou externo (classe), pode-se imaginar que pelo menos as transações mais básicas para esse arquivo devam existir. Essas transações, usualmente chamadas CRUD ou CRUCL, implicam criação (entrada externa), alteração (entrada externa), remoção (entrada externa), consulta (consulta externa) e listagem (consulta externa), nem sempre estão todas presentes na aplicação. É possível, por exemplo, que a aplicação crie um arquivo de dados que só vai ser consultado por outra aplicação, mas sempre deve-se pensar se elas existem ou não como transações.

Nem sempre, porém, um arquivo sofre apenas transações tão básicas em seu ciclo de vida. Em uma livraria virtual, por exemplo, um livro não será apenas criado, alterado, consultado e deletado. Em determinados momentos ele será colocado em oferta, reservado ou vendido. Essas transações podem ser consideradas simples alterações, já incluídas no CRUCL?

A técnica responde a essa questão da seguinte forma: se uma transação de alteração de um objeto, como “colocar em oferta”, implica usar apenas um conjunto ou um subconjunto de DET e os arquivos FTR ou um subconjunto deles, já considerados na operação de alteração do CRUCL, então a alteração será contemplada pelas transações CRUCL. Por exemplo, se colocar um livro em oferta implica simplesmente mudar o valor de um de seus atributos, o que já pode ser feito pela operação CRUCL de alteração, então “colocar um livro em oferta” não será uma nova função nem vai contar pontos de função.

De outro lado, se a alteração implica usar dados DET não incluídos no conjunto da operação de alteração ou ainda arquivos FTR não incluídos na alteração CRUCL, então a transação deverá ser considerada uma nova função e contar seus próprios pontos de função. Por exemplo, se “colocar livro em oferta” implica obter dados de um arquivo de ofertas padrão, o que não é feito quando da operação de alteração CRUCL, então trata-se de uma nova função.

A título de exemplo de aplicação da técnica, considere-se a seguinte lista de requisitos:

- a) O sistema deve permitir o gerenciamento (CRUCL) de informações sobre livros e usuários. Dos livros incluem-se título, ISBN, autor, número de páginas, editora e ano de publicação. Dos usuários incluem-se nome, documento, endereço, telefone e e-mail.
- b) O sistema deve permitir o registro de empréstimos, em que são informados o documento do usuário e o ISBN de cada livro.
- c) Quando um empréstimo for executado, o sistema deve armazenar as informações em uma tabela relacional, usando chaves estrangeiras para identificar o usuário e os livros.
- d) Após o registro de um empréstimo, deve ser impresso um recibo com o nome do usuário, além de título e data de devolução prevista para cada livro, que deve ser calculada como a data atual somada ao prazo do livro.

Analizando-se o primeiro requisito, percebe-se que se trata de dois cadastros independentes com suas operações CRUCL. O segundo requisito é uma entrada externa. O terceiro requisito é uma função interna que não deve ser contabilizada. O quarto requisito será considerado uma saída externa, pois realiza um cálculo para obter a data de entrega em função da data atual e do prazo do livro. As funções a serem consideradas são apresentadas na Tabela 7.42.

Portanto, aplicando-se a técnica às funções identificadas nos requisitos mencionados, considerando que o modelo conceitual tenha as classes Pessoa, Livro, Empréstimo, Editora e Autor, sem nenhuma relação de agregação ou composição entre elas, chega-se a um cálculo de 59 pontos de função não ajustados. Note que, se Editora e Autor não fossem considerados classes (arquivos), mas meramente campos a serem adicionados à classe Livro, a complexidade das funções relacionadas com Livro seria menor, pois haveria menos classes envolvidas.

De outro lado, a consideração dessas informações como classes leva à conclusão de que os requisitos ainda não estão completos, pois as classes Editora e Autor deveriam ser consideradas cadastros e, portanto, arquivos internos ou externos com suas respectivas operações CRUCL. Assim, após essa revisão nos requisitos, o número de pontos de função não ajustado deveria ser recalculado.

7.4.3 AFP – PONTOS DE FUNÇÃO AJUSTADOS

O método de pontos de função não tem fatores de escala, como COCOMO, então presume que o esforço será linear em relação à quantidade de funcionalidades implementadas. Porém, o método possui um conjunto de fatores

TABELA 7.42 Exemplo de identificação de funções a partir de requisitos

| Função | Tipo | FTR | RET | DET | #FTR | #RET | #DET | Complex. | UFP |
|-----------------------------|------------------|---------------------------|-----------------------|--|------|------|------|----------|-----|
| Cadastro de livros | Arquivo interno | | Livro, editora, autor | Título, ISBN, autor, número de páginas, editora, ano de publicação | | 3 | 6 | Baixa | 7 |
| Inserir livro | Entrada externa | Livro, editora, autor | | Título, ISBN, autor, número de páginas, editora, ano de publicação | 3 | | 6 | Alta | 6 |
| Alterar livro | Entrada externa | Livro, editora, autor | | Título, ISBN, autor, número de páginas, editora, ano de publicação | 3 | | 6 | Alta | 6 |
| Excluir livro | Entrada externa | Livro | | Título, ISBN, ano de publicação | 1 | | 3 | Baixa | 3 |
| Consultar livro | Consulta externa | Livro, editora, autor | | Título, ISBN, autor, número de páginas, editora, ano de publicação | 3 | | 6 | Média | 4 |
| Listar livros | Consulta externa | Livro, editora, autor | | Título, ISBN, autor, número de páginas, editora, ano de publicação | 3 | | 6 | Média | 4 |
| Cadastro de usuários | Arquivo interno | | Pessoa | Nome, documento, endereço, telefone, e-mail | | 1 | 5 | Baixa | 7 |
| Inserir usuário | Entrada externa | Pessoa | | Nome, documento, endereço, telefone, e-mail | 1 | | 5 | Baixa | 3 |
| Alterar usuário | Entrada externa | Pessoa | | Nome, documento, endereço, telefone, e-mail | 1 | | 5 | Baixa | 3 |
| Excluir usuário | Entrada externa | Pessoa | | Nome, documento | 1 | | 2 | Baixa | 3 |
| Consultar usuário | Consulta externa | Pessoa | | Nome, documento, endereço, telefone, e-mail | 1 | | 5 | Baixa | 3 |
| Listar usuários | Consulta externa | Pessoa | | Nome, documento | 1 | | 2 | Baixa | 3 |
| Registrar empréstimo | Entrada externa | Pessoa, livro, empréstimo | | Nome da pessoa, ISBN dos livros | 3 | | 2 | Média | 4 |
| Imprimir recibo | Consulta externa | Pessoa, livro, empréstimo | | Nome da pessoa, título do livro, data de hoje, prazo do livro, data de devolução | 3 | | 5 | Baixa | 3 |
| TOTAL | 59 | | | | | | | | |

de ajuste técnico, já que diferentes projetos e equipes poderão produzir funcionalidades em ritmos diferentes. Por exemplo, um simples *front-end* para banco de dados terá suas funções desenvolvidas muito mais rapidamente do que um software para controle de um sistema de distribuição de energia elétrica. Assim, a funcionalidade dada por UFP, ou pontos de função não ajustados, não leva em consideração a complexidade técnica interna das funções, mas apenas a percepção que um usuário tem delas.

Para chegar a valores de esforço de desenvolvimento mais realistas, portanto, será necessário ajustar esse valor a partir dos fatores técnicos.

A técnica de pontos de função sugere 14 fatores de ajuste técnico, conhecidos como GSC (*General Systems Characteristics*). Todos têm o mesmo peso, e a eles deve ser atribuída uma nota de zero a cinco, em que zero significa que o fator não tem nenhuma influência no projeto e cinco significa que o fator tem influência determinante no projeto, sendo os valores de 1 a 4 intermediários.

Os 14 GSC são os seguintes (mais detalhes na Seção 7.4.5):

- a) Comunicação de dados.
- b) Processamento de dados distribuído.
- c) *Performance*.
- d) Uso do sistema.
- e) Taxa de transações.
- f) Entrada de dados *on-line*.
- g) Eficiência do usuário final.
- h) Atualização *on-line*.
- i) Processamento complexo.
- j) Reusabilidade.
- k) Facilidade de instalação.
- l) Facilidade de operação.
- m) Múltiplos locais.
- n) Facilidade para mudança.

A avaliação dos GSC é feita para o projeto como um todo (não para cada função). Então, como são 14 GSC e cada um receberá uma nota de 0 a 5, o somatório das notas ficará entre 0 e 70. Esse valor ajustado é conhecido como VAF (*Value Adjustment Factor*):

$$VAF = 0,65 + \left(0,01 * \sum_{i=1}^{14} GSC_i \right)$$

Assim, o somatório dos GSC, multiplicado por 0,01 e somado a 0,65, vai fazer que VAF varie de 0,65 a 1,35.

Esse valor é multiplicado pelo número de UFP para obter o AFP, ou número de pontos de função ajustados:

$$AFP = UFP * VAF$$

Em um projeto no qual todos os fatores técnicos sejam mínimos (nota 0), o AFP será igual a 65% do valor nominal de UFP. Já em um sistema em que todos os fatores técnicos sejam máximos (nota 5) o AFP será igual a 135% do valor nominal de UFP. Um sistema nominal seria aquele no qual todos os fatores técnicos têm nota 3, o que levaria o AFP a ser igual ao UFP.

7.4.4 DURAÇÃO E CUSTO DE UM PROJETO

Uma vez que o AFP do projeto tenha sido calculado, o esforço total será calculado multiplicando-se o AFP pelo índice de produtividade (IP) da equipe. Esse índice deve ser calculado para o ambiente local e pode variar muito em função do ambiente de trabalho, da experiência da equipe e de outros fatores. Assim, o esforço total do projeto é calculado como:

$$E = AFP * IP$$

O IP de uma equipe pode ser calculado da seguinte forma: toma-se um projeto já desenvolvido, para o qual se saiba quanto esforço foi despendido. O esforço E deve ser contado em *desenvolvedor-mês*, ou seja, se 4 pessoas trabalharam durante 10 meses, então o valor do esforço é de 40 desenvolvedor-mês. Para esse mesmo projeto, estimam-se os pontos de função ajustados a partir dos requisitos iniciais (a versão dos requisitos que existia antes de se iniciar o desenvolvimento propriamente dito). Se, por exemplo, eram 800 AFP, então o índice de produtividade da equipe para esse projeto é: $IP = AFP/E = 800/40 = 20$. Nesse caso, o índice de produtividade da equipe é de 20 pontos de função ajustados por desenvolvedor-mês.

Se houver mais de um projeto disponível para cálculo, podem-se somar os AFP de todos os projetos e dividir pelo esforço despendido em todos os projetos, obtendo sua média aritmética.

Já o custo do projeto é calculado como o esforço total multiplicado pelo custo médio da hora do desenvolvedor e ambiente:

$$Custo = E * Custo_{hora}$$

Há um *site*¹² que armazena editais brasileiros de contratação de software em que a medida de custo é o ponto de função. No *site*, o preço por ponto de função varia de 100 a 1.000 reais, a maioria ficando entre 400 e 600 reais, o que pode ser explicado pelo tipo de sistema que se está contratando.

O tempo linear e o tamanho médio da equipe podem ser calculados como em COCOMO (Seção 7.2) ou CII (Seção 7.3).

Outra maneira mais simples de calcular o tempo linear ideal, sem usar KSLOC, é através da seguinte equação:

$$T = 2,5 * \sqrt[3]{E}$$

Para essa equação funcionar, é necessário que o esforço E seja expresso em *desenvolvedor-mês*. O uso de outras unidades, como desenvolvedor-semana ou desenvolvedor-hora, vai provocar distorções no resultado.

O tamanho médio da equipe, nesse caso, será:

$$P = E / T$$

Além do tempo linear ideal, pode-se também falar em tempo mínimo de projeto, caso exista urgência para desenvolvê-lo e seja possível gastar mais recursos para tentar fazer isso num prazo mais curto. O tempo mínimo de um projeto pode ser calculado da seguinte forma:

$$T_{min} = 0,75 * \sqrt[3]{E}$$

Nesse caso, é bem possível que o tamanho médio da equipe deva ser maior do que E/T_{min} . Deve-se também ter em mente que um projeto nessas condições estará sendo gerenciado em condições limite e, portanto, a princípio, apenas equipes muito experientes e organizadas teriam capacidade para dar conta desse ritmo de desenvolvimento.

Todavia, de certa forma essas fórmulas são “mágicas”, em razão da grande variedade de projetos e equipes de desenvolvimento de software. Antes de usá-las seriamente em uma empresa, convém sempre testá-las e ajustá-las para a realidade local.

7.4.5 DETALHAMENTO DOS FATORES TÉCNICOS

Nesta seção será detalhada a interpretação dos fatores técnicos que compõem o VAF (Longstreet, 2012),¹³ para que notas consistentes possam ser atribuídas.

O GSC *comunicação de dados* avalia o grau em que necessidades especiais de comunicação afetam o sistema. Sistemas isolados estariam no extremo inferior de avaliação; já os sistemas que fazem uso intensivo de dados obtidos em outros lugares e que enviam informação para muitos lugares diferentes, através de diferentes protocolos de comunicação, estariam no extremo oposto. As notas são atribuídas assim:

¹²Disponível em: <www.fattocs.com.br/editais.asp>. Acesso em: 21 jan. 2013.

¹³Disponível em: <www.softwaremetrics.com/Function%20Point%20Training%20Booklet%20New.pdf>. Acesso em: 21 jan. 2013.

- a) 0: para aplicações que são somente processamento em *batch* ou que rodam isoladas em um PC.
- b) 1: para aplicações em *batch*, mas com entrada de dados remota *ou* saída remota.
- c) 2: para aplicações em *batch* com entrada de dados remota *e* saída remota.
- d) 3: para aplicações que incluem coleta de dados *on-line* *ou* *front-end* de teleprocessamento para um sistema em *batch* ou sistema de consultas.
- e) 4: para aplicações que são mais do que um *front-end*, mas suportam um único tipo de protocolo de comunicação.
- f) 5: para aplicações que são mais do que um *front-end* e suportam vários tipos de protocolos de comunicação.

O fator *processamento de dados distribuído* avalia o grau em que dados distribuídos são usados pela aplicação. No extremo inferior estão os sistemas que armazenam os dados todos no mesmo lugar ou que, havendo necessidade de transferir dados, nada fazem para automatizar essa transferência. No extremo superior estão as aplicações que distribuem o processamento dinamicamente entre diferentes nodos, escolhendo sempre o melhor nodo possível para efetuar o processamento específico. As notas são atribuídas assim:

- a) 0: para aplicações que não ajudam na transferência de dados ou funções de processamento entre os componentes do sistema.
- b) 1: para aplicações que preparam os dados para o processamento do usuário final em outro componente do sistema, tal como sistemas que geram dados para serem lidos em uma planilha ou arquivo de processador de texto.
- c) 2: para aplicações que preparam dados para transferência e então transferem e processam os dados em outro componente do sistema (não para processamento do usuário final).
- d) 3: para aplicações em que o processamento distribuído e a transferência de dados ocorrem *on-line* e apenas em uma direção.
- e) 4: para aplicações em que o processamento distribuído e a transferência de dados ocorrem *on-line* e nas duas direções.
- f) 5: para aplicações em que as funções são executadas dinamicamente no componente mais apropriado do sistema.

O fator *performance* avalia o grau em que a eficiência do sistema precisa ser considerada em sua construção. Sistemas eficientes sempre são desejáveis, mas esse fator avalia o quanto a eficiência é crítica para o sistema, de forma que se invistam recursos de tempo e dinheiro para melhorar esse aspecto. Um sistema de empréstimo de vídeos em uma videolocadora poderia estar no extremo inferior, porque normalmente trabalha com quantidades de dados pequenas e algoritmos simples, de forma que a eficiência não será uma preocupação. No extremo superior poder-se-ia ter sistemas que trabalham com quantidades absurdamente grandes de dados e/ou precisam apresentar respostas rápidas, como um sistema que interpreta dados de geoprocessamento em tempo real ou um sistema que controla a injeção eletrônica de combustível em um automóvel. As notas são atribuídas assim:

- a) 0: nenhum requisito de *performance* especial foi definido pelo cliente.
- b) 1: requisitos de *performance* foram estabelecidos e revisados, mas nenhuma ação especial precisa ser realizada.
- c) 2: o tempo de resposta e a taxa de transferência são críticos durante as horas de pico. Nenhum *design* especial para utilização de CPU é necessário. O prazo para a maioria dos processamentos é o dia seguinte.
- d) 3: o tempo de resposta e a taxa de transferência são críticos durante o horário comercial. Nenhum *design* especial para utilização de CPU é necessário. Os requisitos de prazo de processamento com sistemas interfaceados são restritivos.
- e) 4: em adição, os requisitos de *performance* são suficientemente restritivos para que se necessite estabelecer tarefas de análise de *performance* durante a fase de *design*.
- f) 5: em adição, ferramentas de análise de *performance* devem ser usadas nas fases de *design*, desenvolvimento e/ou implementação para atender aos requisitos de *performance* do cliente.

O fator *uso do sistema* avalia o grau em que o sistema necessita ser projetado para compartilhar recursos de processamento. No extremo inferior estão as aplicações nas quais não há nenhuma preocupação com o compartilhamento de hardware; no extremo superior estão as aplicações nas quais o hardware estará sendo compartilhado, enquanto existem restrições operacionais da aplicação em si. As notas são atribuídas assim:

- a) 0: nenhuma restrição operacional implícita ou explícita é incluída.
- b) 1: restrições operacionais existem, mas são menos restritivas do que em uma aplicação típica. Nenhum esforço especial é necessário para satisfazer as restrições.

- c) 2: são incluídas algumas considerações sobre tempo e segurança.
- d) 3: requisitos específicos de processador para uma parte específica da aplicação são incluídos.
- e) 4: restrições sobre operações estabelecidas requerem que a aplicação tenha um processador dedicado ou prioridade de tempo no processador central.
- f) 5: em adição, existem restrições especiais na aplicação em relação aos componentes distribuídos do sistema.

O fator *taxa de transações* avalia a quantidade de transações simultâneas esperada. Os limites inferior e superior podem variar em função do desenvolvimento tecnológico do hardware e das redes de comunicação. O que há dez anos poderia ter sido considerado difícil em termos de taxa de transações hoje pode ser trivial. Então, é necessário que se avalie, em termos da tecnologia atual, qual seria a taxa de influência desse fator no desenvolvimento do produto. No limite inferior estariam sistemas com tão poucas transações que a tecnologia atual daria conta delas sem nenhuma preocupação extra. No extremo superior estariam sistemas no limite da tecnologia, os quais poderiam precisar de vários contêineres de processadores servidores para poder atender aos usuários (em 2012, na casa de possíveis milhões de acessos simultâneos). Exemplos de sistemas com essa taxa de acesso são o Facebook^{®14} e o Google^{®15}. As notas são atribuídas assim:

- a) 0: não são antecipados períodos de picos de transações.
- b) 1: períodos de picos de transações (por exemplo, mensal, semestral ou anualmente) são antecipados.
- c) 2: picos de transação semanais são antecipados.
- d) 3: picos de transação diárias são antecipados.
- e) 4: altas taxas de transação são estabelecidas pelo cliente nos requisitos da aplicação ou nos acordos de nível de serviço, as quais são suficientemente altas para necessitar de atividades de análise de *performance* na fase de *design*.
- f) 5: em adição, requer-se o uso de ferramentas de análise de *performance* nas fases de *design*, desenvolvimento e/ou instalação.

O fator *entrada de dados on-line* avalia a porcentagem de informação que o sistema deve obter *on-line*, ou seja, dos usuários em tempo real. No extremo inferior estão os sistemas que tomam toda a informação necessária de arquivos ou repositórios. No extremo superior estão os sistemas nos quais mais de 30% das transações são entradas de dados *on-line*. As notas são atribuídas assim:

- a) 0: todas as transações são processadas em modo *batch*.
- b) 1: 1 a 7% das transações são entradas de dados interativas.
- c) 2: 8 a 15% das transações são entradas de dados interativas.
- d) 3: 16 a 23% das transações são entradas de dados interativas.
- e) 4: 24 a 30% das transações são entradas de dados interativas.
- f) 5: mais de 30% das transações são entradas de dados interativas.

O fator *eficiência do usuário final* avalia o grau em que a aplicação será projetada para melhorar a eficiência do usuário final. Não ter preocupação nenhuma com isso leva o sistema ao limite inferior desse fator. Uma preocupação extrema em melhorar a eficiência do trabalho do usuário leva ao limite superior. As notas são atribuídas a partir de uma lista de itens que podem ser considerados ou não:

- a) Ajuda navegacional (por exemplo, teclas de função, menus gerados dinamicamente etc.).
- b) Menus.
- c) Ajuda e documentação *on-line*.
- d) Movimentação de cursor automatizada.
- e) *Scrolling*.
- f) Impressão remota (a partir de transações *on-line*).
- g) Teclas de função predefinidas.
- h) Tarefas em *batch* submetidas a partir de transações *on-line*.
- i) Seleção por cursor na tela de dados.

¹⁴Disponível em: <www.facebook.com>. Acesso em: 21 jan. 2013.

¹⁵Disponível em: <www.google.com>. Acesso em: 21 jan. 2013.

- j) Alto uso de cores e destaque visual em tela.
- k) Cópias impressas de documentação de usuário de transações *on-line*.
- l) Interface por *mouse*.
- m) Janelas *pop-up*.
- n) Minimização do número de janelas para realizar objetivos de negócio.
- o) Suporte bilíngue (conta como quatro itens).
- p) Suporte multilíngue (conta como seis itens).

As notas são então calculadas em função da lista anterior:

- a) 0: nenhuma das opções anteriores.
- b) 1: de uma a três das opções anteriores.
- c) 2: de quatro a cinco das opções anteriores.
- d) 3: seis ou mais das opções anteriores, mas não há requisitos específicos relacionados à eficiência de usuário final.
- e) 4: seis ou mais das opções anteriores e requisitos estabelecidos para a eficiência de usuário final são suficientemente fortes para requerer a inclusão de atividades de *design* para fatores humanos (por exemplo, minimizar a quantidade de cliques e movimentos de mouse, maximização de *defaults* e uso de *templates*).
- f) 5: seis ou mais das opções anteriores e requisitos estabelecidos para a eficiência de usuário são suficientemente fortes para requerer o uso de ferramentas e processos especiais para demonstrar que os objetivos foram atingidos.

O fator *atualização on-line* avalia o percentual de arquivos internos que podem ser atualizados *on-line*. Se nenhum arquivo interno pode ser atualizado dessa forma, então deve-se atribuir nota mínima a esse fator. Quando todos os principais arquivos podem ser atualizados dessa forma e existem restrições especiais de volume e de segurança, então se está no extremo superior do fator. As notas são atribuídas assim:

- a) 0: nenhuma atualização *on-line*.
- b) 1: é incluída a atualização *on-line* para um a três arquivos. O volume de atualização é baixo e a recuperação é simples.
- c) 2: a atualização *on-line* de quatro ou mais arquivos é incluída. O volume de atualização é baixo e a recuperação é simples.
- d) 3: a atualização *on-line* dos principais arquivos lógicos internos é incluída.
- e) 4: em adição, proteção contra a perda de dados é essencial, e o sistema deve ser especialmente projetado para isso.
- f) 5: em adição, altos volumes de atualização trazem considerações de custo para o processo de recuperação. Procedimentos de recuperação altamente automatizados com intervenção mínima do operador são incluídos.

O fator *processamento complexo* avalia o grau em que a aplicação utiliza processamento lógico ou matemático complexo. No extremo inferior estariam sistemas que apenas armazenam dados e eventualmente fazem algum tipo de soma ou produto; no outro extremo estariam sistemas baseados em inteligência artificial e processamento numérico complexo nas áreas de Física e Engenharia.

Os seguintes componentes são considerados para avaliação da complexidade do processamento da aplicação:

- a) Controle cuidadoso (por exemplo, processamento especial de auditoria) e/ou processamento seguro específico da aplicação.
- b) Processamento lógico extensivo.
- c) Processamento matemático extensivo.
- d) Muito processamento de exceções resultantes de transações incompletas que precisam ser reprocessadas, como transações de caixa-automático incompletas, causadas pela interrupção do teleprocessamento, valores de dados que faltam ou edições que falharam.
- e) Processamento complexo para gerenciar múltiplas possibilidades de entrada e saída, como a multimídia ou a independência de dispositivos.

As notas são, então, atribuídas em função desses itens:

- a) 0: nenhuma das opções anteriores.
- b) 1: qualquer uma das opções anteriores.

- c) 2: quaisquer duas das opções anteriores.
- d) 3: quaisquer três das opções anteriores.
- e) 4: quaisquer quatro das opções anteriores.
- f) 5: todas as cinco opções anteriores.

O fator *reusabilidade* avalia em que grau a aplicação é projetada para ser reusável. No extremo inferior estão aplicações que são desenvolvidas sem nenhuma preocupação com reusabilidade e no extremo superior estão aplicações desenvolvidas para gerar vários componentes parametrizados reusáveis. As notas são atribuídas assim:

- a) 0: não há nenhuma preocupação para produzir código reusável.
- b) 1: código reusável é gerado para uso dentro da própria aplicação.
- c) 2: menos de 10% da aplicação deve considerar mais do que simplesmente as necessidades do usuário.
- d) 3: 10% ou mais da aplicação deve considerar mais do que as necessidades do usuário.
- e) 4: a aplicação deve ser especificamente empacotada e/ou documentada para facilitar o reúso, e a aplicação deve ser personalizável pelo usuário em nível de código-fonte.
- f) 5: a aplicação deve ser especificamente empacotada e/ou documentada para facilitar o reúso, e a aplicação deve ser personalizável por meio de manutenção de usuário baseada em parâmetros.

O fator *facilidade de instalação* avalia em que grau haverá preocupação em facilitar a instalação do sistema e a conversão dos dados. Um sistema para o qual não haja necessidade de conversão de dados e que deva ser instalado sem nenhuma automatização estará no limite inferior. Um sistema que se deva autoinstalar automaticamente e converter automaticamente dados de sistemas legados estará no limite superior. As notas são atribuídas assim:

- a) 0: nenhuma consideração especial foi estabelecida pelo usuário, e nenhum *setup* especial é necessário para a instalação.
- b) 1: nenhuma consideração especial foi estabelecida pelo usuário, mas um *setup* especial é requerido para a instalação.
- c) 2: requisitos de conversão e instalação foram estabelecidos pelo usuário, e guias de conversão e instalação devem ser fornecidos e testados. O impacto da conversão no projeto não é considerado importante.
- d) 3: requisitos de conversão e instalação foram estabelecidos pelo usuário, e guias de conversão e instalação devem ser fornecidos e testados. O impacto da conversão no projeto é considerado importante.
- e) 4: em adição à nota 2, ferramentas de conversão e instalação automática devem ser fornecidas e testadas.
- f) 5: em adição à nota 3, ferramentas de conversão e instalação automática devem ser fornecidas e testadas.

O fator *facilidade de operação* avalia em que grau a aplicação deverá fazer automaticamente processos de *startup*, recuperação de falhas e *backup*. Aplicações que não façam nada disso automaticamente estão no limite inferior, e aplicações que automatizem totalmente os processos de operação estão no limite superior. As notas são atribuídas assim:

- a) 0: nenhuma consideração operacional especial, além dos procedimentos normais de *backup*, foram estabelecidos pelo usuário.
- b) 1-4: um, alguns ou todos os itens a seguir se aplicam ao sistema (devem-se selecionar todos os que se aplicam. Cada item vale um ponto, exceto se for dito o contrário):
 - Processos efetivos de inicialização, *backup* e recuperação devem ser fornecidos, mas a intervenção do operador é necessária.
 - Processos efetivos de inicialização, *backup* e recuperação devem ser fornecidos, e nenhuma intervenção do operador é necessária (conta como dois itens).
 - A aplicação deve minimizar a necessidade de armazenamento em fitas (ou qualquer outro meio de armazenamento *off-line*).
 - A aplicação deve minimizar a necessidade de manuseio de papel.
- c) 5: a aplicação é projetada para operar de forma não supervisionada. “Não supervisionada” significa que não é necessária nenhuma intervenção do operador do sistema, a não ser, talvez, na sua primeira inicialização ou desligamento final. Uma das características da aplicação é a recuperação automática de erros.

O fator *múltiplos locais* avalia o grau em que a aplicação é projetada para funcionar de forma distribuída. No limite inferior estão as aplicações isoladas *stand-alone* e no superior estão as aplicações formadas por

diferentes módulos que rodam em máquinas geograficamente distribuídas ou dispositivos móveis. As notas são atribuídas assim:

- a) 0: os requisitos do usuário não exigem a consideração da necessidade de mais do que um usuário ou instalação.
- b) 1: a necessidade de múltiplos locais deve ser considerada no projeto, e a aplicação deve ser projetada para operar apenas em ambientes idênticos de hardware e software.
- c) 2: a necessidade de múltiplos locais deve ser considerada no projeto, e a aplicação deve ser projetada para operar apenas em ambientes de hardware e software similares.
- d) 3: a necessidade de múltiplos locais deve ser considerada no projeto, e a aplicação é projetada para operar em ambientes de hardware e software diferentes.
- e) 4: o plano de documentação e suporte deve ser fornecido e testado para suportar a aplicação em múltiplos locais, e a aplicação é como descrita nas notas 1 e 2.
- f) 5: o plano de documentação e suporte deve ser fornecido e testado para suportar a aplicação em múltiplos locais, e a aplicação é como descrita na nota 3.

O fator *facilidade para mudança* avalia o grau em que a aplicação é projetada para facilitar mudanças lógicas e estruturais. Aplicações desenvolvidas sem esse tipo de preocupação estão no limite inferior, e aplicações projetadas para acomodar extensivamente mudanças futuras estão no limite superior.

As seguintes características podem se aplicar ao software:

- a) Facilidades de consulta e relatório flexíveis devem ser fornecidas para tratar consultas simples, por exemplo, operadores lógicos binários aplicados apenas a um arquivo lógico interno (conta como um item).
- b) Facilidades de consulta e relatório flexíveis devem ser fornecidas para tratar consultas de complexidade média, por exemplo, operadores lógicos binários aplicados a mais do que um arquivo lógico interno (conta com dois itens).
- c) Facilidades de consulta e relatório flexíveis devem ser fornecidas para tratar consultas de complexidade alta, por exemplo, combinações de operadores lógicos binários em um ou mais arquivos lógicos internos (conta como três itens).
- d) Dados de controle de negócio são mantidos em tabelas gerenciadas pelo usuário e com processos interativos *on-line*, mas as mudanças só têm efeito no dia seguinte (conta como um item).
- e) Dados de controle de negócio são mantidos em tabelas gerenciadas pelo usuário e com processos interativos *on-line*, e as mudanças têm feito imediatamente (conta como dois itens).

Considerando os itens anteriores, a nota deve ser estabelecida assim:

- a) 0: nenhum item.
- b) 1: um item.
- c) 2: dois itens.
- d) 3: três itens.
- e) 4: quatro itens.
- f) 5: cinco itens ou mais.

7.5 Pontos de Caso de Uso

A técnica de *Pontos de Caso de Uso* surgiu em 1993, a partir da Tese de Gustav Karner (1993). O método é baseado em Análise de Pontos de Função, especificamente MK II, que é um modelo relativamente mais simples do que o IFPUG.

A técnica foi incorporada como parte do Processo Unificado, já que este é fortemente baseado em casos de uso. Um dos maiores problemas para a aplicação do método não está no método em si, mas na falta de padronização de entendimento sobre o que é efetivamente um caso de uso (Anda, 2001). Uma explicação detalhada sobre esse assunto é apresentada nos Capítulos 4 e 5 de Wazlawick (2011). Em resumo, um caso de uso deve ser:

- a) um processo de negócio que tenha significado para o usuário;
- b) um processo completo, com início e fim, que deixe a informação do sistema em um estado consistente;

- c) um processo iterativo em que as informações entrem e saiam do sistema para os atores;
- d) um processo que seja executado em uma única sessão de uso do sistema, ou seja, que, uma vez iniciado, não possa ser interrompido. Ou o caso de uso vai até seu final (ou um de seus finais) ou é abortado.

O método se baseia na análise da quantidade e complexidade dos atores e casos de uso, o que gera os UUCP, ou pontos de caso de uso não ajustados. Depois, a aplicação e os fatores técnicos e ambientais levam aos UCP, ou pontos de caso de uso (ajustados).

7.5.1 UAW – COMPLEXIDADE DE ATORES

Os atores são pessoas ou sistemas externos à aplicação em análise com os quais a aplicação se comunica. O método de classificação de complexidade de atores é bem simples. Cada ator é contado uma única vez, mesmo que apareça em vários casos de uso. No caso de atores que são especializações de um ator mais genérico, contam-se apenas as especializações mais elementares, ou seja, atores que não possuem subtipos.

Uma vez identificados os atores, sua complexidade é definida da seguinte forma:

- a) Atores humanos que interagem com o sistema através de interface gráfica são considerados complexos e recebem 3 pontos de caso de uso.
- b) Sistemas que interagem por um protocolo como TCP/IP e atores humanos que interagem com o sistema apenas por linha de comando são considerados de média complexidade e recebem 2 pontos de caso de uso.
- c) Sistemas que são acessados por interfaces de programação (API) são considerados de baixa complexidade e recebem 1 ponto de caso de uso.

O valor de UAW (*Unadjusted Actor Weight*) é, então, simplesmente, a soma dos pontos atribuídos a todos os atores relacionados no sistema.

O uso da medida UAW na estimação de esforço não é, porém, uma unanimidade. Há autores que recomendam que apenas os casos de uso sejam ponderados e que os atores sejam ignorados na contagem. Em geral, inclusive ferramentas de contagem colocam o uso de UAW como uma medida opcional, cujo uso fica a critério do analista.

7.5.2 UUCW – COMPLEXIDADE DOS CASOS DE USO

Os casos de uso também devem ser contados uma única, vez. Apenas processos completos devem ser contados, ou seja, extensões, variantes e casos de uso incluídos não.

Na proposta original de Karner (1993), a complexidade de um caso de uso era definida em função do número estimado de transações (movimentos de informação para dentro ou para fora do sistema), incluindo as sequências alternativas do caso de uso. Desse modo, os casos de uso seriam assim classificados:

- a) Casos de uso simples devem possuir no máximo 3 transações e recebem 5 pontos de caso de uso.
- b) Casos de uso médios devem possuir de 4 a 7 transações e recebem 10 pontos de caso de uso.
- c) Casos de uso complexos devem possuir mais de 7 transações e recebem 15 pontos de caso de uso.

Uma forma alternativa de estimar a complexidade de um caso de uso é em função da quantidade de classes necessária para implementar as funções do caso de uso. Assim:

- a) Casos de uso simples devem ser implementados com 5 classes ou menos.
- b) Casos de uso médios devem ser implementados com 6 a 10 classes.
- c) Casos de uso complexos devem ser implementados com mais de 10 classes.

Outra forma de estimar a complexidade de um caso de uso é pela análise de seu risco. Assim:

- a) Casos de uso como relatórios têm apenas uma ou duas transações e baixo risco, pois não alteram dados e podem ser considerados simples.
- b) Casos de uso padronizados como crud têm um número conhecido e limitado de transações, médio risco (pois, embora a lógica de funcionamento seja conhecida, regras de negócio obscuras podem existir) e podem ser considerados médios;

- c) Casos de uso não padronizados têm um número desconhecido de transações e alto risco, pois, além de as regras de negócio serem desconhecidas, ainda deve-se descobrir qual é o fluxo principal e quais as sequências alternativas. Assim, esse tipo de caso de uso deverá ser considerado complexo.

O valor de UUCW (*Unadjusted Use Case Weight*) é dado pela soma dos valores atribuídos a cada um dos casos de uso da aplicação.

7.5.3 UUCP – PONTOS DE CASO DE USO NÃO AJUSTADOS

O valor de pontos de caso de uso não ajustados, ou UUCP, é calculado simplesmente como:

$$UUCP = UAW + UUCW$$

Como já mencionado, o peso dos atores pode ser excluído do cálculo, o que faria com que UUCP fosse imediatamente igual a UUCW.

7.5.4 TCF – FATORES TÉCNICOS

Pontos de caso de uso fazem o ajuste dos pontos em função de dois critérios: fatores técnicos (que pertencem ao projeto) e fatores ambientais (que pertencem à equipe).

Cada fator recebe uma nota de 0 a 5, em que 0 indica nenhuma influência do projeto, 3 é a influência nominal, e 5 é a máxima influência no projeto.

Os fatores técnicos são 13 e, ao contrário do método de pontos de função, cada um tem um peso específico, conforme mostrado na Tabela 7.43.

Portanto, cada um dos 13 fatores terá uma nota de 0 a 5, multiplicada pelo seu peso. O somatório desses produtos corresponde ao *TFactor*, ou impacto dos fatores técnicos, cujo valor varia entre 0 e 75.

O valor TCF (*Technical Complexity Factor*) pode então ser calculado fazendo-se a normalização do *TFactor* para o intervalo de 0,6 a 1,35:

$$TCF = 0,6 + (0,01 * T\ Factor)$$

TABELA 7.43 Fatores técnicos de ajuste de pontos de caso de uso

| Sigla | Fator | Peso |
|-------|---------------------------------|------|
| T1 | Sistema distribuído | 2 |
| T2 | Performance | 2 |
| T3 | Eficiência de usuário final | 1 |
| T4 | Complexidade de processamento | 1 |
| T5 | Projeto visando código reusável | 1 |
| T6 | Facilidade de instalação | 0,5 |
| T7 | Facilidade de uso | 0,5 |
| T8 | Portabilidade | 2 |
| T9 | Facilidade de mudança | 1 |
| T10 | Concorrência | 1 |
| T11 | Segurança | 1 |
| T12 | Acesso fornecido a terceiros | 1 |
| T13 | Necessidades de treinamento | 1 |

7.5.5 EF – FATORES AMBIENTAIS

Um aspecto que distingue a técnica de pontos de caso de uso de pontos de função e CII é que pontos de caso de uso têm um fator de ajuste específico para as características da equipe de desenvolvimento. Assim, o mesmo projeto, com os mesmos fatores técnicos, poderá ter pontos de caso de uso ajustados distintos para equipes diferentes.

Os fatores ambientais são oito e avaliam o ambiente de trabalho. Cada um tem um peso diferente, e dois deles têm peso negativo. Novamente, as notas devem variar de 0 a 5, mas seu significado é um pouco diferente daquele das notas atribuídas aos fatores técnicos. Enquanto as notas dos fatores técnicos medem a influência desses fatores no projeto, as notas dos fatores ambientais medem a qualidade desses fatores no ambiente de trabalho. Por exemplo, o fator “motivação”, com nota 0, significa que a equipe não está nem um pouco motivada; com nota 3, significa que a motivação da equipe é média ou normal; já com a nota 5 significa que a equipe está altamente motivada.

Assim, os fatores ambientais são definidos conforme a Tabela 7.44.

Assim, o valor das notas, multiplicado pelo peso dos fatores ambientais, pode variar de -10 a 32,5. Esse valor é chamado de *EFactor*.

O *EFactor* é ajustado para o intervalo de 0,425 a 1,7. Assim, os fatores ambientais são calculados da seguinte forma:

$$EF = 1,4 - (0,03 * EFactor)$$

7.5.6 UCP – PONTOS DE CASO DE USO AJUSTADOS

Uma vez calculados os pesos dos atores e casos de uso, os fatores técnicos e ambientais, os pontos de caso de uso ajustados podem ser obtidos pela simples multiplicação desses três valores:

$$UCP = UUCP * TCF * EF$$

7.5.7 ESFORÇO

Uma vez calculado o número de pontos de caso de uso ajustados, isso deve ser transformado em esforço pela aplicação do índice de produtividade da equipe em termos de pontos de caso de uso:

$$E = UCP * IP$$

Assim como no caso de pontos de função, o *IP* pode ser calculado tomando-se projetos já desenvolvidos, somando-se o tempo trabalhado pelo desenvolvedor e dividindo-se o resultado pelo número de pontos de caso de uso estimados. Assim, se o esforço real de um conjunto de projetos foi de 900 desenvolvedores-hora e o UCP total desses projetos era de 60, então o índice de produtividade dessa equipe é de 900/60, ou seja, 15 horas de trabalho por desenvolvedor, por ponto de caso de uso.

TABELA 7.44 Fatores ambientais de ajuste de pontos de caso de uso

| Sigla | Fator | Peso |
|-------|--|------|
| E1 | Familiaridade com o processo de desenvolvimento | 1,5 |
| E2 | Experiência com a aplicação | 0,5 |
| E3 | Experiência com orientação a objetos | 1 |
| E4 | Capacidade do analista líder | 0,5 |
| E5 | Motivação | 1 |
| E6 | Estabilidade de requisitos obtida historicamente | 2 |
| E7 | Equipe em tempo parcial | -1 |
| E8 | Dificuldade com a linguagem de programação | -1 |

A literatura de pontos de caso de uso refere-se normalmente a *horas por UCP*, enquanto outros métodos, como COCOMO, preferem o uso da medida de *desenvolvedor-mês*. Considerando-se que um mês equivale a 158 horas de trabalho, 15 horas por UCP equivalem a cerca de 10,5 (= 158/15) UCP por desenvolvedor-mês.

O trabalho original de Karner (1993) estimava que 20 horas por ponto de caso de uso ajustado seria um bom valor médio. Posteriormente, Ribu (2001) determinou que esse valor poderia variar de 15 a 30.

Outro trabalho (Schneider & Winters, 1998) propõe que os fatores ambientais E1 a E6 com nota menor do que 3 sejam somados à quantidade de fatores ambientais E7 a E8 com nota maior do que 3:

- a) Se o total for 2 ou menos, assumir 20 horas por UCP.
- b) Se o total for 3 ou 4, assumir 28 horas por UCP.
- c) Se o total for maior do que 4, os fatores ambientais apresentam um risco muito grande ao projeto e devem ser melhorados antes que se assuma qualquer compromisso de desenvolvimento, ou então, assumir 36 horas por UCP.

7.5.8 MÉTODOS DE CONTAGEM PARA CASOS DE USO DETALHADOS

A técnica de contagem de pontos de caso de uso apresentada até o momento considera apenas uma *estimativa* de complexidade de casos de uso, porque trabalha com casos de uso de alto nível, os quais são representados unicamente pelo seu nome ou eventualmente por uma ou duas frases explicativas adicionais.

Existem técnicas de contagem de casos de uso detalhados, mas sua utilidade é mais limitada, pois no Processo Unificado os casos de uso só serão detalhados normalmente durante os ciclos iterativos. Na fase de planejamento de projeto, normalmente, os analistas vão contar apenas com casos de uso de alto nível (apenas uma frase ou parágrafo explicativo, sem indicar a sequência de transações, exceções e variantes).

Mesmo assim, essas técnicas podem ser úteis se os casos de uso estiverem detalhados ou se for necessário refinar as estimativas de esforço no início de um ciclo iterativo. Por esse motivo, elas serão resumidas aqui.

Robiolo e Orosco (2008) sugerem que o texto do caso de uso detalhado seja analisado e que sejam contadas as entidades e transações. Dessa forma, o tamanho de uma aplicação em pontos de caso de uso não ajustados seria o somatório desses valores.

Braz e Vergilio (2006) propõem uma métrica chamada FUSP (*Fuzzy Use Case Size Points*), cuja contagem é baseada na complexidade dos atores que participam do caso de uso, na complexidade das pré-condições e pós-condições do caso de uso, no somatório do número de entidades referenciadas e passos do caso de uso, e na complexidade dos fluxos alternativos do caso de uso, sejam variantes, sejam tratadores de exceção. Assim, a medida é bastante criteriosa, e também bastante trabalhosa, e precisa ter os casos de uso totalmente detalhados para poder ser aplicada.

Mohagheghi, Anda e Conradi (2005) propõem uma técnica (*Adapted Use Case Points*) que assume que, inicialmente, todos os atores são de complexidade média e todos os casos de uso são de complexidade alta. Posteriormente, os casos de uso devem ser quebrados em casos de uso menores e reclassificados em simples e médios, se for o caso, à medida que o processo de refinamento vai ocorrendo. Casos de uso estendidos e fluxos alternativos também são contabilizados.

Kamal e Ahmed (2011)¹⁶ apresentam uma extensiva comparação entre esses e outros métodos de contagem de pontos de caso de uso.

7.5.9 FUCS – FULL USE CASE SIZE

O método FUCS, ou *Full Use Case Size* (Ibarra, 2011), foi criado com o objetivo de medir o tamanho de casos de uso para gerar uma medida de complexidade de software. O método mede aspectos funcionais e não funcionais e, para isso, necessita, como entrada, dos casos de uso de alto nível e da lista de requisitos do sistema. A medição de um caso de uso é baseada em três componentes:

- a) O tipo de caso de uso.
- b) O número de operações de sistema e regras de negócio.
- c) O número de requisitos de interface.

¹⁶Disponível em: <www.sdiwc.net/noahjohn/web-admin/upload-pdf/00000078.pdf> . Acesso em: 21 jan. 2013.

O método identifica, primeiramente, dez tipos de casos de uso:

- a) *Operação básica sobre uma entidade*: caso de uso que realiza apenas uma das operações básicas CRUD, ou seja, inserir, alterar, excluir ou consultar um objeto. Esse tipo de caso de uso corresponde à menor unidade funcional possível com significado para o usuário.
- b) *Consulta parametrizada*: caso de uso em que as informações sobre um ou mais objetos são recuperadas a partir de um ou mais parâmetros. O caso de uso normalmente tem dois passos: (1) o ator passa parâmetros e (2) o sistema envia dados. Esse tipo de caso de uso pode apresentar inúmeras variantes, como consultar cliente por data, consultar cliente por produto comprado, consultar cliente por bairro. Com a utilização de técnicas como *objeto-filtro* (Wazlawick, 2011), todas essas variações podem ser implementadas em uma única função.
- c) *Relatório*: semelhante à consulta parametrizada, mas, em vez de retornar apenas os dados armazenados dos objetos, retorna dados derivados (totalizações, por exemplo), organizando esses dados em grupos e seções, por filtros e ordenação.
- d) *CRUD*: corresponde à união das quatro operações básicas sobre uma entidade, e, sempre que possível, deve ser usado no lugar delas.
- e) *CRUD tabular*: corresponde a um CRUD em que um grupo de objetos da mesma classe pode ser editado ao mesmo tempo, ou seja, os dados são apresentados como uma tabela em que, possivelmente, as linhas contêm os objetos e as colunas contêm seus atributos. Normalmente é usado apenas no caso de cadastros mais simples, com poucos atributos e poucos objetos, pois, do contrário, a utilização da tabela deixa de ser prática.
- f) *CRUD mestre/detalhe*: é uma variação do CRUD em que existem pelo menos duas classes com uma relação de agregação ou composição entre elas (podem existir mais níveis de agregação ou composição). No caso do mestre/detalhe com duas classes, existe a classe Mestre, que representa o “todo”, e a classe Detalhe, que representa as partes do todo, como um livro e seus capítulos, ou um CD e suas músicas.
- g) *CRUD detalhes*: é uma variação do CRUD Mestre/Detalhe em que os dados do Mestre podem, eventualmente, ser visualizados (ou nem isso), mas não podem ser alterados. Assim, apenas os atributos da classe Detalhe poderão ser gerenciados pelo usuário.
- h) *Serviço*: são casos de uso que implementam a interação automática com outros sistemas, na maioria das vezes sem a intervenção de um usuário humano. Normalmente, esses casos de uso não necessitam de interfaces gráficas. Um exemplo disso seria um caso de uso de sincronização diária de arquivos de uma aplicação com outro sistema.
- i) *Biblioteca de função ou componente*: esse tipo de caso de uso não é propriamente um processo de usuário com início e fim, mas um fragmento de processo. Por vezes, o modelador de casos de uso se defronta com funcionalidades que são repetidas várias vezes em outros casos de uso, como “Receber pagamento”, mas que sozinhas não são um processo completo (deve-se pagar por algum produto ou serviço). Contudo, por serem reusáveis, esses fragmentos de caso de uso acabam sendo representados no diagrama como extensões ou como fragmentos incluídos em outros casos de uso. Dependendo da situação, uma biblioteca de funcionalidades de caso de uso pode ser criada para ser reusada ao longo do projeto ou até mesmo em outros projetos.
- j) *Não padronizado*: trata-se dos casos de uso complexos não classificáveis pelos critérios anteriores, cuja diversidade é tão grande que não justifica a criação de padrões, já que eles raramente se repetem.

A atribuição de pontos aos tipos de caso de uso é definida em função de três perguntas:

- a) Existe um modelo de especificação disponível para o caso de uso?
- b) Existe algum modelo de interface que pode ser usado para o caso de uso?
- c) Existe algum *framework* de suporte para esse tipo de caso de uso? Em média, qual o percentual de redução de esforço obtido com o uso do *framework*?

O método sugere que todos os casos de uso iniciem com 5 pontos e, em seguida, tenham desconto em função das respostas às três perguntas anteriores, da seguinte forma:

- a) Se a resposta à questão (a) for “sim”, subtraia 1 ponto.
- b) Se a resposta à questão (b) for “sim”, subtraia (mais) 1 ponto.
- c) Se a resposta à questão (c) for “sim” e o percentual de redução for de 20 a 40%, subtraia mais 1 ponto; se for superior a 40%, subtraia mais 2 pontos.

Conforme o tipo de caso de uso e a existência de modelos e tecnologia, então, uma pontuação específica UC_{TP} entre 1 e 5 será atribuída a cada tipo. Ibarra (2011, p. 107) apresenta uma possível avaliação, conforme a Tabela 7.45.

TABELA 7.45 Atribuição de pontos aos casos de uso de acordo com seu tipo

| Tipo | UC_{TP} |
|------------------------------------|-----------|
| Operação básica sobre uma entidade | 1 |
| CRUD | 2 |
| CRUD Tabular | 2 |
| CRUD Mestre/Detalhe | 4 |
| CRUD Detalhe | 3 |
| Consulta parametrizada | 1 |
| Relatório | 1 |
| Serviço | 5 |
| Biblioteca de função ou componente | 5 |
| Não padronizado | 5 |

Essa pontuação, porém, pode variar à medida que novos padrões e *frameworks* forem definidos para os respectivos tipos de caso de uso. Além disso, novos tipos podem ser identificados e sua pontuação pode ser definida de acordo com o método.

A essa pontuação serão acrescidos mais dois componentes:

- a) Número de operações de sistema extras e regras de negócio.
- b) Número de requisitos de interface com usuário.

Operações de sistema devem ser acrescidas ao caso de uso sempre que ele apresentar alguma transação (entrada ou saída de informação) extra em relação ao seu padrão definido. Por exemplo, um caso de uso CRUD que inclui uma operação de geração de senha para o usuário (uma função que não pertence ao padrão CRUD) deve ter seu valor em pontos de caso de uso aumentado, pois sua implementação demandará mais esforço.

Da mesma forma, regras de negócio aumentam a complexidade de um caso de uso. Regras de negócio costumam ser requisitos não funcionais não tecnológicos, ou seja, restrições sobre como as funções do sistema são executadas. Um exemplo seria: “A totalização das comissões deve ser feita sempre no 5º dia útil e referir-se aos pedidos faturados no mês anterior que já tenham sido pagos”.

Assim, para cada caso de uso, a técnica propõe que sejam contadas as operações de sistema extras e as regras de negócio e que seu valor seja somado ao valor do caso de uso. A influência desses fatores, porém, ainda vai depender do grau de esforço α que tais elementos produzem, o que pode ser avaliado da seguinte forma:

- a) $\alpha = 5$ quando não existem ferramentas, padrões e *frameworks* que facilitem a implementação de operações de sistema e regras de negócio.
- b) $\alpha = 3$ quando existe um conjunto de bibliotecas padrão com pontos de extensão que facilitam a implementação desse tipo de requisito.
- c) $\alpha = 1$ quando, além das bibliotecas, existem ferramentas que possibilitam implementar esses requisitos de forma visual ou automatizada.

Além disso, os requisitos de interface gráfica costumam ter um grande peso no desenvolvimento de um sistema, pois é o que o usuário vê e usa. Assim, o método também propõe que esses requisitos sejam contados e somados ao valor do caso de uso. Um exemplo de requisito de interface é a “quando o usuário digita o CEP, o sistema deve preencher automaticamente o endereço”. O peso relativo β dos requisitos de interface será definido a partir das seguintes questões:

- a) $\beta = 5$ quando não existem ferramentas, padrões e *frameworks* que facilitem a implementação de recursos de interface gráfica.
- b) $\beta = 3$ quando existe um conjunto de bibliotecas padrão e componentes de interface gráfica que facilitam a implementação desse tipo de requisito.
- c) $\beta = 1$ quando, além das bibliotecas e componentes de interface gráfica, existem ferramentas que possibilitam implementar esses requisitos de forma visual ou automatizada.

Assim, o tamanho de cada caso de uso é dado pela fórmula:

$$UC_S = UC_{TP} + \alpha * N_{SOBR} + \beta * N_{GUIR}$$

Em que:

- a) UC_S é o tamanho (*Use Case Size*) do caso de uso;
- b) UC_{TP} é o tamanho do caso de uso em função de seu tipo (Tabela 7.45);
- c) α é o peso das operações de sistema e regras de negócio, conforme definido acima;
- d) N_{SOBR} é o número total de operações de sistema extras e regras de negócio do caso de uso;
- e) β é o peso dos requisitos de interface com usuário;
- f) N_{GUIR} é o número total de requisitos de interface com usuário do caso de uso.

Um dos principais diferenciais desse método em relação aos outros é que ele não classifica os casos de uso em categorias de complexidades. Os casos de uso são medidos em UC_S (que varia de 1 a infinito) em função de seu tipo e do número de operações de sistema, regras de negócio e requisitos de interface com o usuário. O tamanho do sistema será simplesmente o somatório do tamanho de cada um de seus casos de uso. A transformação desse valor em esforço depende somente de multiplicar o valor pelo índice de produtividade da equipe.

7.6 Pontos de Histórias

*Pontos de histórias*¹⁷ (PH) é a estimativa de esforço preferida (embora não exclusiva) de métodos ágeis como *Scrum* e *XP*. Um ponto de história não é uma medida de complexidade funcional, como os pontos de função ou pontos de caso de uso, mas uma medida de esforço relativa à equipe de desenvolvimento.

Segundo Kniberg (2007), uma estimativa baseada em pontos de histórias deve ser feita pela equipe. Inicialmente, pergunta-se à equipe quanto tempo X pessoas que se dedicassem unicamente a uma história de usuário levariam para terminá-la, gerando uma versão executável funcional. Se a resposta for, por exemplo, “3 pessoas levariam 4 dias”, então atribua à história $3 \times 4 = 12$ pontos de história.

Assim, um ponto de história pode ser definido como o esforço de desenvolvimento de uma pessoa durante um dia ideal de trabalho, que consiste em uma pessoa dedicada de 6 a 8 horas a um projeto, sem interrupções nem atividades paralelas.

7.6.1 ATRIBUIÇÃO DE PONTOS DE HISTÓRIAS

Nos métodos ágeis, a importância da estimativa normalmente está na comparação entre histórias, ou seja, mais importante do que saber quantos dias uma história efetivamente levaria para ser implementada é saber que uma história levaria duas vezes mais tempo do que outra para ser implementada. Por esse motivo, os pontos de histórias são atribuídos normalmente não como valores da série dos números naturais, mas como valores da série aproximada de números de Fibonacci¹⁸. Um número de Fibonacci é definido como a soma dos dois números de Fibonacci anteriores na série (com exceção dos dois primeiros, que, por definição, são 1 e 1). Assim, o início da série de Fibonacci é constituído pelos números: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 etc. Porém, pode ser estranho mensurar pontos de histórias em 89 ou 34 pontos. Então, na prática, faz-se uma aproximação desses valores para uma série como 1, 2, 3, 5, 8, 15, 25, 40, 60, 100 etc. A ideia é que os pontos de história apresentem uma ordem de grandeza natural para o esforço, e não uma medida exata.

Outra opção para estimar pontos de histórias é usar o sistema “camiseta”, com valores “pequeno”, “médio” e “grande”.

O procedimento de atribuição de pontos funciona assim: tomam-se da lista de histórias de usuário previamente preparada aquelas consideradas mais simples e atribuem-se a elas 1 ou 2 pontos. Depois, sequencialmente, pegam-se outras mais complexas, inicialmente de 3 pontos, depois de 5, e assim por diante. Segundo Toledo (2009)¹⁹, o motivo é que, para o ser humano, é muito mais fácil fazer medidas relativas do que medidas absolutas. Assim, é difícil, por

¹⁷Story Points.

¹⁸Disponível em: <pt.wikipedia.org/wiki/N%C3%A3o_Amero_de_Fibonacci> . Acesso em: 21 jan. 2013.

¹⁹Disponível em: <visaoagil.files.wordpress.com/2009/01/storypoints.pdf> . Acesso em: 21 jan. 2013.

exemplo, uma pessoa estimar o peso de um cavalo sem ter uma balança ou conhecimento prévio do valor. Mas uma pessoa consegue estimar facilmente que um cavalo pesa menos do que um elefante e mais do que um cachorro.

Felix (2009)²⁰ comenta que a atribuição de pontos de histórias usualmente segue critérios subjetivos de complexidade, esforço e risco, sendo caracterizada por frases como:

- a) *Complexidade*: “Essa regra de negócio tem muitos cenários possíveis”.
- b) *Esforço*: “Essa alteração é simples, mas precisa ser realizada em muitas telas”.
- c) *Risco*: “Precisamos utilizar o framework X, mas ninguém na equipe tem experiência”.

Nos métodos ágeis, o esforço precisa ser estimado pela equipe como um todo, e não por um gerente, porque é medida justamente a capacidade das pessoas de realizar as tarefas. Por isso, é necessário que as pessoas que vão fazer isso efetivamente possam opinar e avaliar a carga de trabalho, que, depois de decidida, passará a ser um contrato de desenvolvimento, ou seja, um compromisso.

7.6.2 MEDAÇÃO DE VELOCIDADE

Pontos de histórias são usados por equipes ágeis para medir sua velocidade de projeto. Pode-se fazer um gráfico e deixar à vista de todos onde, a cada ciclo, são medidos os pontos de histórias efetivamente desenvolvidos. Se essa velocidade começar a cair, a equipe deverá verificar o motivo. Vários motivos podem ser listados: desmotivação, erros de estimação, erros de priorização (a equipe começou a tratar histórias de usuário mais simples e deixou as mais complexas e arriscadas para depois) etc.

Em todo caso, quando se deseja aumentar a velocidade da equipe, é comum que se faça a aquisição de mais desenvolvedores para trabalhar nos ciclos restantes. Também é natural que, enquanto os novos desenvolvedores estiverem se ambientando, a velocidade em PH seja reduzida para mais tarde aumentar. A Figura 7.4 mostra um exemplo de gráfico de velocidade de projeto em que, após a aquisição de novos membros para a equipe, há uma redução na velocidade em PH que é compensada após 3 ciclos, quando então os novos membros da equipe passam a ser efetivamente produtivos.

Normalmente, um gráfico de velocidade é relativamente estável. Embora os valores possam variar de um ciclo para outro, sua derivada se mantém constante. Se houver medidas de melhoria de produtividade, pode-se esperar aumentos na velocidade, ou seja, uma derivada positiva. Então, a principal utilidade do gráfico de velocidade consiste em ajudar a diagnosticar possíveis problemas de ambiente, caso a derivada se torne negativa, bem como verificar a efetividade de melhorias no processo produtivo.

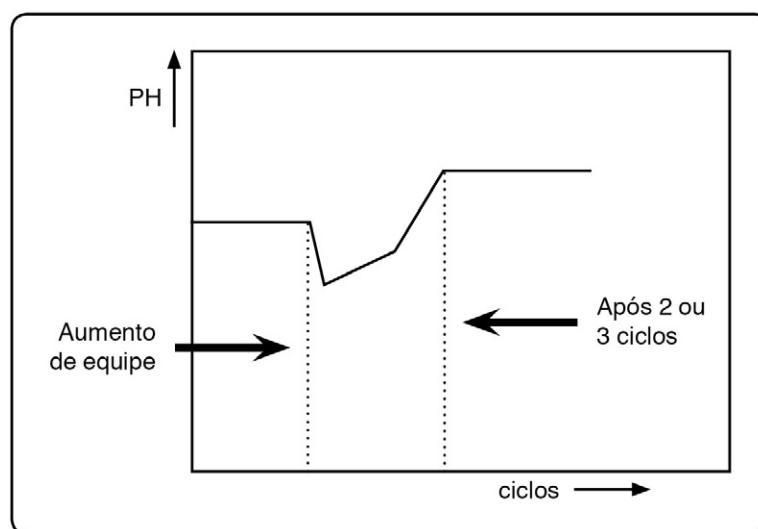


Figura 7.4 Gráfico de velocidade de projeto em pontos de histórias²¹.

²⁰Disponível em: <lucianofelix.wordpress.com/2009/06/10/o-que-significam-story-points/> . Acesso em: 21 jan. 2013.

²¹Toledo (2009).



Capítulo

08

RISCOS

Este capítulo discute um dos principais problemas que causam o insucesso de projetos de software, o *risco*, cujo estudo muitas vezes é subestimado. Inicialmente, o capítulo apresenta o *plano de gerência de riscos* (Seção 8.1), que será construído a partir de uma *identificação de riscos* (Seção 8.2), incluindo um *checklist* (Seção 8.3) para sua efetiva identificação. Os riscos identificados passam por um processo de *análise* (Seção 8.4), que vai determinar o grau de importância ou exposição efetiva de cada risco. Em seguida, são apresentados os *planos de mitigação de riscos* (Seção 8.5), que podem ser executados para reduzir a probabilidade ou o impacto do risco antes que ele se torne um problema, e os *planos de contingência* (Seção 8.6), que devem ser executados quando, apesar de todos os esforços, o risco efetivamente se tornar um problema. O capítulo termina com uma discussão sobre o *monitoramento* (Seção 8.7), o *controle* (Seção 8.8) e a *comunicação* dos riscos (Seção 8.9).

Todo projeto de desenvolvimento de software apresenta um conjunto de incertezas em diferentes graus que podem causar problemas. Um planejador que não esteja atento aos riscos do projeto não terá planos para tratar situações que podem vir a prejudicar ou até a inviabilizar todo um projeto.

Pode-se dizer, genericamente, que a falta de planejamento em relação aos riscos é uma das maiores causas de fracasso em projetos na área de software. Afinal, o que faz o projeto falhar é o cronograma que não foi cumprido, os custos que extrapolaram o orçamento, a qualidade que ficou abaixo do esperado... Tudo isso são riscos que se tornaram problemas.

Um gerente de projeto sem planos de tratamento de riscos não terá parâmetros para ação. É por falta de planejamento relacionado a riscos que muitos projetos de software fracassam ou não conseguem manter seu cronograma e orçamento dentro do previsto. A necessidade de planejar ações referentes aos riscos decorre do fato de que, se isso não for feito, cronogramas e orçamentos dificilmente serão cumpridos.

Na grande maioria dos casos, um planejamento adequado pode fazer que os riscos nunca se tornem problemas ou, caso isso aconteça, seu prejuízo seja minimizado.

O modelo de gerenciamento de riscos do SEI (Carr, Konda, Monarch, Ulrich, & Walker, 1993) envolve seis atividades, conforme listado abaixo:

- a) *Identificação*: antes que os riscos possam ser tratados, eles precisam ser identificados. A Seção 8.2 apresenta algumas técnicas de identificação de riscos.
- b) *Análise*: a análise transforma a lista de riscos potenciais em um documento mais útil para o planejador e o gerente de um projeto, pois, a partir dela, os riscos são priorizados, e o planejador e o gerente podem se concentrar nos riscos mais importantes sem perder tempo com os insignificantes. A Seção 8.4 trata da análise de riscos.

- c) *Planejamento*: o planejamento quanto aos riscos permite ao gerente prevenir problemas, em geral de três formas: 1) planejando e executando planos para reduzir a probabilidade de o risco ocorrer (Seção 8.5.1); 2) planejando e executando planos para reduzir o impacto do risco, caso ocorra (Seção 8.5.2); e 3) planejando as atividades de recuperação de projeto, caso o risco efetivamente ocorra (Seção 8.6).
- d) *Rastreamento*: o rastreamento (ou monitoramento) de riscos consiste em avaliar, ao longo do projeto, as propriedades do risco (por exemplo, a probabilidade de ele ocorrer). O rastreamento deve ser baseado em métricas de avaliação de risco (Seção 8.7).
- e) *Controle*: em função de mudanças no *status* de um risco, alguns planos podem ter que ser executados. Muitas vezes, talvez seja necessário improvisar a resposta ao problema causado pelo risco (Seção 8.8).
- f) *Comunicação*: a comunicação é um processo fundamental ao longo de um projeto de software, especialmente em relação à prevenção e ao tratamento de riscos. Assim, não há uma atividade específica de comunicação em riscos, pois se trata de uma prática que permeia todas as outras atividades (Seção 8.9).

8.1 Plano de Gerência de Riscos

O *Plano de Gerência de Riscos* inclui os vários elementos apresentados neste capítulo. Em resumo, ele deve mostrar:

- a) Quais são os riscos identificados (Seções 8.2 e 8.3).
- b) Uma análise qualitativa ou quantitativa de cada risco que indique, por exemplo, a probabilidade de sua ocorrência e de seu impacto sobre o projeto, caso ocorra (Seção 8.4).
- c) Como a probabilidade de o risco ocorrer pode ser reduzida (Plano de Redução de Probabilidade, Seção 8.5.1).
- d) Como o impacto do risco, caso ocorra, pode ser reduzido (Plano de Redução de Impacto, Seção 8.5.2).
- e) O que fazer se o que era um risco se tornar efetivamente um problema (Plano de Resposta ao Risco, Seção 8.6).
- f) Como monitorar os riscos (Seção 8.7).

Os planos de redução de probabilidade e impacto também são chamados de *planos de mitigação de risco*, ou seja, são planos executados de forma preventiva para evitar que o risco ocorra e, se ainda assim ele ocorrer, seu prejuízo seja reduzido.

O planejamento de riscos conta com alguns fatores inibidores, que, muitas vezes, fazem a equipe não estar preparada para lidar com imprevistos. Entre outras coisas, existe uma cultura de aversão ao risco. Mas, em projetos de software, nem todo o otimismo do mundo vai proteger o andamento das coisas, especialmente se problemas ocorrerem ao longo dele. De qualquer modo, é necessário estar preparado. Ninguém espera que ocorra um incêndio; mesmo assim, os bombeiros estão sempre de prontidão.

Uma característica altamente desejada para um planejador e mesmo para um gerente de projeto em relação ao risco é a capacidade de visão antecipada. Um bom planejador e um bom gerente são capazes de visualizar com antecedência possíveis situações anômalas que poderiam impedir o bom andamento do projeto. Essa previsão, longe de ser uma atitude pessimista, poderá salvar o projeto no futuro ou pelo menos mantê-lo dentro do cronograma e do orçamento previstos. Além disso, nem todos os riscos precisam preocupar o planejador de projeto. Como será visto mais adiante, apenas os riscos de maior importância merecerão mais atenção.

Outro princípio importante é a antecipação das atividades de maior risco, como preconizado nos modelos Espiral, UP e métodos ágeis. A ideia é a seguinte: se um risco pode inviabilizar um projeto, é melhor que isso seja analisado e descoberto o quanto antes, porque quanto mais tempo passar, maior terá sido o investimento e, por conseguinte, o custo.

Os riscos podem ser classificados em três grupos em relação ao conhecimento que se tem deles:

- a) *Conhecidos*: são aqueles já identificados e para os quais a equipe possivelmente estará preparada.
- b) *Desconhecidos*: são aqueles que poderiam ter sido descobertos se as medidas de identificação adequadas tivessem sido tomadas.
- c) *Impossíveis de prever*: são aqueles que não teriam sido identificados nem mesmo com as melhores técnicas de identificação.

Assim, o plano de gerência de riscos vai poder tratar apenas o primeiro grupo de riscos. Para minimizar o segundo grupo, boas atividades de identificação de riscos devem ser desenvolvidas. Já o terceiro grupo vai depender da capacidade do gerente de responder de forma organizada a situações totalmente imprevistas.

8.2 Identificação de Riscos

Normalmente, um risco compõe-se de dois elementos:

- a) Uma *causa*, na forma de uma condição incerta.
- b) Um *problema*, que pode ocorrer em função da causa, provocando um *efeito* ou *impacto* em um ou mais objetivos do projeto ou iteração.

Assim, um risco é um problema que pode resultar de uma causa. Se esse problema ocorrer, estima-se que um dos objetivos do projeto será impactado, seja no tempo, custo, qualidade, seja em outro aspecto. Por exemplo: “Como consequência do uso de um novo hardware (uma exigência definida), erros inesperados de integração do sistema podem ocorrer (um risco incerto), o que levaria ao estouro dos custos do projeto (um efeito sobre o objetivo do orçamento)” (Dinsmore, Cabanis-Brewin, Abdollahyan, Anselmo, Cota & Cavalieri, 2009).

O PMBOK¹ (PMI, 2004), referência em gerenciamento de projetos, define que o risco é uma condição incerta que pode ter tanto um efeito positivo quanto um efeito negativo sobre o projeto. Assim, existiriam também os riscos positivos. Mas, certamente, os riscos mais importantes a serem identificados são aqueles que podem prejudicar o projeto.

Assim como os requisitos de um projeto, os riscos devem ser identificados e priorizados para serem abordados adequadamente.

O ideal é que o planejador tenha a sua disposição um catálogo com riscos que ocorreram no passado em projetos semelhantes. É importante que esse histórico nunca se perca, mas, caso não exista tal registro, uma identificação de riscos pode ser feita em uma reunião com a equipe e discussão sobre possíveis incertezas relacionadas aos tópicos mencionados. Sugestões para essa discussão são apresentadas nas subseções seguintes.

O Processo Unificado associa diferentes tipos de risco com as diferentes fases do projeto:

- a) *Concepção*: riscos de requisitos e de negócio.
- b) *Elaboração*: riscos de tecnologia e arquitetura de sistema.
- c) *Construção*: riscos de programação e teste de sistema.
- d) *Transição*: riscos de utilização do sistema no ambiente final.

A literatura apresenta várias técnicas para identificação de riscos na fase de planejamento de um projeto. Podem-se citar as seguintes:

- a) Uso de *checklists* predefinidos com possíveis riscos: tais listas podem ser obtidas tanto na literatura ou na internet quanto a partir de projetos anteriores executados pela mesma equipe. Uma excelente base para iniciar uma lista desse tipo é o relatório SEI de taxonomia de riscos (Carr, Konda, Monarch, Ulrich & Walker, 1993)², apresentado na Seção 8.3.
- b) Reuniões e *brainstormings* com gerente e equipe de projeto com experiência em outros projetos.
- c) Análise de cenários e lições aprendidas em projetos anteriores com contexto semelhante.

A identificação de riscos deve considerar diferentes fontes, tais como:

- a) Tecnologia (hardware e software).
- b) Pessoas (cliente, equipe, mercado etc.).
- c) Projeto (atrasos, custos exagerados etc.).

¹Disponível em: <www.pmi.org>. Acesso em: 21 jan. 2013.

²Disponível em: <www.sei.cmu.edu/reports/93tr006.pdf>. Acesso em: 21 jan. 2013.

8.2.1 RISCOS TECNOLÓGICOS

Os riscos tecnológicos estão relacionados a todas as incertezas referentes ao modo como a equipe será capaz de lidar com a tecnologia necessária para realizar o projeto. Quanto menos experiência a equipe tiver com essas tecnologias, maiores serão os riscos.

Projetos que envolvam diferentes sistemas de software e hardware enfrentarão problemas de compatibilidade frequentes. Tornar tais sistemas compatíveis demandará tempo e custo extras ao projeto. Assim, o planejador deve saber se diferentes tecnologias terão de interagir e qual é a experiência da equipe com esse tipo de integração.

Outro ponto que pode oferecer risco tecnológico a um projeto é a questão da obsolescência. Quão rápido as tecnologias usadas ou produzidas serão suplantadas por outras mais eficientes?

8.2.2 RISCOS RELACIONADOS A PESSOAS

Há vários tipos de interessado em um projeto. Cada um dos papéis pode produzir um risco característico, como os descritos a seguir.

- a) *Riscos de pessoal*: projetos são executados por pessoas. Então, perder uma pessoa da equipe de forma permanente ou temporária pode ter um grande impacto na medida em que essa pessoa for insubstituível.
- b) *Riscos de cliente*: até que ponto o cliente se manterá interessado no projeto? Mudanças políticas ou administrativas poderão afetar o interesse da empresa em investir no projeto? Mesmo que a empresa mantenha o interesse no projeto, o cliente estará disponível para esclarecer requisitos e realizar testes?
- c) *Riscos de negócio*: muitas vezes, a empresa até constrói um bom produto no prazo e com o custo definidos, mas mesmo assim o projeto fracassa. Entre outras coisas, a empresa pode não ter a habilidade necessária para vender o produto, ou a empresa pode ter essa habilidade, mas o produto não ter efetivamente apelo comercial.
- d) *Riscos legais*: existem problemas ou possibilidade de litígio? Uso de material protegido por direitos autorais? Necessidade de celebração de contrato com terceiros? Normas e leis específicas em outros estados ou países?

8.2.3 RISCOS DE PROJETO

Os riscos de gerenciamento do projeto envolvem a capacidade da equipe de planejar e seguir o plano dentro do cronograma e do custo previstos. Esse tipo de risco costuma se manifestar das seguintes formas:

- a) *Riscos de requisitos*: a equipe, por ser inexperiente, pode não ter sido capaz de identificar corretamente os requisitos do projeto, o que causará problemas ao longo dele. Requisitos poderão ser insuficientes, excessivos ou incorretos. Ou, ainda, os requisitos podem ser naturalmente instáveis, em razão de características do próprio projeto.
- b) *Riscos de processo*: o modelo de processo escolhido é adequado às características do projeto? A equipe tem experiência com o processo? O gerente tem experiência em projetos anteriores?
- c) *Riscos de orçamento*: até que ponto a verba necessária para o projeto está garantida? Os custos foram corretamente previstos em projetos anteriores?
- d) *Riscos de cronograma*: é possível que os prazos sejam alterados e a ordem em que as funcionalidades devem ser entregues possa mudar? O planejador deve saber em que grau a equipe mostrou-se capaz de ater-se ao cronograma em projetos passados.

A inexistência de qualquer uma dessas informações caracteriza um risco importante ao projeto na medida da sua incerteza.

8.3 Checklist de Riscos

O método de identificação de riscos do SEI (Carr, Konda, Monarch, Ulrich & Walker, 1993) é baseado em uma taxonomia que contém termos relacionados ao processo de desenvolvimento de software. Para cada item dessa taxonomia, pode ser elaborado um questionário ou *checklist*, a partir do qual riscos podem ser identificados.

São definidas três grandes classes de risco:

- a) Engenharia do produto.
- b) Ambiente de desenvolvimento.
- c) Restrições externas.

Cada uma dessas categorias possui seus próprios elementos de risco, e cada elemento apresenta suas propriedades, a partir das quais são formuladas perguntas que permitem analisar se o projeto corre ou não algum risco referente a elas.

Inicialmente, apresenta-se o *checklist* referente à *engenharia do produto*, que tem como elementos de risco os requisitos, o *design*, a codificação e o teste de unidade, a integração e o teste, além de outros aspectos específicos de engenharia:

a) Requisitos

- Estabilidade: os requisitos podem mudar durante o desenvolvimento?
 - i. Os requisitos são estáveis?
 - 1. Se não, quais os efeitos disso no sistema? (qualidade/funcionalidade/cronograma/integração/design/teste)
 - ii. As interfaces externas do sistema estão mudando ou vão mudar?
- Completeza: estão faltando requisitos ou eles estão especificados de forma incompleta?
 - i. Existem tópicos a serem esclarecidos nas especificações?
 - ii. Existem requisitos que deveriam estar nas especificações, mas não estão?
 - 1. Se sim, é possível obter esses requisitos e colocá-los na especificação?
 - iii. O cliente tem expectativas ou requisitos que não estão escritos?
 - 1. Se sim, há como de capturá-los?
 - 2. As interfaces externas são completamente definidas?
- Clareza: os requisitos estão obscuros ou necessitam de interpretação?
 - i. Você é capaz de entender os requisitos da forma como eles estão escritos?
 - 1. Se não, há ambiguidades sendo resolvidas satisfatoriamente?
 - 2. Se sim, não há ambiguidades ou problemas de interpretação?
- Validez: os requisitos vão levar ao produto que o cliente tem em mente?
 - i. Existem requisitos que podem não especificar exatamente o que o cliente quer?
 - 1. Se sim, como você está resolvendo isso?
 - ii. Você e o cliente compreendem a mesma coisa a partir dos requisitos?
 - 1. Se sim, existe um processo para determinar isso?
 - iii. Como você valida os requisitos com o cliente?
 - 1. Prototipação?
 - 2. Análise?
 - 3. Simulação?
- Exequibilidade: os requisitos são exequíveis de um ponto de vista analítico?
 - i. Há requisitos que sejam tecnicamente difíceis de se implementar?
 - 1. Se sim, quais são?
 - 2. Se sim, por que eles são difíceis de implementar?
 - 3. Se não, foram feitos estudos de exequibilidade sobre os requisitos?
 - a. Se sim, qual é o seu grau de confiança em tais estudos?
- Precedentes: os requisitos especificam algo que nunca foi feito antes ou que a empresa nunca fez antes?
 - i. Existe algum requisito do estado da arte?
 - 1. Tecnologias?
 - 2. Métodos?
 - 3. Linguagens?
 - 4. Hardware?
 - 5. Se não, algum deles é novo para a equipe?
 - 6. Se sim, a equipe tem conhecimento suficiente nessas áreas?
 - a. Se não, há um plano para adquirir conhecimento nessas áreas?

- Escala: os requisitos especificam um produto maior, mais complexo ou que requer mais organização do que a empresa tem experiência para proporcionar?
 - i. O tamanho e complexidade do sistema são uma preocupação?
 1. Se não, você já fez algo desse tamanho e com tal complexidade antes?
 - ii. O tamanho requer uma organização maior do que a usual para a empresa?

b) Design

- Funcionalidade: existem problemas potenciais para obter os requisitos funcionais?
 - i. Há algum algoritmo especificado que possa não satisfazer os requisitos?
 1. Se não, há algum algoritmo ou *design* que obtenha os requisitos de forma marginal?
 - ii. Como você determina a exequibilidade dos algoritmos e do *design*?
 1. Prototipação?
 2. Modelagem?
 3. Análise?
 4. Simulação?
- Dificuldade: o *design* ou a implementação serão difíceis de realizar?
 - i. Alguma parte do *design* depende de hipóteses otimistas ou não realistas?
 - ii. Existe algum requisito para o qual seja difícil obter um *design*?
 1. Se não, você tem soluções para todos os requisitos?
 2. Se sim, quais são os requisitos e por que eles são difíceis?
- Interfaces: as interfaces internas de hardware e software são bem definidas e controladas?
 - i. Há interfaces internas bem definidas?
 1. Software para software?
 2. Software para hardware?
 - ii. Há um processo para definir interfaces internas?
 1. Se sim, há um processo de controle de mudança para interfaces internas?
 - iii. Há hardware sendo desenvolvido em paralelo com o software?
 1. Se sim, as especificações do hardware estão mudando?
 2. Se sim, todas as interfaces com o software já foram definidas?
 3. Se sim, há modelos de *design* de engenharia que possam ser usados para testar o software?
- Performance: existem tempos de resposta ou taxas de transferência rigorosos?
 - i. Existem problemas com *performance*?
 1. Taxa de transferência?
 2. Escalonamento de eventos de tempo real assíncronos?
 3. Respostas em tempo real?
 4. Tempo para recuperação de falhas (*recovery timeline*)?
 5. Tempo de resposta?
 6. Tempo de resposta, acesso e número simultâneo de usuários da base de dados?
 - ii. Foi feita uma análise de *performance*?
 1. Se sim, qual é o seu nível de confiança na análise feita?
 2. Se sim, você tem um modelo para rastrear a *performance* através do *design* e da implementação?
- Testabilidade: o produto é difícil ou impossível de testar?
 - i. O software vai ser fácil de testar?
 - ii. O *design* inclui características que facilitam o teste?
 - iii. Os testadores se envolveram com a análise dos requisitos?
- Restrições de hardware: existem restrições apertadas no hardware-alvo?
 - i. Arquitetura?
 - ii. Capacidade de memória?
 - iii. Taxa de transferência?
 - iv. Resposta em tempo real?
 - v. Tempo para recuperação de falhas?

- vi. *Performance* da base de dados?
 - vii. Funcionalidade?
 - viii. Confiabilidade?
 - ix. Disponibilidade?
- Software não desenvolvido: há problemas com software usado mas não desenvolvido pela equipe?
 - i. Você está reusando ou fazendo reengenharia em software não desenvolvido pela equipe?
 - 1. Se sim, você prevê algum problema?
 - a. Documentação?
 - b. *Performance*?
 - c. Funcionalidade?
 - d. Prazo de entrega?
 - e. Personalização?
 - ii. Se estiver usando COTS, há algum problema com esse tipo de software?
 - 1. Documentação insuficiente para determinar interfaces, tamanho ou *performance*?
 - 2. *Performance* fraca?
 - 3. Requer uma grande parcela de memória ou da base de dados?
 - 4. É difícil de interfacear com o software?
 - 5. Não foi testado sistematicamente?
 - 6. Não está livre de defeitos?
 - 7. Não foi adequadamente mantido?
 - 8. O vendedor demora para responder?
 - iii. Você prevê algum problema com a integração de atualizações ou revisões de COTS?
 - c) Codificação e teste de unidade
 - Exequibilidade: a implementação do *design* é difícil ou impossível?
 - i. Existem partes do produto não completamente especificadas no *design*?
 - ii. Os algoritmos e o *design* são fáceis de implementar?
 - Teste: os níveis e tempos especificados para os testes de unidade são adequados?
 - i. Você começa o teste de unidade antes de verificar código com respeito ao *design*?
 - ii. Testes de unidade suficientes são especificados?
 - iii. Há tempo suficiente para realizar todo o teste de unidade que você acha necessário?
 - iv. Serão feitos relaxamentos nos testes de unidades se houver problemas de cronograma?
 - Codificação/Implementação: há problemas com codificação e implementação?
 - i. Os *designs* e especificações estão em um nível adequado para permitir a codificação?
 - ii. O *design* muda à medida que o código é escrito?
 - iii. Há restrições de sistema que tornam o código difícil de ser feito?
 - 1. Tempo?
 - 2. Memória?
 - 3. Armazenamento externo?
 - iv. A linguagem de programação é adequada para esse projeto?
 - v. O projeto usa mais de uma linguagem?
 - 1. Se sim, há compatibilidade entre o código produzido pelos diferentes compiladores?
 - vi. O computador de desenvolvimento é o mesmo em que o sistema vai rodar?
 - 1. Se não, há diferenças relativas à compilação entre os dois computadores?
 - vii. Se estiver sendo desenvolvido hardware, suas especificações são suficientes para o desenvolvimento do software?
 - viii. As especificações do hardware mudam à medida que o software é codificado?
 - d) Integração e teste
 - Ambiente: o ambiente de integração e teste é adequado?
 - i. Haverá hardware suficiente para uma adequada integração e teste?
 - ii. Há problemas para desenvolver cenários realísticos e testar dados para demonstrar algum dos requisitos?
 - 1. Tráfego de dados especificado?

- 2. Resposta em tempo real?
 - 3. Tratamento de eventos assíncronos?
 - 4. Interação multiusuário?
 - iii. Você é capaz de verificar a *performance* nas suas instalações?
 - 1. Se sim, isso é suficiente para todos os testes?
 - Produção: a definição de interfaces e instalações são inadequadas ou o tempo insuficiente?
 - i. O hardware-alvo estará disponível quando necessário?
 - ii. Critérios de aceitação foram acordados com o cliente para todos os requisitos?
 - 1. Se sim, há um acordo formal?
 - iii. As interfaces externas foram definidas, documentadas e transformadas em *baselines*?
 - iv. Há algum requisito que seja difícil de testar?
 - v. Foi especificada a integração de um produto suficiente?
 - vi. Foi alocado tempo suficiente para integração e teste do produto?
 - vii. Se usar COTS, os dados do vendedor serão aceitos na verificação dos requisitos alocados a COTS?
 - 1. Se sim, o contrato é claro nesse ponto?
 - Sistema: a integração de sistema é descoordenada, a definição de interface é pobre ou as instalações são inadequadas?
 - i. Integração de sistema suficiente foi especificada?
 - ii. Tempo suficiente para integração de sistema e testes foi previsto?
 - iii. O produto será integrado a um sistema existente?
 - 1. Se sim, haverá um período de transição definido?
 - a. Se não, como se vai garantir que o programa funcionará corretamente depois de integrado?
 - iv. A integração do sistema vai ocorrer nas instalações do cliente?
- e) Aspectos específicos de engenharia
- Capacidade de manutenção: a implementação será difícil de se entender e manter?
 - i. A arquitetura, o *design* ou o código criam dificuldades de manutenção?
 - ii. A equipe de manutenção foi envolvida cedo no processo?
 - iii. A documentação do produto é suficiente para que ele seja mantido por uma organização externa?
 - Confiabilidade: os requisitos de confiabilidade ou disponibilidade são difíceis de se obter?
 - i. Existem requisitos de confiabilidade?
 - ii. Existem requisitos de disponibilidade?
 - 1. Se sim, os prazos de recuperação de falhas são um problema?
 - Riscos à segurança (*safety*): os requisitos relacionados a riscos à segurança são inexequíveis ou não demonstráveis?
 - i. Existem requisitos relacionados a riscos à segurança?
 - 1. Se sim, você vê alguma dificuldade em obtê-los?
 - ii. Será difícil verificar a satisfação desses requisitos?
 - Segurança do sistema (*security*): os requisitos de segurança do sistema são mais rigorosos do que o usual?
 - i. Existem requisitos de segurança no estado da arte ou sem precedentes?
 - ii. O sistema deve seguir regulamentos estritos de segurança estabelecidos por agências governamentais (como, por exemplo, *Orange Book*³)?
 - iii. Você já implementou esse nível de segurança antes?
 - Fatores humanos: o sistema será difícil de se usar por conta de interface com usuário mal definida?
 - i. Você vê alguma dificuldade em satisfazer os requisitos de fatores humanos?
 - 1. Se não, como você garante que vai satisfazer esses requisitos?
 - ii. Se estiver usando prototipação, é um protótipo do tipo *throw-away*?
 - 1. Se não, está fazendo prototipação evolucionária?
 - a. Se sim, você tem experiência nesse tipo de desenvolvimento?
 - b. Se sim, há versões preliminares entregáveis?
 - c. Se sim, isso complica o controle de mudança?

³Disponível em: <www.hm-treasury.gov.uk/d/orange_book.pdf>. Acesso em: 21 jan. 2013.

- Especificações: a documentação é adequada para o *design*, a implementação e o teste do sistema?
 - i. A especificação dos requisitos do software é adequada para o *design* do sistema?
 - ii. A especificação dos requisitos de hardware é adequada para o *design* e a implementação do sistema?
 - iii. As interfaces externas necessárias foram bem especificadas?
 - iv. As especificações de teste são adequadas para testar completamente o sistema?
 - v. Se já estiver na fase de implementação ou após essa fase, as especificações de *design* são adequadas para implementar o sistema?
- 1. Interfaces internas?

Outro conjunto de questionamentos diz respeito ao *ambiente de desenvolvimento*, que tem como elementos de risco o processo de desenvolvimento, o sistema de desenvolvimento, o processo de gerência, os métodos de gerência e o ambiente de trabalho. As questões relacionadas a esses elementos de risco são apresentadas abaixo:

a) Processo de desenvolvimento

- Formalidade: a implementação será difícil de entender e manter?
 - i. Há mais de um modelo de desenvolvimento sendo usado?
 1. Espiral?
 2. Cascata?
 3. Incremental?
 - a. Se sim, a coordenação entre eles é um problema?
 - ii. Há planos formais e controlados para todas as atividades de desenvolvimento?
 1. Análise de requisitos?
 2. *Design*?
 3. Codificação?
 4. Integração e teste?
 5. Instalação?
 6. Garantia de qualidade?
 7. Gerenciamento de configuração?
 - a. Se sim, os planos especificam bem o processo?
 - b. Se sim, os desenvolvedores são familiarizados com os planos?
- Adequação: o processo é adequado para o modelo de desenvolvimento?
 - i. O processo de desenvolvimento é adequado para esse produto? (ver início do Capítulo)
 - ii. O processo de desenvolvimento conta com o suporte de um conjunto de procedimentos, métodos e ferramentas compatíveis?
- Controle de processo: o processo de desenvolvimento é executado, monitorado e controlado com métricas? Os locais de desenvolvimento distribuídos são coordenados?
 - i. Todos seguem o processo de desenvolvimento?
 1. Se sim, como isso é garantido?
 - ii. Você consegue mensurar se o processo de desenvolvimento está atingindo suas metas de qualidade e produtividade?
 - iii. Se há locais de desenvolvimento distribuídos, há coordenação adequada entre eles?
- Familiaridade: os membros do projeto têm experiência no uso do processo? O processo é compreendido por toda a equipe?
 - i. As pessoas estão confortáveis com o processo de desenvolvimento?
- Controle de produto: há mecanismos para controlar a mudança no produto?
 - i. Existe um mecanismo de controle de rastreabilidade de requisitos, que rastreia requisitos desde sua especificação até os casos de teste?
 - ii. O mecanismo de rastreabilidade é usado na avaliação de impacto de mudanças de requisitos?
 - iii. Existe um mecanismo formal de controle de mudança?
 1. Se sim, ele cobre todas as mudanças nas *baselines* de requisitos, *design*, código e documentação?
 - iv. As mudanças em qualquer nível são mapeadas para cima até o nível de sistema e para baixo até o nível de teste?

- v. Há análise adequada quando novos requisitos são adicionados ao sistema?
- vi. Você tem algum meio de rastrear interfaces?
- vii. Os planos e procedimentos de teste são atualizados como parte do processo de mudança?

b) Sistema de desenvolvimento

- Capacidade: existe poder de processamento, estações de trabalho, memória e espaço de armazenamento suficientes?
 - i. Há estações de trabalho e poder de processamento para toda a equipe?
 - ii. Há capacidade suficiente para fases que se entrelaçam, como codificação, integração e teste?
- Adequação: o sistema de desenvolvimento dá suporte a todas as fases, atividades e funções?
 - i. O sistema de desenvolvimento suporta todos os aspectos do projeto?
 1. Análise de requisitos?
 2. Análise de *performance*?
 3. *Design*?
 4. Codificação?
 5. Teste?
 6. Documentação?
 7. Gerenciamento de configuração?
 8. Requisitos e gerenciamento de rastreabilidade?
- Usabilidade: qual é a facilidade de uso do sistema de desenvolvimento?
 - i. As pessoas avaliam que o sistema de desenvolvimento é fácil de usar?
 - ii. Existe boa documentação sobre o sistema de desenvolvimento?
- Familiaridade: existe pouca experiência anterior da empresa ou membros da equipe com o sistema de desenvolvimento?
 - i. As pessoas já usaram essas ferramentas e métodos antes?
- Confiabilidade: o sistema de desenvolvimento sofre de erros, paralisação e capacidade de *backup* nativa insuficiente?
 - i. O sistema de desenvolvimento é considerado confiável?
 1. Compilador?
 2. Ferramentas de desenvolvimento?
 3. Hardware?
- Suporte ao sistema: existe suporte ágil de vendedor ou especialista no sistema de desenvolvimento?
 - i. A equipe foi treinada no uso das ferramentas?
 - ii. Você tem acesso a especialistas nesse uso?
 - iii. Os vendedores respondem rapidamente aos problemas?
- Capacidade de entrega (*deliverability*): os requisitos de definição e aceitação para a entrega do sistema de desenvolvimento para o cliente não foram incluídos no orçamento?
 - i. Você vai entregar o sistema e o desenvolvimento ao cliente?
 1. Se sim, recursos, cronograma e orçamento foram alocados para essa entrega?

c) Processo de gerência

- Planejamento: o planejamento é oportuno, os líderes técnicos foram incluídos e os planos de contingência foram realizados?
 - i. O desenvolvimento é gerenciado de acordo com o plano?
 1. Se sim, as pessoas apagam incêndios com frequência?
 - ii. É feito replanejamento quando ocorrem imprevistos no plano?
 - iii. As pessoas de todos os níveis estão incluídas no planejamento de seu próprio trabalho?
 - iv. Existem planos de contingência para riscos conhecidos?
 1. Se sim, como você determina a hora de ativar esses planos?
 - v. Questões de longo prazo foram adequadamente tratadas?
- Organização de projeto: os papéis e relacionamentos estão claros?
 - i. A organização do desenvolvimento é efetiva?
 - ii. As pessoas compreendem seus próprios papéis e os papéis dos outros?
 - iii. As pessoas sabem quem tem autoridade para o quê?

- Experiência em gerência: os gerentes têm experiência em desenvolvimento de software, gerenciamento de projeto de software no domínio de aplicação, no processo de desenvolvimento ou em grandes projetos?
 - i. O projeto tem gerentes experientes?
 1. Gerenciamento de projeto de software?
 2. Com desenvolvimento de software?
 3. Com o processo de desenvolvimento em uso?
 4. Com o domínio da aplicação?
 5. Com projetos desse tamanho ou dessa complexidade?
- Interfaces da equipe de desenvolvimento: existem interfaces fracas com o cliente, outros contratados ou gerentes de nível superior?
 - i. A gerência comunica problemas para cima e para baixo na linha hierárquica?
 - ii. Os conflitos com o cliente são documentados e resolvidos no tempo apropriado?
 - iii. A gerência envolve as pessoas apropriadas nos encontros com o cliente?
 1. Líderes técnicos?
 2. Desenvolvedores?
 3. Analistas?
 - iv. A gerência trabalha de forma que todas as facções do lado do cliente estejam representadas nas decisões sobre funcionalidade e operação?
 - v. É uma boa política apresentar uma visão otimista ao cliente ou aos gerentes de nível superior?

d) Métodos de gerência

- Monitoramento: as métricas de gerência são definidas e o progresso do desenvolvimento é rastreado?
 - i. Existem relatórios de *status* estruturados periódicos?
 1. Se sim, as pessoas obtêm respostas aos seus relatórios de *status*?
 - ii. A informação apropriada é reportada aos níveis organizacionais corretos?
 - iii. O progresso é comparado com o plano?
 1. Se sim, a gerência tem uma visão clara do que está acontecendo?
- Gerenciamento de pessoal: o pessoal do projeto é treinado e usado adequadamente?
 - i. O pessoal é treinado nas habilidades requeridas pelo projeto?
 1. Se sim, isso é parte do plano do projeto?
 - ii. Há pessoas alocadas ao projeto que não possuem o perfil necessário?
 - iii. É fácil para os membros do projeto conseguir ação da gerência?
 - iv. Os membros em todos os níveis estão conscientes de seu status *versus* plano?
 - v. As pessoas sentem que é importante manter o plano?
 - vi. A gerência consulta as pessoas antes de tomar decisões que afetam o trabalho delas?
- Garantia de qualidade: existem procedimentos e recursos adequados para garantir a qualidade do produto?
 - i. A função de garantia de qualidade tem pessoal adequado para o projeto?
 - ii. Existem mecanismos adequados para garantir a qualidade?
 1. Se sim, todas as áreas e fases possuem procedimentos de qualidade?
 2. Se sim, as pessoas estão acostumadas a trabalhar com esses procedimentos?
- Gerenciamento de configuração: os procedimentos de gerenciamento de configuração, mudança e instalação são adequados?
 - i. Há um sistema gerenciador de configuração adequado?
 - ii. A função de gerenciamento de configuração tem pessoal adequado?
 - iii. É necessário haver coordenação com algum sistema instalado?
 1. Se sim, há gerenciamento de configuração adequado para o sistema instalado?
 2. Se sim, o sistema de gerenciamento de configuração sincroniza seu trabalho com as mudanças no local?
 - iv. O sistema vai ser instalado em múltiplos locais?
 1. Se sim, o sistema de gerenciamento de configuração permite múltiplos locais?

e) Ambiente de trabalho

- Atitude em relação à qualidade: falta orientação relacionada ao trabalho com qualidade?
 - i. Todos os níveis de pessoal estão orientados para procedimentos de qualidade?
 - ii. O cronograma costuma ficar no caminho da qualidade?
- Cooperação: falta espírito de equipe? Os conflitos resultantes requerem intervenção da gerência?
 - i. As pessoas trabalham cooperativamente através das fronteiras funcionais?
 - ii. As pessoas trabalham cooperativamente na direção de metas comuns?
 - iii. A intervenção da gerência às vezes é necessária para fazer as pessoas trabalharem juntas?
- Comunicação: há pouca consciência a respeito de metas ou missão e comunicação pobre de informação técnica entre membros da equipe e gerentes?
 - i. Há boa comunicação entre os membros do projeto?
 1. Gerentes?
 2. Líderes técnicos?
 3. Desenvolvedores?
 4. Testadores?
 5. Gerente de configuração?
 6. Equipe de qualidade?
 - ii. Os gerentes são receptivos às comunicações da equipe?
 1. Se sim, você se sente à vontade para pedir ajuda ao seu gerente?
 2. Os membros da equipe são capazes de comunicar riscos mesmo sem ter uma solução pronta?
- Moral: há uma atmosfera não produtiva ou não criativa? As pessoas sentem que não há reconhecimento ou recompensa por trabalho feito acima das expectativas?
 - i. Como está o moral no projeto?
 1. Se não estiver bem, qual é o principal motivo para o baixo moral?
 - ii. Há problemas para manter as pessoas que são necessárias?

Finalmente, são apresentados questionamentos referentes às *restrições externas* ao projeto, os quais incluem os seguintes elementos: recursos, contrato e interfaces de comunicação externas. Os questionamentos são os seguintes:

a) Recursos

- Cronograma: o cronograma é inadequado ou instável?
 - i. O cronograma tem sido estável?
 - ii. O cronograma é realístico?
 1. Se sim, o método de estimação é baseado em dados históricos?
 2. Se sim, o método de estimação funcionou bem no passado?
 - iii. Existe alguma coisa para a qual um cronograma realístico não foi preparado?
 1. Análises e estudos?
 2. Garantia de qualidade?
 3. Treinamento?
 4. Cursos e treinamento em manutenção?
 5. Equipamentos?
 6. Sistema de desenvolvimento que vai ser entregue?
 - iv. Há dependências externas que poderiam impactar o cronograma?
- Pessoal: a equipe é inexperiente, sem conhecimento de domínio, sem habilidades ou em número insuficiente?
 - i. Há alguma área em que esteja faltando pessoal qualificado?
 1. Engenharia de software e métodos de análise de requisitos?
 2. Especialistas em algoritmos?
 3. *Design* e métodos de *design*?
 4. Linguagens de programação?
 5. Métodos de integração e teste?
 6. Confiabilidade?

7. Capacidade de manutenção?
 8. Disponibilidade?
 9. Fatores humanos?
 10. Gerenciamento de configuração?
 11. Garantia de qualidade?
 12. Ambiente-alvo?
 13. Nível de segurança?
 14. COTS?
 15. Reúso de software?
 16. Sistema operacional?
 17. Banco de dados?
 18. Domínio de aplicação?
 19. Análise de *performance*?
 20. Aplicações de tempo crítico?
 - ii. Há pessoal adequado para formar a equipe de projeto?
 - iii. A equipe é estável?
 - iv. Você tem acesso às pessoas certas quando precisa delas?
 - v. Os membros da equipe já implementaram sistemas desse tipo?
 - vi. O projeto depende de algumas poucas pessoas?
 - vii. Há problemas para conseguir pessoal?
 - Orçamento: o orçamento é insuficiente ou instável?
 - i. O orçamento é estável?
 - ii. O orçamento é baseado em uma estimativa realística?
 1. Se sim, o método de estimativa é baseado em dados históricos?
 2. Se sim, o método funcionou bem no passado?
 - iii. Funcionalidades ou características do sistema já foram removidas para reduzir custo?
 - iv. Há alguma coisa para a qual um orçamento adequado não foi alocado?
 1. Análises e estudos?
 2. Garantia de qualidade?
 3. Treinamento?
 4. Cursos de manutenção?
 5. Equipamento?
 6. Sistema de desenvolvimento a ser entregue?
 - v. Mudanças no orçamento acompanham mudanças nos requisitos?
 1. Se sim, essa é uma parte padrão do processo de controle de mudança?
 - Instalações: as instalações são adequadas para construir e entregar o produto?
 - i. As instalações de desenvolvimento são adequadas?
 - ii. O ambiente de integração é adequado?
- b) Contrato**
- Tipo de contrato: o tipo de contrato é uma fonte de risco para o projeto?
 - i. Que tipo de contrato foi feito? (custo mais prêmio, preço fixo, ...)
 1. Isso apresenta algum problema?
 - ii. O contrato representa uma carga para algum aspecto do projeto?
 1. Declarações de trabalho?
 2. Especificações?
 3. Descrições de itens de dados?
 4. Partes?
 5. Envolvimento excessivo do cliente?
 - iii. A documentação requerida é uma carga?
 1. Quantidade excessiva?
 2. Cliente detalhista e exigente?
 3. Longo ciclo de aprovações?

- Restrições: o contrato causa alguma restrição?
 - i. Há problemas com direitos de dados?
- Dependências: o projeto depende de algum produto ou serviço externo que possam afetar o produto?
 - i. Contratados associados?
 - ii. Contratado principal? (se você for um subcontratado)
 - iii. Subcontratados?
 - iv. Vendedores ou fornecedores?
 - v. Equipamento ou software desenvolvido pelo cliente?
- c) Interfaces de comunicação externas
 - Cliente: há problemas com o cliente, como ciclo longo de aprovação de documentos, comunicação fraca ou conhecimento inadequado do domínio?
 - i. O ciclo de aprovação do cliente é adequadamente rápido?
 1. Documentação?
 2. Revisões de projeto?
 3. Revisões formais?
 - ii. Você sempre segue em frente antes de receber a aprovação do cliente?
 - iii. O cliente entende os aspectos técnicos do sistema?
 - iv. O cliente entende de software?
 - v. O cliente interfere em processos ou na relação com as pessoas?
 - vi. O gerente trabalha com o cliente para alcançar entendimentos e decisões mútuas em tempo aceitável?
 1. Entendimento sobre requisitos?
 2. Critérios de teste?
 3. Ajustes de cronograma?
 4. Interfaces?
 - vii. Quão efetivos são seus mecanismos para chegar a um acordo com o cliente?
 1. Grupos de trabalho (contratuais)?
 2. Encontros de intercâmbio técnico (contratuais)?
 - viii. Todas as facções do cliente são envolvidas para chegar a acordos?
 1. Se sim, há um processo formalmente definido?
 - ix. A gerência apresenta um quadro realista ou otimista ao cliente?
 - Contratados associados (se houver): há problemas com outras empresas que desenvolvem de forma associada para o mesmo cliente, como interfaces fracas ou inadequadamente definidas, comunicação pobre ou falta de colaboração?
 - i. As interfaces externas mudam sem notificação adequada, coordenação ou procedimentos formais de mudança?
 - ii. Há um plano de transição adequado?
 1. Se sim, ele conta com suporte de todos os contratados e pessoal no local?
 - iii. Existem problemas em conseguir cronogramas ou dados de interface de contratados associados?
 1. Se não, eles são acurados?
 - Subcontratados (se houver): o projeto depende de subcontratados em alguma área crítica?
 - i. Há alguma ambiguidade nas definições de tarefas de subcontratados?
 - ii. Os procedimentos de monitoramento e relatório dos subcontratados são diferentes dos usados no projeto?
 - iii. A administração dos subcontratados e seu gerenciamento técnico são feitos por uma organização diferente?
 - iv. Você é altamente dependente de conhecimento especializado de subcontratados em alguma área?
 - v. O conhecimento dos subcontratados está sendo transferido para a empresa?
 - vi. Há problemas para se obter cronogramas ou dados de interface com subcontratados?
 - Contratado principal (se o projeto é um subcontrato): o projeto enfrenta dificuldades com o contratador principal?
 - i. Suas definições de tarefas do contratado principal são ambíguas?
 - ii. Você interfaceia com duas organizações diferentes do contratado principal em relação à administração e ao gerenciamento técnico?

- iii. Você é altamente dependente de conhecimento especializado do contratado principal?
- iv. Há problemas em obter cronograma ou dados de interface do contratado principal?
- Gerenciamento corporativo: falta suporte ou microgerenciamento por parte da gerência superior?
 - i. O gerente de projeto comunica problemas à gerência superior?
 - 1. Se sim, isso parece ser efetivo?
 - ii. A gerência corporativa dá apoio oportuno à resolução de problemas?
 - iii. A gerência corporativa tem tendência a microgerenciar?
 - iv. A gerência apresenta uma visão realista ou otimista para a gerência superior?
- Vendedores: os vendedores atendem rapidamente às necessidades dos projetos?
 - i. Você confia nos vendedores para entregas de componentes críticos?
 - 1. Compiladores?
 - 2. Hardware?
 - 3. COTS?
- Políticas: as políticas causam problemas ao projeto?
 - i. As políticas afetam o projeto?
 - 1. Empresa?
 - 2. Cliente?
 - 3. Contratados associados?
 - 4. Subcontratados?
 - ii. As políticas afetam decisões técnicas?

Conforme comentado, essa lista é apenas uma base ou referência para a criação de um *checklist* específico para uma empresa ou um projeto. A partir dela, podem-se incorporar novos riscos identificados e, assim, construir um patrimônio de forma que os projetos futuros tenham, desde o início, uma base de conhecimento para serem mais bem planejados.

8.4 Análise de Riscos

Uma vez identificados os riscos potenciais, a análise de riscos vai determinar quais são verdadeiramente relevantes para que se gastem tempo e dinheiro com sua prevenção.

Em geral, a análise de riscos vai tentar determinar a probabilidade de ocorrência e o impacto de cada risco potencial.

Algumas propriedades de riscos, entre outras, podem ser assim identificadas:

- a) *Probabilidade*: é a chance de que o risco realmente se torne um problema. Caso se disponha de séries históricas, a probabilidade pode ser medida numericamente. Por exemplo, se 35% dos projetos anteriores sofreram atraso no cronograma, então pode-se esperar que esse risco tenha uma probabilidade de 35% em projetos futuros. Porém, na maioria das vezes, poderá ser mais factível considerar apenas valores discretos como probabilidade alta (quase certo que vá ocorrer), média (tem alguma chance de ocorrer) e baixa (pouco provável que ocorra) para um risco.
- b) *Impacto*: o impacto é a medida do prejuízo que um risco pode trazer ao projeto. O impacto é independente da probabilidade. Pode-se ter riscos de alto impacto com alta ou baixa probabilidade e riscos de baixo impacto com alta ou baixa probabilidade. O impacto pode ser mensurado, por exemplo, quando se sabe o valor da multa por dia de atraso na entrega do produto, mas normalmente será mais prático avaliar o impacto como alto (pode inviabilizar o projeto), médio (pode aumentar significativamente o custo do projeto) ou baixo (apenas provoca algum contratempo, sem inviabilizar os objetivos do projeto).
- c) *Proximidade*: alguns riscos podem ser de alta probabilidade, mas baixa proximidade, ou seja, podem ocorrer só um futuro distante. Por exemplo, a obsolescência tecnológica é um risco de alto impacto e alta probabilidade, mas possivelmente de baixa ou média proximidade, dependendo da tecnologia. Quando a proximidade é considerada propriedade dos riscos, convém também usá-la para o cálculo da importância do risco.
- d) *Acoplagem*: a acoplagem define o quanto um risco pode afetar outros riscos. Por exemplo, se os requisitos estiverem mal estabelecidos, outros riscos de projeto podem ter sua probabilidade ou seu impacto aumentados.

TABELA 8.1 Forma de cálculo para a importância de um risco

| | | Probabilidade | | |
|---------|-------|-------------------|-------------------|-------------------|
| | | Alta | Média | Baixa |
| Impacto | Alto | Alta importância | Alta importância | Média importância |
| | Médio | Alta importância | Média importância | Baixa importância |
| | Baixo | Média importância | Baixa importância | Baixa importância |

O produto da probabilidade pelo impacto consiste na *importância* do risco (ou *exposição*). Assim, riscos de maior importância (alta probabilidade e alto impacto) precisarão ter uma abordagem detalhada no projeto para que sejam tratados. Já os riscos de baixa importância não necessitam de tanto investimento. Eventualmente, será suficiente manter o gerente ciente de sua existência para tomar providências caso a importância do risco se altere.

Deve-se lembrar que, apesar dessa identificação das propriedades de riscos no momento do planejamento, durante a execução do projeto, o gerente deve ter em mente que todas as propriedades podem mudar com o tempo. Assim, o monitoramento dos riscos também é uma atividade que ele deverá realizar com frequência.

Quando a probabilidade e o impacto do risco são avaliados qualitativamente, pode-se usar uma tabela como a Tabela 8.1, em que, a partir da combinação da probabilidade pelo impacto, se chega a um valor para a importância do risco.

Normalmente, os atributos de riscos são definidos em função da experiência e do conhecimento de especialistas. Mas, se houver um registro de projetos anteriores, pode-se ter uma expectativa mais concreta sobre a ocorrência de alguns riscos. Assim, muitas vezes, o planejador de projeto poderá adicionar à sua análise de riscos um conjunto de dados ou observações que justifiquem sua escolha para os valores de determinadas propriedades dos riscos.

8.5 Planos de Mitigação de Riscos

Planos de mitigação de riscos são executados antes que o risco ocorra. Para os riscos de alta importância, os planos de mitigação são definidos ainda na fase de planejamento do projeto. Para riscos de média importância, os planos são definidos e guardados para serem aplicados caso essa importância aumente ao longo do projeto.

Há dois tipos de planos de mitigação: *plano de redução de probabilidade* e *plano de redução de impacto*.

Para exemplificar a apresentação de planos de mitigação, será usado um conjunto de riscos identificados e analisados conforme a Tabela 8.2. Os riscos cujo código é prefixado com “t” são *riscos tecnológicos*. Os riscos prefixados com “pe” são *riscos de pessoal*. Os riscos prefixados com “pr” são *riscos de projeto* e os riscos prefixados com “l” são *riscos legais*.

O projeto trata da produção de uma ferramenta CASE em tecnologia *touch-screen* com comandos de voz e interface em realidade virtual. O projeto seria desenvolvido como projeto de pesquisa por alunos bolsistas. Os riscos foram identificados e avaliados por uma turma de alunos da disciplina de Análise e Projeto de Sistemas II, na UFSC, em 2011. Eles foram avaliados em função de sua probabilidade (P) e impacto (I), e a exposição (E) ou importância, a resultante foi calculada de acordo com a Tabela 8.1, com os valores baixo (B), médio (M) e alto (A). Os riscos aparecem na tabela ordenados de acordo com sua importância.

Os riscos de importância (exposição) alta tiveram planos de mitigação elaborados de forma sumária e executados. Os riscos de importância média deveriam ter seus planos apenas elaborados e guardados caso a importância do risco aumentasse. Os riscos de baixa importância deveriam apenas continuar sendo monitorados e, caso sua importância aumentasse para média, deveriam ter seus planos elaborados (e executados, caso a importância passasse para alta).

TABELA 8.2 Identificação e análise de riscos de um projeto fictício

| Id | Causa | Risco | Efeito | P | I | E |
|-----|---|--|---|---|---|---|
| pr3 | Requisitos ainda muito instáveis. | Pode haver mudanças importantes nos requisitos ao longo do desenvolvimento. | Perda de tempo desenvolvendo partes que depois não serão usadas e atrasos no cronograma. | A | A | A |
| pr2 | O tempo de desenvolvimento pode ser longo. | Pode haver concorrentes que lancem produtos antes. | Chegar ao mercado depois da janela de oportunidade. | A | A | A |
| t8 | Necessidade de muitos comandos baseados em gestos. | Gestos muito parecidos podem significar comandos diferentes. | O sistema pode interpretar erroneamente os comandos (desenhos, formas). Usuário pode ter que decorar muitos comandos diferentes. | A | M | A |
| pe1 | Ainda não se sabe se será possível contratar equipe com experiência nas tecnologias. | Necessidade de treinamento. | Atrasos de cronograma e custos com treinamento. | A | M | A |
| t4 | O processo implementado pela ferramenta pode não atender aos desejos do usuário. | O usuário não vai escolher a ferramenta porque usa um processo de desenvolvimento diferente. | Problemas relacionados à venda. Pode haver necessidade de implementar vários processos, o que vai contra a filosofia inicial da ferramenta. Grandes empresas já têm processo estabelecido e teriam que mudar. | M | A | A |
| t6 | A tecnologia de comando de voz ainda não é bem desenvolvida. | Comandos de voz podem não ser corretamente entendidos. | Usuários frustrados. | A | B | M |
| t9 | Não existem ferramentas CASE com gráficos 3D ou em níveis de profundidade. | Não existe um padrão ou referência para tais interfaces nem estudos de usabilidade. | Necessidade de pesquisar padrões de usabilidade para interfaces 3D em ferramentas CASE. | A | B | M |
| t3 | Não é conhecido um padrão de usabilidade para CASE em <i>touchscreen</i> . | Poderá ser desenvolvida uma ferramenta com usabilidade falha. | Problemas com usuário final (desinteresse). | M | M | M |
| pe1 | O projeto será desenvolvido por bolsistas. | Bolsistas não veem o projeto como carreira. | Podem-se perder desenvolvedores ao longo do projeto, necessitando de substituição. | M | M | M |
| l1 | Uso de tecnologia de terceiros. | Pagamento de direitos autorais. | Aumento de custo. | M | M | M |
| t1 | Superfície de toque é tecnologia nova. | Podem ocorrer mudanças nos padrões. Qual o melhor sistema operacional? | Produto obsoleto ou necessidade de desenvolver para vários sistemas operacionais. | B | M | B |
| t2 | Tecnologia nova. | Podem não existir bibliotecas suficientemente adequadas para o desenvolvimento. | Necessidade de desenvolver novas bibliotecas básicas. | B | B | B |
| t5 | O acesso aos recursos avançados será secundário na interface, que prioriza as ações mais elementares. | Pode gerar problemas de usabilidade para usuários mais avançados. | Gerar desinteresse por usuários avançados. É necessária uma boa análise de caso de uso e usabilidade na ferramenta durante seu desenvolvimento. | B | B | B |
| t7 | O código gerado pela ferramenta, por <i>default</i> , não será modificável. | O código pode não ser o mais eficiente possível. | Sistemas gerados pela ferramenta podem ser ineficientes. | B | B | B |

TABELA 8.3 Planos de redução de probabilidade

| Id | Causa do risco | Risco | Plano de redução de probabilidade |
|-----|--|--|--|
| pr3 | Requisitos ainda muito instáveis. | Pode haver mudanças importantes nos requisitos ao longo do desenvolvimento. | Realizar reuniões de elicitação de requisitos. Iinspecionar requisitos. Procurar produtos semelhantes na Internet e analisá-los. Planejar o desenvolvimento de protótipos. |
| pr2 | O tempo de desenvolvimento pode ser longo. | Pode haver concorrentes que lancem produtos antes. | Planejar desenvolvimento orientado a cronograma com entregas de versões parciais usáveis em intervalos de 6 meses. |
| t8 | Necessidade de muitos comandos baseados em gestos. | Gestos muito parecidos podem significar comandos diferentes. | Pesquisar padrões existentes para comandos baseados em gestos e catalogá-los. Definir hierarquia de comandos e comandos baseados em contextos para reduzir a quantidade de comandos necessários. |
| pe1 | Ainda não se sabe se será possível contratar equipe com experiência nas tecnologias. | Necessidade de treinamento. | Publicar anúncio solicitando currículos para pessoas com as habilidades desejadas. |
| t4 | O processo implementado pela ferramenta pode não atender aos desejos do usuário. | O usuário não vai escolher a ferramenta porque usa um processo de desenvolvimento diferente. | Pesquisar qual o processo mais usado no mercado-alvo. Adaptar a ferramenta para uso com o(s) processo(s) dominantes. |
| t6 | A tecnologia de comando de voz ainda não é bem desenvolvida. | Comandos de voz podem não ser corretamente entendidos. | Pesquisar aplicativos operados por tecnologia de voz e testá-los. Verificar existência de módulos reusáveis de comando por voz. |
| t9 | Não existem ferramentas CASE com gráficos 3D ou em níveis de profundidade. | Não existe um padrão ou referência para tais interfaces, nem estudos de usabilidade. | Catalogar e estudar ferramentas semelhantes com interfaces 3D ou em níveis. |
| t3 | Não é conhecido um padrão de usabilidade para CASE em touchscreen. | Poderá ser desenvolvida uma ferramenta com usabilidade falha. | Catalogar e estudar ferramentas semelhantes já desenvolvidas para superfícies de toque. Estudar normas de usabilidade em geral e normas específicas para ferramentas CASE e para sistemas baseados em superfície de toque. |
| pe1 | O projeto será desenvolvido com bolsistas. | Bolsistas não veem o projeto como carreira. | Verificar o valor de salário de mercado. Verificar a possibilidade de oferecer salários mais atraentes. Verificar a possibilidade de subcontratar desenvolvimento. |
| I1 | Uso de tecnologia de terceiros. | Pagamento de direitos autorais. | Verificar valores e condições de uso de potenciais tecnologias. Verificar existência de soluções livres. |

8.5.1 Plano de Redução de Probabilidade de Risco

O plano de redução de probabilidade de risco consiste nas ações identificadas como necessárias para diminuir a probabilidade de que um risco ocorra. Esse tipo de plano deve agir nas *causas* do risco, ou seja, na segunda coluna da Tabela 8.2. Por exemplo, se é bem provável que a equipe tenha dificuldade com uma nova tecnologia a ser usada, podem-se realizar cursos de treinamento para diminuir a probabilidade de isso se tornar um estorvo ao andamento do projeto.

Riscos classificados como de alta importância devem ter planos de mitigação e contingência com atividades bem definidas, prazos e responsáveis pela sua execução. É importante que tais atividades sejam incluídas no plano do projeto ou iteração e que suas ações sejam contabilizadas no tempo e no custo do projeto.

É interessante lembrar que alguns ciclos de vida, como Espiral, Cascata com Redução de Risco, métodos ágeis e o UP, já preveem que atividades de estudo e mitigação de riscos sejam previstas e incorporadas ao projeto.

No caso de riscos de média importância, pode ser recomendado apenas elaborar um plano de ação genérico para ser aplicado caso seja necessário.

Riscos de baixa importância podem apenas ser monitorados para verificar se a probabilidade ou o impacto aumentam ao longo do projeto e, nesse caso, o plano de redução de probabilidade pode ser criado como resposta a essa mudança de estado.

Considerando o exemplo da Tabela 8.2, os planos de redução de probabilidade poderiam ser elaborados como mostrado na Tabela 8.3.

8.5.2 Plano de Redução de Impacto de Risco

O plano de redução de impacto de risco é definido e aplicado de forma semelhante ao plano de redução de probabilidade, exceto pelo fato de que, nesse caso, as ações devem procurar diminuir o impacto do risco. Assim, tais planos vão procurar diminuir os *efeitos* desse risco, e não suas causas.

Por exemplo, se determinado membro da equipe é o único a dominar a tecnologia a ser usada no projeto, sua saída pode causar grande impacto no andamento do projeto. Assim, um plano de redução de impacto poderia consistir em treinar outro funcionário nessa tecnologia ou colocá-lo para trabalhar em conjunto com o funcionário crítico para que possa aprender o máximo possível sobre a tecnologia. Assim, no caso da falta do outro, o prejuízo ao projeto não será tão grande.

Novamente, os riscos de alta importância determinam a existência de um plano detalhado de redução de impacto com atividades previstas, prazos e responsáveis. Riscos de média importância poderão receber apenas uma lista de ações a serem efetuadas em caso de necessidade, e os riscos de baixa importância serão apenas monitorados.

Considerando o exemplo da Tabela 8.2, os planos de redução de probabilidade poderiam ser elaborados como mostrado na Tabela 8.4.

Os planos de redução de probabilidade e impacto mostrados aqui são apenas sugestões elaboradas a partir de uma rápida reflexão sobre os riscos. Outros planos mais detalhados ou mesmo em direções diferentes poderiam ter sido elaborados.

Por vezes, também pode haver certa ambiguidade entre o plano ser de redução de impacto ou de probabilidade, isto é, pode haver determinadas ações que reduzem tanto a probabilidade quanto o impacto de um risco. Se o planejador de projeto começar a ter problemas em classificar um plano como de um ou outro tipo, será mais interessante que ele crie apenas um conjunto de planos de mitigação de risco, pois sua classificação como redução de probabilidade ou redução de impacto não é tão importante quanto a necessidade de que seja efetivamente planejado e executado, se for o caso.

8.6 Plano de Contingência

O *plano de contingência* ou *plano de resposta ao risco* consiste em um conjunto de ações a serem realizadas caso o risco efetivamente se torne um problema⁴. Esses planos também devem ser incorporados ao plano do projeto, embora de forma opcional, pois sua execução nem sempre será necessária.

A resposta a um risco, depois que ele já se tornou um problema, pode ocorrer de diferentes formas:

- Por eliminação*: procura-se eliminar o problema e o risco, alterando, por exemplo, o escopo do projeto, renegociando contratos, reestruturando a equipe, repensando tecnologias etc.
- Por transferência*: procura-se transferir o problema e o risco a outra parte, por exemplo, subcontratando outra empresa para desenvolver a parte do sistema que apresenta o risco. Isso pode ser feito tanto como ação de resposta ao risco como na forma de ação de mitigação, ou seja, prevenção.
- Por aceitação*: simplesmente aceitam-se as perdas ocasionadas pelo problema e segue-se em frente, se possível. O impacto do risco é que vai determinar a gravidade das consequências, que poderão ir desde um leve inconveniente até o cancelamento do projeto.

Todas essas possibilidades implicam um custo para a empresa desenvolvedora. Assim, sempre será necessário avaliar qual é a estratégia de melhor custo/benefício antes de se implementar qualquer decisão.

⁴A esse respeito, os fãs de Scott Adams poderão apreciar o filme disponível em: <www.youtube.com/watch?v=K-qY_b8lo-k>. Acesso em: 21 jan. 2013.

TABELA 8.4 Planos de redução de impacto

| Id | Risco | Efeito | Plano de redução de impacto |
|-----|--|--|--|
| pr3 | Pode haver mudanças importantes nos requisitos ao longo do desenvolvimento. | Perda de tempo desenvolvendo partes que depois não serão usadas e atrasos no cronograma. | Enfatizar desenvolvimento modular com baixo acoplamento entre módulos. Estabilizar arquitetura-base o quanto antes. Implementar um sistema eficiente de gerenciamento de versões. |
| pr2 | Pode haver concorrentes que lancem produtos antes. | Chegar ao mercado depois da janela de oportunidade. | Manter constante estudo de mercado para garantir que o produto tenha características inovadoras. |
| t8 | Gestos muito parecidos podem significar comandos diferentes. | O sistema pode interpretar erroneamente os comandos (desenhos, formas). Usuário pode ter que decorar muitos comandos diferentes. | Elaborar <i>design</i> de interface alternativo, considerando gestos e alguma forma de eliminar possíveis ambiguidades em gestos. Implementar sistema de ajuda <i>on-line</i> para gestos. |
| pe1 | Necessidade de treinamento. | Atrasos de cronograma e custos com treinamento. | Pesquisar e encomendar bibliografia para treinamento de equipe nas tecnologias necessárias. Prever orçamento para treinamento. |
| t4 | O usuário não vai escolher a ferramenta porque usa um processo de desenvolvimento diferente. | Problemas relacionados à venda. Pode haver necessidade de se implementar vários processos, o que vai contra a filosofia inicial da ferramenta. Grandes empresas já têm processo estabelecido e teriam que mudar. | Verificar se existe possibilidade de definir um metaprocesso adaptável para a ferramenta. |
| t6 | Comandos de voz podem não ser entendidos corretamente. | Usuários frustrados. | Projetar interfaces alternativas a comandos de voz. |
| t9 | Não existe um padrão ou referência para tais interfaces, nem estudos de usabilidade. | Necessidade de pesquisar padrões de usabilidade para interfaces 3D em ferramentas CASE. | Prever a realização de testes de usabilidade para a ferramenta. Prever ciclos de prototipação de interface. |
| t3 | Poderá ser desenvolvida uma ferramenta com usabilidade falha. | Problemas com usuário final (desinteresse). | Idem ao risco t9. |
| pe1 | Bolsistas não veem o projeto como carreira. | Podem-se perder desenvolvedores ao longo do projeto, necessitando de substituição. | Usar programação em pares e padrões de codificação. Planejar integrações frequentes e posse coletiva de código. |
| l1 | Pagamento de direitos autorais. | Aumento de custo. | Prever custos com direitos autorais no orçamento. Verificar existência de tecnologias livres. |

8.7 Monitoramento de Riscos

O processo de gerência de um projeto envolve conscientizar o gerente de que, embora existam planos de mitigação e contingência de riscos, esses riscos são tão volúveis e imprevisíveis quanto os requisitos. Portanto, novos riscos podem surgir ou ser descobertos – ou sua importância pode mudar – com o passar do tempo. Assim, o gerente de projeto deverá estar sempre reavaliando o conjunto de riscos e sua importância.

Para monitorar riscos adequadamente, é necessário documentá-los, seja em um sistema eletrônico de controle de riscos, seja em documentos eletrônicos, ou até mesmo em papel. A primeira opção é quase sempre a mais recomendada, porque o monitoramento do risco implica várias pessoas poderem analisar e modificar o *status* de

um risco ao longo do tempo. Então, um sistema de controle de risco automático, que inclusive gere alarmes para responsáveis pela gerência ou execução de planos, é uma boa escolha.

De qualquer forma, um sistema gerenciador de riscos ou um documento em papel deverá conter algumas informações fundamentais. Wiegert (2007) apresenta um exemplo de documento de monitoramento de risco. A lista a seguir contém esses elementos e também o *status* que não consta da lista original:

- a) *ID*: o identificador do risco. Pode ser um número sequencial ou um mnemônico mais significativo. Esse ID é importante porque a descrição do risco pode mudar com o tempo, e o ID, não. Isso permite que um risco seja monitorado por um longo período, mesmo que mude significativamente.
- b) *Descrição*: a descrição do risco. Possivelmente a melhor forma de descrevê-lo seja através de uma causa e uma consequência (se/então). A descrição do risco normalmente não varia com o tempo, mas nada impede que isso aconteça, especialmente se forem obtidas novas informações sobre ele.
- c) *Probabilidade*: diz respeito à chance de o risco se tornar um problema. Pode ser usada a escala numérica (0 a 100%) ou a escala alta/média/baixa. Eventualmente, a probabilidade poderá ser apresentada como uma função do tempo, por exemplo: “pequena em seis meses, mas grande após esse prazo”. Espera-se que a probabilidade dos riscos varie com o passar do tempo, e essa medida é um dos principais itens que um gerente de projeto deve ter em mente ao monitorar riscos. Inicialmente, espera-se que os planos de mitigação reduzam a probabilidade de o risco se tornar um problema, mas qualquer risco pode ter sua probabilidade aumentada em função de causas imprevistas.
- d) *Impacto*: refere-se à perda ou ao dano ocasionados caso o risco se torne um problema. Aqui também pode ser usada a escala numérica ou alta/média/baixa. Da mesma forma, os planos de mitigação poderão alterar o impacto do risco, caso ele ocorra. O impacto previsto também pode variar com o tempo, à medida que novos conhecimentos sobre o risco vêm à tona, embora talvez não varie tanto quanto a probabilidade.
- e) *Importância*: também chamada de *exposição*, é o produto da probabilidade pelo impacto. Conforme visto na Seção 8.4, os riscos de importância alta (ou numericamente acima de um limite predeterminado) devem ter seus planos de mitigação executados para que a probabilidade ou o impacto sejam baixados, baixando também a importância do risco.
- f) *Primeiro indicador*: é uma descrição da situação inicial a partir da qual se pode concluir que o risco está se tornando um problema. O gerente de projeto deverá estar sempre atento a essa condição, e a equipe também deverá ser alertada para observá-la, reportando-a o quanto antes.
- g) *Planos de mitigação e contingência*: são os planos de redução de probabilidade e impacto, além do plano de resposta ao desastre, que devem ser elaborados, caso o risco seja de importância alta ou média. Esses planos são descritos como uma lista de atividades a serem executadas (um subprojeto) e, no caso dos riscos de importância alta, devem ter responsáveis e prazos atribuídos.
- h) *Responsável*: é a pessoa que deve responder pelos planos de mitigação e contingência, caso o risco seja de alta importância. Planos diferentes poderão ter responsáveis diferentes.
- i) *Prazo*: é a data na qual os planos de mitigação deverão ter sido devidamente executados, caso o risco seja de alta importância. No caso de abordagens iterativas, como UP ou métodos ágeis, o prazo poderá ser descrito como o final de um dos ciclos iterativos.
- j) *Status*: é o estado do risco. Mesmo riscos de baixa probabilidade podem, por azar, se tornar problemas. Então, o *status* de um risco poderá ser atribuído independentemente de uma importância, probabilidade ou impacto. Um risco poderá ser somente um problema *potencial* (por *default*) ou um problema *atual*, caso a condição já tenha acontecido e se esteja sofrendo as consequências, ou ainda um problema *em tratamento*, caso os planos de contingência estejam sendo executados, ou, finalmente, um problema *resolvido*. Depois de resolvido, o risco poderá voltar a ser um problema potencial, caso possa ocorrer novamente (Figura 8.1). Caso o risco não seja resolvido com os planos de contingência, presume-se que ele esteja fora de controle; nesse caso, provavelmente o projeto não vai durar muito tempo, a não ser que um novo tratamento seja iniciado.

Essa lista ou arquivo de riscos deve ser ordenada de forma que os riscos de maior importância estejam no topo (mais visíveis) e os de menor importância estejam no final da lista. Sugere-se que os 10 maiores riscos estejam sempre visíveis para a equipe. Se novos riscos surgirem, o menos importante deve ser tirado da vitrine. Isso é sugerido porque, se a lista de riscos for grande demais, talvez não se preste atenção naqueles que realmente são importantes.

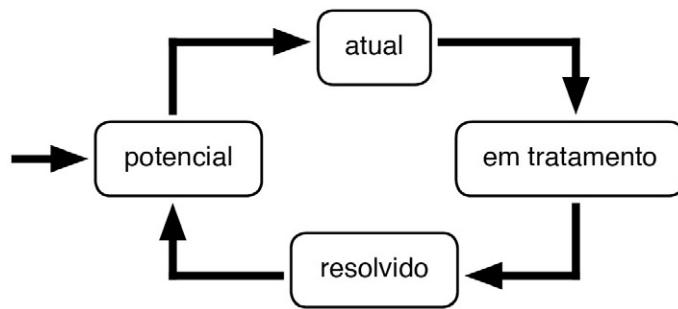


Figura 8.1 Um possível ciclo de vida para o *status* de um risco.

Uma coisa que também não deve ser esquecida é que *novos* riscos podem surgir ao longo do projeto, além daqueles que já estavam na lista.

É interessante colocar a descrição do risco na forma causa/consequência, porque isso mostra mais claramente sua estrutura. Por vezes, os analistas apresentam o risco de forma incompleta. Por exemplo, em vez de apresentar apenas a causa como “não haver acordo sobre os requisitos entre os clientes”, ou apenas a consequência como “vamos poder atender apenas o cliente principal”, devem-se colocar as duas partes formando uma sentença, como “se os clientes não chegarem a um acordo sobre os requisitos, então vamos poder atender apenas ao cliente principal”.

Pode ser vantajoso ter uma pessoa exclusivamente no papel de gerente de riscos de um projeto, aliviando o trabalho do gerente de projeto. Isso é interessante porque talvez seja demais pedir a uma única pessoa que faça um projeto dar certo e, ao mesmo tempo, se preocupe com tudo o que pode (e vai) dar errado. O gerente de risco seria uma pessoa com um pessimismo saudável (sem os exageros de Hardy)⁵, que vai estar sempre analisando as abordagens e situações e avaliando seus riscos.

No caso dos métodos ágeis, as reuniões diárias sugeridas pelos métodos *Scrum* e *XP* podem ser um excelente momento para a equipe avaliar e acompanhar os maiores riscos do projeto.

8.8 Controle de Risco

O *controle de risco* é o processo de observação e gerência que visa acompanhar o estado dos riscos de forma a evitar que se tornem problemas ou que seu prejuízo seja minimizado.

O controle do risco implica a execução prévia dos planos de mitigação de risco, embora alguns autores, como Wiegers (2007, p. 83), definam essas atividades como *resolução* de risco.

Infelizmente, a estimativa de esforço no caso de atividades relacionadas a riscos ainda não é tão previsível quanto o esforço de desenvolvimento, pois essas ações podem variar desde dar um telefonema até realizar um projeto completo com cronograma, recursos e pessoal próprio.

Assim, para estimar esforço para controle de risco, sugere-se aplicar a Lei de Parkinson (Parkinson, 1955) ou Síndrome do Estudante (procrastinação), que diz, entre outras coisas, que uma tarefa se expande até preencher todo o tempo livre. No caso do estudante, trata-se de verificar que dar um prazo de uma semana ou de um mês para a conclusão de uma tarefa será indiferente em relação aos resultados, pois a tarefa acabará sendo feita em período próximo ao final do prazo ou, caso tenha sido feita no início do período, o estudante gastará o restante do tempo para incluir detalhes, muitas vezes irrelevantes, antes que o prazo se esgote.

Desse modo, a sugestão para alocação de prazos para tratamento de riscos é que se utilize uma análise subjetiva, como é feito com pontos de histórias, por exemplo, e que ao final do prazo estabelecido se avalie quais foram os efeitos das atividades de mitigação em relação ao risco. Se o risco tratado teve sua importância baixada de alta para média ou baixa, então as atividades de mitigação cumpriram seu objetivo e, por ora, nada mais precisa ser feito. Caso a importância do risco continue sendo alta, um novo plano deve ser traçado e executado em relação ao risco, ou um novo prazo deve ser estabelecido para a continuação desse mesmo plano.

⁵Hiena pessimista do desenho de Hanna Barbera que está sempre dizendo ao seu parceiro “Lippy, isso não vai dar certo”. Disponível em: <www.hannabarbera.com.br/lippy/lippy.htm>. Acesso em: 21 jan. 2013.

A execução de planos grandiosos pode até fazer que a probabilidade de um risco se tornar problema chegue bem próximo a zero. Porém, nesse caso, o projeto poderá ficar tão caro que talvez não valha mais a pena executá-lo. Assim, o controle do risco deve ser sempre mensurado por uma análise de custo e benefício.

8.9 Comunicação de Riscos

A comunicação na área de gerenciamento de riscos é fundamental, inicialmente nas atividades de identificação. Em geral, a equipe técnica já tem consciência dos riscos que um projeto vai enfrentar, mas, se não forem incentivados a passar essa informação adiante, não o farão. Assim, uma reunião de planejamento inicial com toda a equipe é fundamental para que riscos sejam levantados e avaliados pela equipe como um todo.

Estatisticamente, poucas crises ocorrem de repente. Os problemas costumam ir crescendo lentamente até se transformarem em uma crise, e isso não é diferente em projetos de software. O que vai fazer a diferença entre uma crise que cresce nas sombras até atingir um tamanho que coloque o projeto a perder e um problema que pode ser mitigado em seu início é justamente a capacidade da equipe de perceber e comunicar esse problema rapidamente.

Duarte (2008)⁶ apresenta alguns problemas (riscos) em comunicação que podem levar a crises, alguns dos quais são relevantes para empresas de desenvolvimento de software:

- a) Declaração inadequada.
- b) Opção discutível.
- c) Boatos.
- d) Vazamento de informação.
- e) Política interna.
- f) Falta de integração.
- g) Atendimento frágil.
- h) Questões pendentes.
- i) Falta de recursos.
- j) *Briefing* obscuro.
- k) Dados conflitantes.
- l) Postergamento de decisões.
- m) Funcionários insatisfeitos.
- n) Falta de orientação interna.

A comunicação de riscos não necessariamente evita o problema, mas, como já foi dito, pode ajudar a fazer que medidas antecipatórias ou corretivas sejam administradas o mais cedo possível. Além disso, a comunicação adequada de riscos pode ajudar a empresa a compreender melhor o risco e o problema, de forma a incorporar essa análise ao patrimônio cognitivo da empresa.

Um dos fatores em comunicação de risco que demandam a atenção do gerente é que diferentes interessados têm diferentes percepções sobre o risco. É atribuída a George Bernard Shaw a frase “O maior problema da comunicação é a ilusão de que ela ocorreu”⁷. Assim, não basta apresentar a lista de riscos aos interessados e esperar que todos entendam o que devem fazer. É necessário, em muitos casos, enfatizar as perdas que podem ocorrer caso o risco não seja adequadamente prevenido. Os desenvolvedores estarão mais interessados nos riscos técnicos que podem dificultar seu trabalho, causando frustração; os clientes nos riscos que poderão atrasar o cronograma ou afetar os custos do projeto; os usuários em riscos que envolvem a qualidade do sistema. Assim, comunicar os riscos de maneira correta às partes interessadas é também um processo que deve ser cuidadosamente pensado e executado pelo gerente de projeto.

⁶Disponível em: <www.comunicacaoecrise.com/palestras/gestaobrasil2008jorge.pdf>. Acesso em: 21 jan. 2013.

⁷Disponível em: <www.whale.to/v/shaw1.html>. Acesso em: 21 jan. 2013.

Gerenciamento de Projeto de Software

Nos capítulos anteriores foi visto como planejar um projeto e atenção especial foi dada aos processos de estimação de esforço e controle de riscos. Este capítulo vai tratar do momento em que o projeto efetivamente se inicia. Alguns aspectos da gerência de um projeto já foram discutidos nos capítulos anteriores, mas aqui será aprofundado o processo de gerenciamento como um todo. Inicialmente, será discutido o papel do *gerente de projeto* (Seção 9.1) e suas atribuições, além de recomendações práticas sobre como ele deve atuar. Em seguida, serão apresentados rapidamente os conceitos de gerenciamento de projetos de uma fonte internacionalmente reconhecida, o PMBOK (Seção 9.2). Depois, será apresentado um método de gerenciamento específico, o PRINCE2 (Seção 9.3). Além disso, serão apresentadas recomendações gerais sobre como *conduzir* um projeto de software (Seção 9.4), *métricas* a serem usadas (Seção 9.5), como *revisar e avaliar* (Seção 9.6) o andamento de um projeto e como realizar o *fechamento ou conclusão* de um projeto (Seção 9.7).

A gerência de projeto pode ser entendida como uma disciplina dentro de um processo de engenharia de software (ver Seção 5.3.2.1), em geral exercida por um único indivíduo (o gerente de projeto) cuja função é levar o projeto a alcançar os objetivos planejados dentro dos prazos, com o custo e a qualidade previstos. Para ter sucesso, o gerente deve manter os riscos do projeto nos níveis de probabilidade e impacto mais baixos possível, avaliando continuamente o progresso e adotando medidas proativas para a redução desses riscos ou medidas de correção se, apesar de tudo, houver problemas.

Uma fonte de referência importante sobre a gerência de projetos é o PMBOK (PMI, 2004). Porém, como ele é um referencial genérico para a condução de projetos, é necessário ter em mente que projetos na área de software têm suas particularidades. Assim, um gerente de projeto de software precisa ter conhecimentos específicos sobre *gerência de projetos de software*.

O SWEBOK (IEEE Computer Society, 2004) indica, entre outras coisas, as seguintes particularidades para os projetos de software:

- a) Os clientes dificilmente percebem as reais complexidades envolvidas no processo de desenvolvimento de software, em especial as relacionadas com as mudanças de requisitos (ver mitos na Seção 1.2).
- b) É praticamente inevitável que os próprios processos de engenharia de software acabem gerando a necessidade de introdução de novos requisitos ou de modificação dos requisitos existentes.
- c) Como resultado disso, o software é frequentemente construído por um processo de refinamento interativo em vez de uma sequência de atividades previamente bem definidas e programadas.
- d) A engenharia de software necessariamente incorpora aspectos de criatividade e disciplina. Manter um balanceamento apropriado entre esses dois aspectos costuma ser uma tarefa difícil.

- e) Em geral, o grau de novidade e de complexidade do software é extremamente alto.
- f) A tecnologia subjacente ao software muda com muita frequência.

O SWEBOK divide a área de gerenciamento de projeto em engenharia de software nas seguintes subáreas:

- a) Inicialização e definição de escopo.
- b) Planejamento de projeto de software.
- c) Condução de projeto de software.
- d) Revisão e avaliação.
- e) Fechamento.
- f) Medição em engenharia de software.

As primeiras duas subáreas são tratadas no Capítulo 6. Este capítulo vai abordar as outras subáreas: condução de um projeto de software, sua revisão, medição, avaliação e fechamento.

9.1 O Gerente de Projeto

Segundo Levinson,¹ um projeto bem-sucedido depende muito de um bom gerente. Isso vale tanto em situações nas quais existe a figura do gerente como indivíduo quanto nas situações em que a equipe se autogerencia. De uma forma ou de outra, o processo de planejamento, condução, avaliação e fechamento do projeto estará presente. Levinson indicou seis habilidades-chave observadas em gerentes de sucesso:

- a) *Têm o dom de prever*: bons gerentes têm a capacidade de visualizar e antecipar problemas antes que eles ocorram, e tomar ações preventivas.
- b) *São organizados*: a organização tem vários aspectos, mas um que parece ser especialmente importante para o gerente de projeto é a capacidade de visualizar prioridades e passá-las para sua equipe. Assim, em meio à complexidade de um projeto, o gerente será capaz de enxergar o que é realmente importante e concentrar os esforços da equipe nisso.
- c) *Sabem liderar*: o gerente de projeto não é apenas um chefe. Além da equipe, ele precisa interagir com pessoas que não estão sob seu comando (clientes, usuários e especialistas de domínio, por exemplo). Assim, ele precisa ter o carisma de líder e ser capaz de motivar as pessoas a gastarem seu tempo nas atividades que são necessárias para o projeto.
- d) *São bons comunicadores*: eles são capazes de utilizar múltiplos meios de comunicação, como *e-mail*, telefone, reuniões, apresentações etc., para obter e transmitir as informações necessárias. Eles são efetivos, objetivos e pragmáticos quando se comunicam (não ficam fazendo rodeios). Além disso, são capazes de ouvir. Um gerente que não ouve sua equipe ou outros interessados poderá levar um projeto a fracassar, pois pequenos problemas não resolvidos de início muitas vezes acabam virando grandes problemas no final de um projeto de desenvolvimento de software.
- e) *São pragmáticos*: existem dois extremos quando se pensa na forma de tomada de decisão de um gerente: os que adiam a decisão até conhecer todas as implicações das opções e os que tomam decisões por impulso, sem pensar. No equilíbrio entre esses dois extremos está o gerente pragmático: ele analisa os prós e contras da forma mais eficiente possível, e é capaz de avaliar rapidamente se está em condições de tomar uma decisão fundamentada ou não. Contudo, caso a análise dos prós e contras tome mais tempo do que tentar uma das opções, ele vai perceber isso e decidir se tenta um caminho ou o outro.
- f) *São empáticos*: um gerente não faz o trabalho da equipe nem faz seu trabalho sozinho. Ele precisa se apoiar em outras pessoas. Para obter essa colaboração, ele precisa entender o que motiva as pessoas, ter empatia, que é a capacidade de colocar-se no lugar do outro e entender suas necessidades. A partir disso, o gerente balizará suas ações de motivação.

Classicamente, assumia-se que o gerente de projeto precisava equilibrar os três vértices do *triângulo de restrições*, que envolvia *tempo*, *custo* e *escopo*. Diminuir qualquer um desses ângulos provocaria aumento em pelo menos

¹Disponível em: <cio.uol.com.br/gestao/2008/09/10/caracteristicas-dos-gerentes-de-projeto-bem-sucedidos/> . Acesso em: 21 jan. 2013.

TABELA 9.1 Equivalência entre grupos de processo de gerência no PMBOK e no SWEBOK

| PMBOK | SWEBOK |
|--------------------------|----------------------------------|
| Iniciação | Iniciação Definição de escopo |
| Planejamento | Planejamento |
| Monitoramento e controle | Medição Revisão Avaliação |
| Encerramento | Fechamento |

um dos outros dois. Porém, hoje esse triângulo é substituído por um conjunto de seis variáveis: *escopo, qualidade, cronograma, orçamento, recursos e riscos*. Assume-se ainda que qualquer mudança em um desses elementos afetará pelo menos um dos outros.

9.2 Gerenciamento de Projetos segundo o PMBOK

Apesar de não tratar as particularidades dos projetos de software, o PMBOK (PMI, 2004) é uma excelente referência em termos de gerenciamento de projetos. Ele estrutura o corpo de conhecimentos em duas dimensões: *grupos de processo e áreas de conhecimento*.

Os grupos de processo PMBOK possuem suas equivalentes no SWEBOK, conforme mostrado na Tabela 9.1.

As áreas de conhecimento do PMBOK são descritas a seguir. As áreas de conhecimento não abrangem necessariamente todos os grupos de processo; apenas o gerenciamento de integração faz isso. As nove áreas de gerenciamento são, portanto:

- a) *Gerenciamento de integração*: atividades que o gerente de projetos executa de forma a garantir que todas as partes do projeto funcionem juntas.
- b) *Gerenciamento do escopo*: atividades necessárias para que o projeto execute de fato o que for preciso para gerar o produto e somente isso.
- c) *Gerenciamento de tempo*: atividades mais visíveis em gerência de projeto, consistem em garantir que as atividades do projeto ocorram dentro dos tempos previamente definidos.
- d) *Gerenciamento de custos*: atividades que buscam garantir que o projeto ocorra dentro do orçamento definido.
- e) *Gerenciamento da qualidade*: do ponto de vista externo, visa garantir que o produto atenda às expectativas do cliente; do ponto de vista interno, visa garantir que o produto seja suficientemente maleável para não dificultar desnecessariamente o trabalho da equipe.
- f) *Gerenciamento de recursos humanos*: atividades de aquisição, dispensa, formação e motivação da equipe, bem como de alocação de funções e relações hierárquicas.
- g) *Gerenciamento das comunicações*: controle das comunicações internas e externas ao projeto.
- h) *Gerenciamento de riscos*: uma das áreas mais importantes da gerência de projetos, implica acompanhar o nível de probabilidade e impacto dos riscos e tomar medidas para diminuí-los.
- i) *Gerenciamento de aquisições*: atividades relacionadas à aquisição de produtos ou serviços necessários ao projeto que não sejam produzidos ou fornecidos pela equipe de desenvolvimento.

Em vez de apresentar as combinações no formato de tabela² ou mesmo por área de conhecimento, as listas a seguir apresentam o que se espera do gerente de projeto nas várias áreas em cada um dos cinco momentos de um projeto.

²Disponível em: <pt.wikipedia.org/wiki/Project_Management_Body_of_Knowledge> . Acesso em: 21 jan. 2013.

No momento de *iniciação* de um projeto, apenas as áreas de integração e comunicação terão atividades relacionadas:

- a) *Atividade de integração*: desenvolver o termo de abertura do projeto.
- b) *Atividade de comunicação*: identificar as partes interessadas.

No momento de *planejamento* (Capítulo 6), as nove áreas terão atividades relacionadas:

- a) *Atividade de gerenciamento de integração*: desenvolver o plano de gerenciamento do projeto (Seção 6.1).
- b) *Atividades de gerenciamento de escopo*: coletar os requisitos do projeto (não do software), definir o escopo, criar a estrutura analítica do projeto (WBS),³ conforme explicado na Seção 6.5.1. No planejamento por ciclos iterativos, a criação da WBS só ocorre no planejamento das iterações, não do projeto.
- c) *Atividades de gerenciamento de tempo*: definir as atividades, sequenciá-las, estimar seus recursos e sua duração, e desenvolver o cronograma (Seção 6.5). No planejamento por ciclos iterativos, essas atividades também são deixadas para o planejamento dos ciclos, embora existam atividades de planejamento de tempo para o projeto como um todo (número e duração das iterações, por exemplo).
- d) *Atividades de gerenciamento de custos*: estimar os custos e determinar o orçamento.
- e) *Atividades de gerenciamento de qualidade*: planejar a qualidade.
- f) *Atividades de gerenciamento de recursos humanos*: desenvolver o plano de recursos humanos.
- g) *Atividades de gerenciamento de comunicação*: planejar as comunicações.
- h) *Atividades de gerenciamento de riscos*: planejar o gerenciamento de riscos, identificá-los, realizar sua análise qualitativa e sua análise quantitativa, e planejar as respostas aos riscos (Capítulo 8).
- i) *Atividades de gerenciamento de aquisição*: planejar as aquisições.

No momento da *execução* de um projeto, cinco das nove áreas terão atividades relacionadas:

- a) *Atividades de gerenciamento de integração*: orientar e gerenciar a execução do projeto.
- b) *Atividades de gerenciamento de qualidade*: realizar a garantia de qualidade.
- c) *Atividades de gerenciamento de recursos humanos*: mobilizar, desenvolver e gerenciar a equipe do projeto.
- d) *Atividades de gerenciamento de comunicação*: distribuir as informações, gerenciar as expectativas das partes interessadas.
- e) *Atividades de gerenciamento de aquisição*: conduzir as aquisições.

O momento de *monitoramento e controle* não é exatamente distinto do momento de execução – os dois ocorrem simultaneamente. Em relação ao monitoramento e ao controle, as seguintes áreas terão atividades relacionadas:

- a) *Atividades de gerenciamento de integração*: monitorar e controlar o trabalho do projeto e realizar o controle integrado de mudanças.
- b) *Atividades de gerenciamento de escopo*: verificar e controlar o escopo.
- c) *Atividades de gerenciamento de tempo*: controlar o cronograma.
- d) *Atividades de gerenciamento de custos*: controlar os custos.
- e) *Atividades de gerenciamento de qualidade*: realizar o controle de qualidade.
- f) *Atividades de gerenciamento de riscos*: monitorar e controlar os riscos.
- g) *Atividades de gerenciamento de aquisição*: administrar as aquisições.

O momento do *encerramento* de um projeto necessita de atividades gerenciais nas seguintes áreas:

- a) Atividades de gerenciamento de integração: encerrar o projeto ou a fase.
- b) Atividades de gerenciamento de comunicação: reportar o desempenho.
- c) Atividades de gerenciamento de aquisição: encerrar as aquisições.

Essa organização das atividades mostra como é complexo e multidimensional o trabalho do planejador e do gerente de software. Convém que, mesmo que as atividades não sejam efetivamente detalhadas e executadas em momentos distintos, o gerente (especialmente o iniciante) revise suas realizações a cada dia para verificar se todas as dimensões de sua responsabilidade foram realizadas.

³WBS – Work Breakdown Structure

9.3 Prince2 – Projects IN Controlled Environments 2

*Projects IN Controlled Environments 2*⁴, ou simplesmente *Prince2*, é um método estruturado de gerência de projetos aceito como padrão de gerenciamento de projetos pelo governo do Reino Unido⁵. *Prince2* foi lançado em 1996, como sucessor de outros métodos mais antigos. Desde 2006, o método tem sido revisado e atualizado, além de estar se tornando um pouco mais leve. A versão atual é conhecida como “*Prince2:2009*” (Turley, 2010)⁶.

Prince2 é um método prescritivo que não se aplica bem ao uso com modelos ágeis. Ele pode ser bem aplicado com modelos de processos mais prescritivos e é baseado em sete princípios⁷:

- a) *Justificação continuada de negócio*: isso significa que existe uma justificativa para iniciar o projeto e que ela continua válida ao longo dele, tendo sido documentada e aprovada. Em *Prince2*, a justificativa é documentada no caso de negócios, que dirige o processo de tomada de decisão e garante que o projeto permaneça alinhado com os objetivos e benefícios de negócios.
- b) *Aprender com a experiência*: espera-se que as equipes e o projeto aprendam com a experiência. Lições aprendidas são identificadas, registradas e praticadas ao longo do ciclo de vida do projeto. Em *Prince2*, as lições aprendidas são registradas no *lessons log*, ou diário de lições, e se tornam parte do relatório de lições aprendidas preparado pelo gerente de projeto ao final de cada estágio e ao final do projeto.
- c) *Papéis e responsabilidades definidos*: todos os projetos têm papéis e responsabilidades definidos e acordados, engajando todos os aspectos das organizações envolvidas interna e externamente. Os papéis devem ser definidos de forma que cada participante saiba exatamente o que se espera dele.
- d) *Gerenciar por estágios*: o gerenciamento de projetos em *Prince2* é compatível com os modelos baseados em ciclos iterativos. Um ciclo é chamado de “estágio” em *Prince2*. Ao final de cada estágio, o projeto é revisado para verificar se atingiu seus objetivos e se vai produzir a entrega prevista no caso de negócios. Isso é feito pelo uso de dois níveis de planejamento: longo prazo e curto prazo, sendo o segundo bem mais detalhado do que o primeiro.
- e) *Gerenciar por exceção*: os projetos *Prince2* definem limites de tolerância para tempo, custo, qualidade, escopo e risco. Esses limites são usados para definir os níveis de autoridade delegada. Isso permite que a gerência seja feita dentro de um processo de gerência por exceção, ou seja, se esses limites de tolerância forem excedidos ou se for previsto que eles serão excedidos, então um nível superior de gerência deve ser acionado para decidir como proceder.
- f) *Foco nos produtos*: *Prince2* foca na definição e entrega de produtos que satisfazem os critérios estabelecidos de qualidade. Isso inclui o produto final de um projeto, bem como outros subprodutos significativos gerados ao longo do ciclo de vida do projeto. Essa abordagem orientada ao produto resulta na definição de produtos sobre os quais se obtém concordância e depois no planejamento das atividades para obtê-los. Isso é obtido pelo uso de técnicas de planejamento baseadas em produto.
- g) *Personalização para se ajustar ao ambiente de trabalho*: *Prince2* deve ser personalizado para se adequar ao ambiente de trabalho, ou seja, tamanho, complexidade, importância, capacidades e riscos do projeto. Porém, quando se está personalizando o método, é importante não omitir nenhuma parte, pois todas elas são interligadas.

Além disso, o método também possui sete temas⁸, os quais descrevem aspectos críticos do gerenciamento de sistemas e devem ser considerados pelo gerente ao longo do ciclo de vida do projeto:

- a) *Caso de negócio*: o caso de negócios dirige toda a tomada de decisões ao longo do projeto. Ele é criado no início do projeto e deve justificar os investimentos inicial e continuado. Permite que o gerente de equipe defina, a qualquer momento, se o projeto é viável, desejável e factível. O caso de negócios é mantido atualizado ao longo do projeto com relação a riscos, benefícios e custos.
- b) *Organização*: estabelece os papéis e as responsabilidades ligados ao projeto. Os projetos *Prince2* são organizados em quatro níveis de decisão: *corporação* ou *gerenciamento de programa*, responsável pela encomenda

⁴Projetos em ambientes controlados.

⁵Disponível em: <www.prince-officialsite.com/>. Acesso em: 21 jan. 2013.

⁶Disponível em: <upload.wikimedia.org/wikipedia/commons/8/87/The_PRINCE2_Process_Model_Book.pdf>. Acesso em: 21 jan. 2013.

⁷Disponível em: <project-management.learningtree.com/2010/04/03/the-7-prince2-principles-a-closer-look/>. Acesso em: 21 jan. 2013.

⁸Disponível em: <project-management.learningtree.com/2010/04/11/successfully-juggling-the-7-prince2-themes/>. Acesso em: 21 jan. 2013.

do projeto; *diretoria (project board)*, que fornece direção e gerenciamento geral (*governance*); *gerência de projeto*, que lida com as questões do dia a dia do projeto; e *gerência de equipe*, que se ocupa com a entrega dos produtos do projeto.

- c) *Qualidade*: procura garantir que o produto atenda ao seu propósito. A abordagem definida na estratégia de gerenciamento de qualidade requer que exista um entendimento explícito sobre o escopo do projeto, bem como critérios de qualidade contra os quais o produto será avaliado. Em outras palavras, o foco da qualidade está em fazer o produto atender aos requisitos (especialmente aos de qualidade). O método possui duas formas de aplicação de controle de qualidade: *in process*, que implica construir os produtos com qualidade ao longo do projeto, e *appraisal*, usado para verificar se os produtos acabados satisfazem os critérios de qualidade.
- d) *Planos*: são usados para definir *como, quando e por quem* os produtos serão gerados. Planos são usados para permitir e facilitar a comunicação efetiva e o controle. Eles devem estar alinhados com o caso de negócios e precisam ser aprovados por todos os níveis de gerenciamento. Existem três níveis de planos do Prince2: *plano de projeto*, *plano de estágio* e, opcionalmente, *plano de equipe*. Adicionalmente, planos de exceção poderão substituir um dos outros planos em situações em que os níveis de tolerância de gerenciamento forem excedidos. Prince2 requer que o planejamento seja orientado ao produto, estabelecendo que primeiramente o produto do trabalho deve ser decidido e somente depois as atividades, dependências e recursos podem ser determinados, sempre com o objetivo de produzir o produto especificado.
- e) *Riscos*: para Prince2, os riscos são incertezas que podem ser tanto positivas (oportunidades) quanto negativas (ameaças). O gerenciamento de risco trata da identificação proativa, da avaliação e do controle dos riscos do projeto de forma a maximizar as chances de sucesso. A estratégia de gerenciamento de riscos, definida durante a iniciação do projeto, possui cinco passos: *identificar a causa, evento e efeitos do risco; analisar sua probabilidade e impacto; planejar as respostas necessárias; implementar as respostas como requerido; e comunicar os riscos interna e externamente*.
- f) *Mudança*: envolve a gerência de configuração, problemas e solicitações de mudança. Mais detalhes sobre essas disciplinas serão vistos no Capítulo 10 e na Seção 9.4.2.
- g) *Progresso*: refere-se aos mecanismos usados para monitorar e comparar o atual estado do projeto em relação aos planos. Esse mecanismo também permite que o gerente de projeto faça previsões de *performance* baseadas nas tendências atuais e, assim, realize ações proativas visando a alternativas de projeto quando necessário.

Os sete princípios e sete temas aparecem juntos em sete processos⁹:

- a) *Dar partida em um projeto*: as atividades incluem indicar um executivo e um gerente de projeto, planejar e indicar uma equipe de gerenciamento de projeto, preparar um resumo executivo do projeto, definir a abordagem do projeto e planejar para o próximo estágio.
- b) *Iniciar um projeto*: as atividades incluem planejar qualidade, planejar o projeto, refinhar o caso de negócios e riscos, definir os meios de controle do projeto, definir os arquivos do projeto e montar o documento de início de projeto.
- c) *Dirigir um projeto*: as atividades incluem autorizar o início do projeto, autorizar um estágio ou plano de exceção, dar orientações *ad hoc* e confirmar o fechamento do projeto.
- d) *Controlar um estágio*: as atividades incluem autorizar um pacote de trabalho, avaliar o progresso, capturar e examinar assuntos do projeto, revisar o *status* do estágio, relatar destaques, tomar ação corretiva e receber um pacote de trabalho completo.
- e) *Gerenciar limites de estágios*: as atividades incluem planejar um estágio, atualizar um plano de projeto, um caso de negócio e a lista de riscos, relatar o final de um estágio e produzir um plano de exceção.
- f) *Gerenciar entrega de produto*: as atividades incluem aceitar um pacote de trabalho, executá-lo e entregá-lo.
- g) *Fechar um projeto*: as atividades incluem encerrar (*decommissioning*) o projeto, identificar ações posteriores e revisar a avaliação do projeto.

O método Prince2 estipula três níveis de certificação: *foundation*, para os que aprenderam os fundamentos do método; *practitioner*, para aqueles que vão gerenciar projetos usando o método; e *certification*, para aqueles que buscam ainda mais competência em gerenciamento de projetos¹⁰.

⁹Disponível em: <www.prince2.com/prince2-process-model.asp>. Acesso em: 21 jan. 2013.

¹⁰Disponível em: <www.prince2.com/what-is-prince2.asp>. Acesso em: 21 jan. 2013.

9.4 Condução de Projeto de Software

Muitas vezes, após o planejamento, um projeto de software precisa ser executado durante um longo período de tempo. Assim, é papel do gerente garantir que as variáveis de tempo, recursos, qualidade e escopo sejam mantidas nos valores esperados.

Após o planejamento, espera-se que riscos já tenham sido avaliados e existam planos para mitigá-los, responsáveis nomeados, recursos alocados, e que um processo de desenvolvimento já esteja perfeitamente definido. Resta executar o projeto – e pode ser que nessa hora tudo comece a dar errado.

Um projeto, mesmo bem planejado, pode falhar por vários motivos: erros da equipe, erros no próprio projeto, erros na concepção do processo ou, ainda, fatores imprevistos. Isso decorre principalmente do fato de que os executores do projeto são pessoas, e não máquinas, e de que um projeto não é um programa de computador que roda de forma previsível. Muitos fatores de incerteza estão envolvidos mesmo nos projetos mais bem planejados e gerenciados.

Staa (2003)¹¹ indica que um dos maiores desafios que o gerente de projeto enfrenta no momento do acompanhamento da execução de um projeto é a *indisciplina*. Membros da equipe podem não seguir os padrões, os prazos ou prioridades estabelecidos. O folclore da Ciência da Computação apresenta os “gênios” como superdesenvolvedores que não seguem regras, não têm horários preestabelecidos, são desorganizados e, muitas vezes, têm problemas de higiene pessoal. Se não houver disciplina no ambiente de trabalho, esses “heróis” poderão ressurgir e tomar o projeto como refém.

Não se devem confundir os métodos ágeis, mesmo os mais radicais, como o XP, com indisciplina e caos. Para que um modelo como o XP funcione na prática, a equipe deve ser muito bem disciplinada. No caso de *Scrum*, a equipe deve ser necessariamente autodisciplinada, pois se autogerencia.

Assim, o uso de métodos ágeis não é desculpa para fazer as coisas de qualquer maneira. Mesmo que os desenvolvedores trabalhem com criatividade, buscando soluções inovadoras e às vezes ousadas para os problemas, devem entender que existem as prioridades de projeto, que ou são discutidas e mudadas, ou são mantidas e seguidas.

Além disso, existem padrões a serem seguidos. O desenvolvedor baixa uma versão do componente no sistema de controle de versões e tem total liberdade para trabalhar em sua cópia. Mas, no momento de salvar uma nova versão no sistema, o que ele colocar ali vai afetar o restante da equipe. Então, essa versão precisa estar de acordo com os padrões estabelecidos, testada e estabilizada.

É responsabilidade do gerente de projeto, portanto, identificar se há algum tipo de desvio nocivo e motivar os desenvolvedores, conscientizando-os da necessidade de trabalhar em harmonia (lembrando que *harmonia* não significa cada um fazer o que quer, mas todos seguirem as regras que o grupo estabeleceu).

Staa (2003)²³ identifica três perfis de gerência em relação à equipe:

- a) *Ditador*: é o mais danoso dos três. Faz todo o planejamento e as estimativas, e determina sozinho quem faz o quê e quando. Gera belíssimos diagramas Gantt que ninguém leva a sério e usa de força e ameaça para fazer projetos que não estão indo bem voltarem para os trilhos, o que nem sempre consegue.
- b) *Coordenador*: ouve os outros e faz as previsões e o planejamento em conjunto com a equipe. Faz revisões do planejamento periodicamente. Seus diagramas Gantt são levados mais a sério, pois representam um compromisso realista da equipe, e não apenas uma determinação do gerente. Seus projetos têm maior chance de terminar no prazo. Esse tipo de gestão costuma funcionar bem com métodos prescritivos.
- c) *Facilitador*: apenas agiliza e facilita o trabalho da equipe, mas não toma as decisões. A própria equipe define os prazos e o planejamento. Esse tipo de gerência funciona bem com os métodos ágeis.

Staa (2003)²³ apresenta os três objetivos de acompanhamento de projetos, segundo CMMI, e acrescenta um quarto:

- a) Acompanhar os resultados e desempenhos reais, confrontando-os com o plano de desenvolvimento de software.
- b) Realizar ações corretivas e gerenciá-las até sua conclusão, sempre que resultados ou desempenhos reais se desviarem significativamente do que foi estabelecido (estimado) no plano de desenvolvimento de software.
- c) Assegurar que as alterações nos compromissos de software se deem através de acordo entre as pessoas e os grupos envolvidos.
- d) Acompanhar processos e metaprocessos, obtendo indicadores quanto à sua eficácia em instanciar planos e processos.

¹¹Disponível em: <ftp://ftp.inf.puc-rio.br/pub/docs/techreports/03_13_staa.pdf> . Acesso em: 21 jan. 2013.

Os dois primeiros objetivos mencionados estão relacionados com o dia a dia do gerente de projeto e correspondem às atividades de verificação e controle por ações corretivas. O terceiro objetivo está relacionado às mudanças eventualmente necessárias no plano do projeto. O quarto relaciona-se com a necessidade de mudanças no processo quando se detecta que ele não é suficientemente adequado.

Finalmente, convém mencionar que um plano mal feito será difícil de se gerenciar. Assim, é fundamental que no momento do planejamento de um projeto as técnicas apresentadas no Capítulo 6 sejam observadas. Por exemplo, se o plano não prevê artefatos prontos em determinadas datas, fica difícil o gerente avaliar se as atividades previstas foram efetivamente realizadas. Como a verificação do gerente tende a ser pontual (ele avalia o *status* do projeto ou o de cada desenvolvedor em determinados momentos), ele só será capaz de avaliar o estado e a qualidade de artefatos produzidos, mas não diretamente o estado e a qualidade das atividades em si.

Staa (2003)²³ apresenta três formas de acompanhamento de projetos de software: *folha de tempo, acompanhamento de problemas e registro de artefatos*, as quais são resumidas nas subseções seguintes.

9.4.1 FOLHA DE TEMPO

A *folha de tempo (timesheet)* é uma forma de manter o registro das ações dos desenvolvedores ao longo do tempo, visando ao estudo para acompanhamento de desempenho e reavaliação de estimativas e do processo em si.

A folha de tempo deve conter as ações nas quais o desenvolvedor se envolveu ao longo de um dia, como escrever um método, revisar um diagrama, entrevistar usuário, participar de reunião, resolver problemas pessoais etc. Não se devem confundir tais ações com a atividade estruturada de um processo, a qual tem artefatos de entrada e de saída bem definidos.

As atividades de projeto costumam ser realizadas ao longo de um dia ou mais. Assim, não são elas que aparecem listadas na folha de tempo, mas as ações concretas executadas pelo desenvolvedor, independentemente de gerarem ou não um artefato de saída.

A folha de tempo será uma excelente ferramenta não só para o gerente acompanhar o andamento do projeto, mas também para analisar possíveis erros de estimativa de esforço, o que poderá melhorar a capacidade de previsão da equipe em projetos futuros. Além disso, é uma ferramenta de autorreflexão em que o desenvolvedor efetivamente registra em que gastou suas horas.

Além de registrar as ações desenvolvidas ao longo do dia, o desenvolvedor ainda poderá adicionar:

- a) Um resumo das ações do dia e das dificuldades encontradas. Nos métodos ágeis, essa informação costuma ser passada oralmente nas reuniões em pé.
- b) A natureza de cada ação a partir de uma lista previamente selecionada (por exemplo, codificação, teste, depuração, planejamento, revisão etc.). Essa informação poderá ser usada posteriormente para verificar quais tipos de ação mais tomam tempo dos desenvolvedores, o que permitirá tomar providências para a correção de gargalos, se for o caso.
- c) O artefato alterado em cada ação, se for o caso. Espera-se, a princípio, que uma ação do desenvolvedor só vá alterar os artefatos que correspondem a saídas da atividade de projeto em que ele está trabalhando, mas tal ação poderá ter vários artefatos relacionados. De outro lado, se cada atividade de projeto tem apenas um artefato de saída, essa anotação dos artefatos poderá ser redundante.
- d) O conjunto de artefatos consultados para cada ação. Embora acrescente peso à folha de tempo, essa informação poderá ser útil para que se verifique quais foram efetivamente os artefatos de entrada para cada atividade de projeto. Se forem consultados artefatos que não estavam previstos como entradas da atividade de projeto, possivelmente alguma coisa estará errada na descrição da atividade ou nas ações do desenvolvedor.
- e) A referência a uma *ficha de acompanhamento de problema (FAP)* para cada ação, se necessário (ver seção seguinte).

Assim, uma folha de tempo poderá se parecer com a imagem da Figura 9.1.

Embora a folha de tempo possa ser feita em papel, o ideal, para permitir uma gerência mais eficiente, é que ela seja automatizada por uma ferramenta. Mesmo sendo um sistema tão simples, que muitas empresas acabam implementando a sua própria maneira, alguns produtos podem ser encontrados. Entre as ferramentas gratuitas

| Projeto: | | | | | | |
|---------------------------------|-----|----------|-----------|---------------------|-----------------------|-----|
| Atividade de projeto: | | | | | | |
| Funcionário: | | | | | | |
| Data: | | | | | | |
| Resumo das ações do dia: | | | | | | |
| Lista de problemas encontrados: | | | | | | |
| Ações: | | | | | | |
| Início | Fim | Natureza | Descrição | Artefatos alterados | Artefatos consultados | FAP |
| | | | | | | |
| | | | | | | |

Figura 9.1 Uma folha de tempo (*timesheet*).

podem-se citar: *Journix Timesheet*¹², *WR Time Tracker*¹³, *HourGuard Timesheet*¹⁴ e *Timesheet4J*¹⁵. Muitas outras, inclusive comerciais, podem ser encontradas na internet.

9.4.2 ACOMPANHAMENTO DE PROBLEMAS

Uma das grandes diferenças entre o processo de desenvolvimento de software e outros processos de engenharia é que o primeiro não pode ser totalmente definido *a priori*. Como novos requisitos são descobertos ou modificados ao longo do projeto, pode-se considerar que o desenvolvimento de software é, antes de tudo, um processo de amadurecimento. Durante esse processo, riscos e dúvidas surgirão e, com eles, *problemas* que precisam ser resolvidos e rastreados.

Além da garantia de solução dos problemas, um controle eficiente pode ser útil também para identificar sua causa. Assim, eles podem ser evitados no futuro. Essa análise também poderá ajudar a identificar problemas no próprio processo, que então poderá ser ajustado.

A identificação inicial de um problema normalmente tem duas fontes: os desenvolvedores e os usuários. Quando o problema é apontado pelo usuário, existem as seguintes possibilidades:

- a) *Trata-se de um problema urgente que deve ser resolvido de forma emergencial*: nesse caso, ele é imediatamente repassado a uma equipe, que vai parar o que estiver fazendo para achar uma solução para o problema (contudo, essa não pode ser a regra para tratar todos os problemas).
- b) *Trata-se de um problema real, mas não urgente*: nesse caso, o problema vai gerar uma entrada na lista de problemas conhecidos da atual versão do artefato e também na lista de modificações solicitadas. A versão seguinte possivelmente vai resolver o problema.
- c) *Trata-se de um problema real, porém já resolvido numa versão mais atual que a do usuário*: nesse caso, o usuário deve ser orientado a atualizar sua versão do software. Uma ferramenta interessante para armazenar problemas e soluções, que permite a criação de um banco de dados de problemas e consulta inteligente, é o *Responsive Knowledgebase*¹⁶.
- d) *Não se trata de um problema real*: nesse caso, o usuário deve ser orientado sobre como proceder ou compreender o sistema.

Uma FAP (*Folha de Acompanhamento de Problema*) é um documento ou entrada em um sistema que vai indicar um problema identificado, sua origem (cliente ou desenvolvedores), sua localização (quem ou qual setor da empresa está com o problema sob sua responsabilidade) e seu estado. Em geral, os possíveis estados de um problema são os seguintes:

- a) Aguardando análise.
- b) Em análise.
- c) Aguardando solução.

¹²Disponível em: <www.journyx.com/>. Acesso em: 21 jan. 2013.

¹³Disponível em: <www.anuko.com/content/time_tracker/>. Acesso em: 21 jan. 2013.

¹⁴Disponível em: <hourguard-timesheet-software.soft32.com/old-version/142705/1.37> . Acesso em: 21 jan. 2013.

¹⁵Disponível em: <code.google.com/p/timesheet4j/> . Acesso em: 21 jan. 2013.

¹⁶Disponível em: <www.responsivesoftware.com/rkb.htm>. Acesso em: 21 jan. 2013.

- d) Em solução.
- e) Aguardando informação complementar.
- f) Aguardando aprovação.
- g) Aprovado/resolvido.

Em geral, o fato de um problema ter sido analisado e tratado não significa necessariamente que ele tenha sido resolvido. Assim, uma análise com fins de aprovação da resolução do problema deve ser feita ao final dos procedimentos de correção. Quando a solução final for aceita, necessariamente será gerada uma nova versão do artefato no sistema de controle de versões.

Como o acompanhamento de problemas deve ser feito por diferentes interessados, é natural que acabe sendo feito por um sistema informatizado. Um exemplo de software nacional gratuito para controle de problemas (manutenção) é o SIGMA¹⁷.

9.4.3 REGISTRO DE ARTEFATOS

O processo de acompanhamento de projeto e controle de problemas depende, de forma crucial, do sistema gerenciador de configuração ou controle de versões. O gerente deve se certificar de que todos os membros da equipe sigam corretamente a política de controle de versão estabelecida na empresa.

Uma política típica é que os desenvolvedores somente tenham acesso a um artefato quando liberado pelo seu proprietário. Eles podem trabalhar na cópia do artefato até realizar a tarefa a que se propuseram. Depois, devem submeter o artefato modificado ao controle de qualidade, em geral na forma de testes. Se aprovado, devem salvar uma nova versão do artefato no sistema de gerenciamento de configuração. Mais detalhes sobre esse processo serão apresentados no Capítulo 10.

9.5 Medição em Engenharia de Software

Um processo de gerência, para ser mais eficaz, precisa se basear em medições. Como saber se a tarefa não está sendo feita se não houver uma medida para chegar a essa conclusão? Como saber que a qualidade é inaceitável? De uma maneira ou de outra, o gerente de projeto vai acabar se envolvendo com a atividade de medição de software, na qual terá que aplicar uma ou mais métricas.

Inicialmente, alguns termos serão definidos para evitar confusão:

- a) *Medida*: valor obtido para alguma dimensão do software.
- b) *Métrica*: escala na qual os valores de uma medida são tomados.
- c) *Medição*: processo de obtenção de medidas.

O *Software Engineering Institute* (Mills E. E., 1988) indica que uma boa métrica precisa ter cinco qualidades:

- a) Ser *simples*, ou seja, ter uma definição curta e fácil de ser compreendida.
- b) Ser o mais *objetiva* possível, isto é, não depender de opiniões. Se duas pessoas avaliarem o mesmo produto usando essa métrica, deverão obter o mesmo resultado.
- c) Ser *facilmente obtida*, isto é, ter custo de medição razoavelmente baixo.
- d) Ser *válida*, isto é, indicar um valor que seja útil e efetivamente representativo da grandeza que se pretende medir.
- e) Ser *robusta*, isto é, pequenas mudanças no produto devem gerar mudanças proporcionais na medida. Apenas grandes mudanças no produto podem produzir grandes mudanças na medida.

Apesar disso, nem sempre as métricas apresentam todas essas qualidades. Um bom exemplo de métrica é a contagem de linhas de código (LOC; Seção 7.1). Desde que os padrões de contagem tenham sido estabelecidos (o que efetivamente conta e o que não conta), essa métrica deve produzir sempre o mesmo resultado. Já a contagem de pontos de função (Seção 7.3.3) a partir de um conjunto de requisitos poderá apresentar pequenas variações, dependendo da interpretação do analista sobre o que efetivamente se constitui em uma função individual do software e dos fatores técnicos. A avaliação dos fatores técnicos e ambientais, também, em pontos de caso de uso (Seção 7.5), é uma métrica

¹⁷Disponível em: <www.redeindustrial.com.br/site/empresa.aspx>. Acesso em: 21 jan. 2013.

bastante subjetiva. O método COCOMO II (Seção 7.3) tenta reduzir essa subjetividade nos multiplicadores de esforço e fatores de escala ao estabelecer uma série de padrões para atribuição de notas. Ainda assim, a métrica tem uma carga de subjetividade significativa.

A Seção 11.5 apresenta mais alguns detalhes sobre medição e métricas referentes à qualidade de software.

9.5.1 CLASSIFICAÇÕES DE MÉTRICAS

Existem várias classificações para métricas. Inicialmente, pode-se falar em métricas diretas, ou seja, aquelas que podem ser definidas e contadas de modo direto, sem necessidade de interpretação ou incerteza, como:

- a) Custo financeiro.
- b) Esforço em desenvolvedor-mês.
- c) Linhas de código (SLOC).
- d) Velocidade de execução em segundos.
- e) Memória em megabytes.
- f) Número de defeitos localizados (total ou relativo ao número de KSLOC).
- g) Complexidade ciclomática (Seção 13.4.1).

Outras métricas são indiretas e só podem ser determinadas a partir de uma definição operacional (Wazlawick, 2009), ou seja, um procedimento de medição que algumas vezes é *ad hoc*, e nem sempre será consenso. Exemplos de métricas indiretas são os seguintes:

- a) Funcionalidade.
- b) Qualidade.
- c) Complexidade.
- d) Eficácia.
- e) Confiabilidade.
- f) Manutenibilidade.
- g) Usabilidade.

Nota-se que as métricas mais importantes estão fortemente ligadas aos aspectos de qualidade do software (Seção 11.1).

Um aspecto que sempre deve ser lembrado é que só vale a pena coletar medições quando se tem em mente um propósito específico, pois a coleta de dados poderá acarretar trabalho extra antes, ao longo e depois do projeto de desenvolvimento. Assim, a coleta de medições deve ser encarada como um investimento de tempo e esforço com o objetivo de melhorar algum aspecto do produto ou do processo de desenvolvimento.

Do ponto de vista dos processos de gerência, pode ser interessante agrupar as métricas em termos de sua utilidade para o gerente. Assim, temos:

- a) *Métricas de produtividade*: custo, esforço (em AFP ou UCP) e KSLOC, que podem ser usadas para verificar o andamento do projeto e possíveis desvios.
- b) *Métricas de qualidade*: número de defeitos, eficiência, confiabilidade e capacidade de manutenção. São usadas para avaliar se o produto satisfaz aos critérios de aceitação para uso por parte do cliente e também critérios internos, que afetam a eficiência da equipe.
- c) *Métricas técnicas*: outros aspectos ligados ao produto, não necessariamente à qualidade ou à produtividade, mas a aspectos inerentes do sistema, como complexidade ciclomática, modularidade, paralelismo, distribuição etc.

As métricas podem ser *absolutas* ou *relativas*. Por exemplo, um sistema com cinco erros não é necessariamente pior do que um sistema com dois erros. Se o primeiro sistema tiver um milhão de linhas e o segundo tiver cinco mil linhas, então o segundo terá mais erros por linha do que o primeiro.

Assim, a medida relativa é frequentemente usada, em especial quando se deseja avaliar a qualidade do produto e do trabalho. Há pelo menos quatro formas relevantes de relativizar uma métrica:

- a) *Pelo tamanho*: divide-se o valor absoluto da métrica pelo número de linhas de código.
- b) *Pela funcionalidade*: divide-se o valor absoluto da métrica pelo número de pontos de função, pontos de caso de uso ou pontos de histórias.

- c) *Pelo tempo*: divide-se o valor absoluto pelo período de tempo. Por exemplo, número de defeitos detectados por mês.
- d) *Por esforço*: divide-se o valor absoluto pelo esforço despendido, geralmente em desenvolvedor-mês ou desenvolvedor-hora. Por exemplo, o número de linhas de código produzidas por desenvolvedor-mês.

Talvez o trabalho mais formal na área de métricas esteja relacionado à medição da complexidade do software, o que é útil para a área de testes (Capítulo 13). Outro tipo de métrica que recebeu muita atenção é a métrica de qualidade ligada ao produto (Capítulo 11), embora às vezes elas sejam contraditórias entre si. Por exemplo, aumentar a flexibilidade (desejável) pode diminuir a eficiência de tempo (indesejável).

Entretanto, algumas métricas de qualidade podem ser especialmente úteis ao gerente de projeto, pois melhorar suas medições será quase sempre muito salutar:

- a) *Métricas de defeitos*: o número de defeitos em um produto de software deveria ser uma métrica objetivamente contável. Porém, encontrar defeitos não é uma tarefa trivial. Assim, normalmente, a medição de defeitos é feita por uma das seguintes técnicas:
 - Número de alterações de *design* ou código.
 - Número de erros detectados nas inspeções de código (Seção 11.4.2).
 - Número de erros detectados nos testes de programa (preferencialmente pela equipe de teste, mas também, possivelmente, pelos usuários).
- b) *Métricas de confiabilidade*: uma vez que se tenha uma medida nominal dos defeitos encontrados em um produto de software, sua confiabilidade pode ser calculada para determinado período de tempo. Se um sistema complexo apresenta certa taxa de falhas ao longo de um mês ou de um ano, pelas Leis de Lehman (Seção 14.1), pode-se esperar que continue exibindo esse comportamento ao longo dos meses ou anos seguintes. Essas medidas podem ser tomadas tanto durante o processo de desenvolvimento quanto durante o período de operação pós-desenvolvimento do sistema.
- c) *Métricas de manutenibilidade*: embora a manutenibilidade seja, a princípio, uma métrica subjetiva, existem estudos que mostram que a complexidade do produto (a complexidade ciclomática, por exemplo) afeta o esforço necessário para encontrar e reparar defeitos no software. Assim, existe uma relação direta entre o número de defeitos encontrados e a complexidade do software com o esforço necessário para fazer manutenções. Novamente, essas atividades de manutenção podem ser tanto aquelas que ocorrem durante a operação do software quanto as modificações que ocorrem durante o desenvolvimento.

Deve-se ficar atento, porém, para o fato de que essas métricas precisam ser sempre interpretadas em seu contexto. Por exemplo, a detecção de muitos defeitos no software pode significar tanto que a atividade de teste está sendo bem conduzida quanto que as atividades de programação estão sendo mal conduzidas. Mas defeitos detectados por usuários durante o uso do sistema costumam ser péssimas notícias.

9.5.2 PLANEJAMENTO DE UM PROGRAMA DE MÉTRICAS

Para que um gerente possa utilizar métricas para dirigir suas atividades, ele deve primeiro implantar um *programa de métricas*. Segundo o SEI (Mills E. E., 1988), essa atividade deve ser muito bem planejada.

Inicialmente, devem-se estabelecer os objetivos do programa de métricas: Quais falhas ele vai corrigir? Quais aspectos ele vai tentar melhorar? Em seguida, devem ser estimados os custos do programa e deve ser obtido o apoio da gerência superior, pois esse tipo de ação normalmente precisa de um suporte razoável para ocorrer. Em relação aos custos, estes podem ser divididos em custos iniciais de implantação e custos de manutenção do programa.

Em função dos objetivos e do aporte financeiro inicialmente comprometido, deve-se escolher o conjunto de métricas a serem implementadas e o modelo de avaliação (por exemplo, para estimar o tamanho de um projeto, podem-se usar os modelos CII, Pontos de Função ou Pontos de Caso de Uso, cada qual com suas peculiaridades e custos específicos). Durante o processo de seleção das métricas e modelos devem-se observar os seguintes pontos:

- a) *Habilidade projetada para satisfazer objetivos*: as métricas e modelos disponíveis devem ser analisados e comparados em relação a sua capacidade de satisfazer os objetivos do programa de métricas.

- b) Dados e custos necessários estimados:** os modelos que são capazes de satisfazer os objetivos devem ser comparados em função de seu custo de implantação e manutenção e da quantidade e variedade de dados necessários para funcionar. Normalmente, os modelos mais econômicos são preferíveis.

Uma vez que o modelo tenha sido escolhido, deve-se identificar e refinar os dados que serão coletados. Apenas dados que possam satisfazer a algum objetivo imediato ou de longo prazo devem ser coletados. Deve-se evitar coletar quaisquer dados só porque estão disponíveis, pois o excesso de dados pode causar confusão quando se tenta fazer algum tipo de análise. Para executar essa atividade, deve-se observar o seguinte:

- a) Especificidade dos dados:** os dados devem ser definidos e obtidos ao longo de todo o ciclo de desenvolvimento do software. Preferencialmente, deve-se identificar *quando* os dados foram obtidos. Isso permite analisar diferentes significados em diferentes fases ou para diferentes atividades do processo de desenvolvimento.
- b) Procedimentos de obtenção de dados:** uma vez que os dados específicos tenham sido definidos, os procedimentos para sua coleta e as pessoas responsáveis devem ser identificados.
- c) Manutenção do banco de dados:** uma vez que o banco de dados de medições passará a ser um importante patrimônio da empresa, os recursos para sua perfeita manutenibilidade devem ser definidos e destinados.
- d) Previsões refinadas de esforço e custo:** com as informações obtidas nos itens anteriores, deve ser possível obter uma estimativa bem mais realista de custo e esforço para a implantação do programa de métricas.

Assumindo que todos os passos anteriores foram executados com sucesso e os custos são aceitáveis, o programa de métricas pode ser iniciado. Os seguintes itens ainda precisam ser enfatizados nessa fase:

- a) Esclarecimento de uso:** os objetivos do programa de métricas devem ficar claros desde o início. Mas, no momento de iniciar seu uso, pode ser importante relembrá-los a toda a equipe. As pessoas devem ser informadas sobre as medidas que serão obtidas e o uso que será feito delas. Muito cuidado deve ser tomado especialmente se as medidas forem utilizadas para avaliação de membros da equipe. Sem uma ampla discussão e aceitação das métricas, essas iniciativas poderão dar origem a estresse e até sabotagens ao programa.
- b) Pessoal responsável:** os responsáveis pela coleta, manutenção e interpretação dos dados devem ser definidos e informados. Deve-se lembrar que essa atividade deve ser executada continuamente ao longo de qualquer projeto, pois muitos dados poderão não ser mais obtidos depois que ele terminar.

Para que o programa de métrica tenha sucesso, ele deve ser continuamente usado, avaliado e reajustado. Os seguintes aspectos são relevantes:

- a) Avaliação de resultados:** os resultados deveriam ser cuidadosamente resumidos e comparados com a realidade subjetivamente observada. Por vezes, algum desvio em relação à percepção da equipe pode ser resultado de erros no processo de obtenção dos dados.
- b) Ajuste do modelo:** muitos modelos de medição exigem que determinadas constantes sejam calibradas ao longo de seu uso (por exemplo, CII). Assim, esses procedimentos de calibração devem ser executados sempre que o modelo assim o exigir.

Finalmente, convém lembrar que, em termos de capacidade de processos e maturidade de empresas (CMMI), é absolutamente necessário que exista um plano de métrica e que ele seja efetivamente usado para melhorar aspectos dos processos da empresa. Os níveis 4 e 5 de maturidade do CMMI só são atingidos caso exista um plano de métricas (Seção 12.3).

9.6 Revisão e Avaliação

Nos métodos ágeis, como *Scrum* e *XP*, a revisão e a avaliação de projeto são previstas e ocorrem informalmente nas reuniões diárias em pé e com mais formalidade nas reuniões de fechamento de iteração.

Outros métodos poderão definir reuniões de revisão de projeto, de forma periódica, seja ao final de uma iteração ou fase, seja a qualquer momento.

É importante que, para serem efetivas, as reuniões sejam planejadas com antecedência. As pessoas envolvidas devem ser comunicadas, o local adequado deve ser reservado e, o que é mais importante, os objetivos da reunião devem ser claramente transmitidos. Quando se fala em objetivos, deve-se entender isso no sentido de definir quais

serão os *artefatos de saída* da reunião. Pode-se ver, então, que a reunião de revisão e avaliação também é vista como uma atividade de projeto. Quando os participantes de uma reunião começam a divagar e a discutir sem objetividade, cabe ao condutor da reunião (gerente) interromper a divagação e solicitar sugestões de encaminhamento para os assuntos pendentes, de forma que os objetivos da reunião possam ser atingidos o mais rapidamente possível.

Além disso, sempre que possível, todo o material a ser usado na reunião deve ser distribuído previamente. A reunião não é o momento de apresentar novidades ou surpresas para a equipe. É recomendável que, sempre que possível, os assuntos sejam adiantados; caso contrário, a reunião poderá se estender demais.

Não é recomendado que, na reunião de revisão e avaliação, se faça com o grupo uma análise minuciosa, por exemplo, de todo o código produzido pela equipe. Isso é impossível por questões de tempo. As revisões e os relatórios já deverão estar prontos para a reunião, tendo sido feitos por aqueles a quem a tarefa foi delegada. No momento da reunião, apenas uma visão geral deve ser apresentada e, se necessário, o aprofundamento de um ou outro pontos críticos.

Para a pauta da reunião sugere-se que o gerente de projeto procure apresentar os seguintes aspectos:

- a) Os principais marcos de projeto (*milestones*) alcançados.
- b) Os desvios que eventualmente tenham ocorrido em relação ao plano de desenvolvimento do projeto.
- c) Modificações significativas na alocação de esforço, ou seja, se mais (ou menos) tempo do que o previsto foi dedicado a alguma atividade.
- d) Modificações significativas em termos de despesas, ou seja, se as atividades consumiram mais (ou menos) itens de orçamento do que o previsto.
- e) Mudanças no escopo estimado de trabalho, ou seja, se houve alguma modificação em termos das funcionalidades ou outros aspectos que inicialmente consistiam em um objetivo da fase ou ciclo e que foram mudados, removidos ou acrescentados.
- f) Mudanças na métrica de qualidade, ou seja, se por alguma razão a forma de medir a qualidade do trabalho foi mudada.
- g) *Status* dos riscos de projeto, ou seja, quais riscos continuam sendo muito importantes, quais tiveram sua importância alterada ao longo da fase ou do ciclo (especialmente caso a alteração tenha sido no sentido de um risco ficar mais importante do que era anteriormente).
- h) Novos riscos identificados.
- i) Problemas relevantes que tenham surgido e ainda não tenham sido resolvidos.
- j) Andamento de eventuais ações que tenham sido determinadas em reuniões anteriores.
- k) Lições aprendidas para projetos futuros.

Durante a reunião e especialmente ao final dela devem ser tomadas decisões sobre ações a serem executadas, sempre que necessário. O gerente deve indicar responsáveis e prazos para as ações mais urgentes. Já as ações não urgentes devem ser colocadas na lista de modificações ou na lista de riscos de projeto para serem tratadas oportunamente, de acordo com sua prioridade.

9.7 Fechamento

O fechamento ou encerramento de um projeto (ou de uma fase de um projeto) tem como objetivo formalizar a entrega do produto e a sua aprovação pelo cliente.

Será uma reunião ainda mais formal do que as reuniões periódicas de avaliação do projeto e, possivelmente, deverá envolver um número maior de interessados. Dentre os itens tratados nas reuniões de fechamento de projeto, pode-se imaginar que vários serão semelhantes aos itens abordados nas reuniões de avaliação. Porém, em uma avaliação final, será importante observar os seguintes aspectos:

- a) Se todos os objetivos iniciais do projeto foram alcançados.
- b) Se houve algum desvio importante ao longo do projeto em relação ao seu plano.
- c) Se houve alguma mudança significativa em relação ao esforço previsto e executado (o que poderá servir de entrada para a calibragem de métricas de esforço).
- d) Se houve alguma mudança significativa em relação aos recursos alocados e consumidos.
- e) Se houve mudanças de escopo importantes.

- f)** Os resultados finais das métricas de qualidade e, eventualmente, sua evolução.
- g)** Riscos de operação, caso tenham sido identificados.
- h)** Problemas relevantes que tenham surgido e ainda não tenham sido resolvidos.
- i)** Lições aprendidas para projetos futuros.

Além de formalizar a entrega do produto, a reunião de encerramento do projeto também será uma oportunidade importante para que sejam produzidos documentos reportando o desempenho da empresa, bem como os problemas de percurso e as soluções encontradas. Dessa forma, a inteligência criada durante o projeto fica registrada para possível uso futuro.

A finalização do projeto também é importante para que fique claro para todos os envolvidos que as atividades de desenvolvimento terminaram e que quaisquer novas modificações serão alvo de projetos futuros (ou do processo contínuo de manutenção).

Gerenciamento de Configuração e Mudança

Este capítulo vai apresentar a disciplina de gerenciamento de configuração e mudança, iniciando pelos *conceitos básicos* (Seção 10.1), como itens de configuração, versões, *baselines* e *releases*. Na seção seguinte é apresentado o funcionamento de um sistema de controle de versão (Seção 10.2) para desenvolvimento de produtos de software. Em seguida são apresentados os conceitos de *controle de mudança* (Seção 10.3) e *auditoria de configuração* (Seção 10.4). O capítulo termina com uma apresentação de ferramentas para controle de configuração e mudança (Seção 10.5).

O Gerenciamento de Configuração de Software (GCS)¹ ou Gerenciamento de Configuração e Mudança (GCM) é considerado uma disciplina à parte dentro do gerenciamento de projetos. Essa área é especialmente importante na indústria de software, porque componentes e produtos de software são desenvolvidos e modificados muitas vezes ao longo do tempo, durante e após o projeto. Assim, diferentes versões de um mesmo componente ou produto podem estar disponíveis em diferentes momentos.

O gerenciamento de configuração, portanto, é a área que vai indicar como as diferentes versões dos artefatos envolvidos no desenvolvimento de software devem ser modificadas e identificadas.

Pressman (2005) indica que o gerenciamento de configuração e mudança deve ter como objetivo responder às seguintes perguntas:

- a)** O que mudou e quando mudou?
- b)** Por que mudou?
- c)** Quem fez a mudança?
- d)** Pode-se reproduzir essa mudança?

Essas questões serão respondidas pelas três grandes atividades do gerenciamento de configuração e mudança:

- a)** O *controle de versão* vai dizer *o que mudou e quando*.
- b)** O *controle de mudança* vai dizer *por que* as coisas mudaram.
- c)** A *auditoria de configuração* vai dizer *quem fez a mudança e como ela pode ser reproduzida*.

Essas atividades serão detalhadas nas Seções 10.2 a 10.4.

¹Software Configuration Management (SCM).

10.1 Conceitos Básicos

Antes de discorrer mais detalhadamente sobre as atividades de gerenciamento de configuração, alguns conceitos básicos serão apresentados nas próximas subseções.

10.1.1 ITEM DE CONFIGURAÇÃO DE SOFTWARE

Um *Item de Configuração de Software* (ICS)² é um elemento unitário para efeito de controle de versão, ou, ainda, um agregado de elementos que são tratados como uma entidade única no sistema de gerenciamento de configuração.

Não apenas o código de programação deve ser controlado pelo gerenciamento de configuração, mas também a documentação, os diagramas, os planos, as ferramentas, os casos de teste, os dados e quaisquer outros artefatos relacionados ao desenvolvimento do software.

Em geral, cada item de configuração recebe um número, que poderá inclusive ser identificado por várias partes que costumam ser separadas por ponto:

- a) Plano de projeto, versão 2.
- b) Biblioteca de funções matemáticas, versão 4.1.
- c) Disco de instalação do sistema, versão 1.2.4.

Um dos problemas com o gerenciamento de configuração consiste em determinar a melhor granularidade para os itens de configuração. Ter itens demais poderá dificultar seu controle, e as configurações de software, formadas por um conjunto de itens de configuração, serão listas muito extensas. De outro lado, ter itens de menos também poderá dificultar o controle, porque cada item terá um número muito grande de versões.

Em sistemas orientados a objetos, em geral, um item de configuração terá a granularidade de um pacote ou componente de classes afins que não ultrapasse poucas dezenas. Mas, dependendo do projeto, esse tamanho poderá variar. Projetos muito grandes tenderão a ter itens maiores, com mais classes, e projetos muito pequenos poderão ter até classes individuais definidas como itens de configuração.

Os itens de configuração podem ser básicos ou compostos. Os *básicos* são os artefatos arbitrados como individuais para efeito de controle de versão. Os *compostos* podem ser formados por outros itens básicos e compostos. Cada item de configuração será definido por quatro elementos:

- a) Um *nome*, que é uma cadeia de caracteres que identifica claramente o item básico ou composto.
- b) Uma *descrição*, que consiste da definição do tipo de item (documento, diagrama, dados, código fonte etc.) e detalhes sobre ele.
- c) Uma lista de *recursos*, que são outros itens de configuração exigidos pelo item básico. No caso de itens compostos, a lista de recursos poderá ser definida como a união das listas de seus itens básicos.
- d) Uma *realização*, que, no caso do item básico, é um ponteiro ou endereço para o artefato que efetivamente realiza o item. No caso de itens compostos, a realização é definida como o conjunto das realizações de seus itens básicos.

10.1.2 RELACIONAMENTOS DE ITENS DE CONFIGURAÇÃO DE SOFTWARE

Conforme visto na seção anterior, um ICS pode estar ligado a outros a partir de uma lista de recursos exigidos. Podem existir vários tipos de dependência ou relacionamento entre os itens de software. A maioria desses relacionamentos é prevista nas ferramentas CASE, que permitem desenhar diagramas de classe UML. Tais tipos de relacionamento podem tanto ser representados por associações de um tipo específico quanto por associações estereotipadas. A lista a seguir, embora não seja exaustiva, apresenta alguns exemplos de relacionamentos entre ICS que também devem ser mantidos por um sistema de controle de configuração de software:

- a) *Dependência*: costuma indicar que um componente utiliza funções que são implementadas em outro componente. A associação pura e simples da UML não deixa de ser uma forma de dependência, nesse sentido, já que uma classe vai estar associada à outra normalmente para poder utilizar funções implementadas na outra.

²Software Configuration Item (SCI).

- b) *Agregação e composição*: indicam que um componente é formado por outros.
- c) *Realização*: indica que um componente concreto é a implementação de um componente mais abstrato.
- d) *Especialização*: indica que um componente é uma variante mais específica de outro. Em geral, o componente mais especializado depende do componente mais genérico. O inverso só será verdade se o componente genérico tiver métodos abstratos que precisem ser necessariamente implementados no componente mais especializado. Nesse caso, pode-se dizer que existe dependência mútua entre os componentes.

10.1.3 RASTREABILIDADE

Rastreabilidade ou *controle de rastros* refere-se a um dos princípios da Engenharia de Software cuja implementação ainda implica significativa dificuldade.

Manter um controle de rastros entre *módulos de código* não é difícil, porque as dependências entre os módulos costuma ser indicada de forma sintática pelos próprios comandos da linguagem (*import* ou *uses*, por exemplo). Contudo, manter controle de rastros entre artefatos de análise e *design* não é tão simples. A rastreabilidade é importante, porque, quando são feitas alterações em artefatos de análise, *design* ou código, é necessário manter a consistência com outros artefatos, caso contrário a documentação fica rapidamente desatualizada e inútil.

A técnica de rastreabilidade mais utilizada nas ferramentas CASE é a *matriz de rastreabilidade*, que associa elementos entre si, por exemplo, casos de uso e seus respectivos diagramas de sequência ou classes e módulos de programa. Porém, esse tipo de visualização matricial torna-se impraticável à medida que a quantidade de elementos cresce, especialmente se o controle das relações entre os elementos precisar ser feito manualmente (Cleland-Huang, Zemont & Lukasik, 2004).

Também são reportadas pesquisas que procuram automatizar a recuperação de relações de rastreabilidade entre artefatos a partir de evidências sintáticas. Porém, em razão das escolhas arbitrárias dos nomes de artefatos (falta de padrão), esses sistemas de recuperação costumam retornar muitos falso-positivos, o que inviabiliza seu uso (Settimi, Cleland-Huang, Khadra, Mody, Lukasik & de Palma, 2004).

Outra abordagem, ainda experimental, é apresentada por Santos e Wazlawick (2009). Consiste em modificar a maneira como os elementos são criados nos editores de diagramas das ferramentas CASE, de forma que, com exceção dos requisitos que são produzidos externamente, outros elementos quaisquer (como casos de uso, classes, código, diagramas etc.) só podem ser criados como uma derivação de algum outro elemento que já exista.

Dessa forma, sempre que um item é criado no repositório do projeto, um rastro é automaticamente criado entre ele e o elemento que lhe deu origem. Apenas itens que representam requisitos (que vêm do cliente, ou seja, são externos ao projeto) podem ser adicionados ao repositório sem que isso ocorra por derivação de outros itens.

10.1.4 VERSÕES DE ITENS DE CONFIGURAÇÃO DE SOFTWARE

A *versão* de um ICS é um estado particular desse item durante o desenvolvimento de um sistema. Normalmente, uma versão é identificada por um número.

Versões de ICS sucedem-se no tempo. Pode-se dizer que há basicamente dois tipos de sucessores para uma versão:

- a) uma *revisão*, que é uma nova versão de um item que foi criada para substituir uma versão anterior;
- b) uma *variante*, que é uma nova versão do item que será adicionada à configuração sem a intenção de substituir uma versão antiga.

Pode-se falar, ainda, em versões experimentais para determinados itens: uma revisão é feita no item, mas ainda não está decidido se a nova versão será efetivamente mantida. Nesse caso, a revisão pode até ser considerada uma variante, já que ainda não existe a determinação de que a versão anterior é obsoleta.

10.1.5 CONFIGURAÇÃO DE SOFTWARE

Uma *configuração de software* é o estado em que o software se encontra em determinado momento. A configuração de um sistema pode incluir todos os elementos físicos ou abstratos relacionados com o produto. Porém, uma configuração de software normalmente incluirá apenas o estado dos artefatos que são mantidos em formato eletrônico,

como os programas, documentos eletrônicos, ferramentas CASE utilizadas no desenvolvimento do sistema, sistema operacional, bibliotecas de suporte, e assim por diante.

A configuração de software é, portanto, definida como uma lista dos ICS que compõem o software e suas respectivas versões. Cada uma dessas versões deve ser armazenada de maneira que possa ser recuperada sempre que determinada configuração do software precise ser reproduzida. A Seção 10.5 vai tratar de sistemas de controle de versão que auxiliam o gerente de configuração a manter o controle das diferentes versões desses itens.

10.1.6 *BASELINE E RELEASE*

Uma *baseline* é uma configuração de software especialmente criada para uma situação específica. Ela é formalmente aprovada em determinado momento do ciclo de vida do software e servirá de referência para um desenvolvimento posterior.

A *baseline* só pode ser modificada através de um processo formal de mudança. Juntamente com todas as mudanças aprovadas para ela, representa a configuração correntemente aprovada. Um exemplo de *baseline* é um projeto que foi aprovado para execução. Uma vez aprovado por todos os interessados, ele só pode ser modificado mediante processos formais (por exemplo, termos aditivos). Outro exemplo de *baseline* poderia ser um conjunto de requisitos e interfaces aprovados pelo cliente para desenvolvimento. Outro exemplo, ainda, poderia ser uma versão do sistema entregue ao cliente e aprovada nos testes de aceitação.

Em todos os casos, a ideia da *baseline* é que ela deverá servir de base para desenvolvimentos posteriores e que o que ali está, a princípio, não deve ser mudado. Já no caso de uma configuração de software que não seja *baseline*, esta pode ser alterada sem maiores formalidades.

Também é possível considerar que mudanças feitas após o estabelecimento de uma *baseline* serão consideradas mudanças “delta” até que uma nova *baseline* seja definida.

Release ou *entrega* é a distribuição de uma versão do software (ou mesmo de um ICS) para fora do ambiente e desenvolvimento. A vantagem do uso de sistemas de controle de versão reside no fato de que *releases* atuais ou anteriores podem ser geradas a qualquer momento a partir das *baselines* e das mudanças armazenadas para elas.

10.2 Controle de Versão

A falta de controle de versão pode levar a problemas bastante característicos. Responder “sim” a qualquer uma das questões a seguir indica que um projeto carece desse tipo de controle³:

- a) Já perdeu uma versão anterior do arquivo do projeto e precisou dela?
- b) Já teve dificuldade em manter duas versões diferentes do sistema rodando ao mesmo tempo?
- c) Alguém já modificou indevidamente um código-fonte e o original não poderia ter sido perdido?
- d) Tem dificuldade em saber quem modificou o que em um projeto?

O controle de versão vai rastrear todos os artefatos do projeto (itens de configuração) e manter o controle sobre o trabalho paralelo de vários desenvolvedores através das seguintes funcionalidades:

- a) Manter e disponibilizar cada versão já produzida para cada item de configuração de software.
- b) Ter mecanismos para disponibilizar diferentes ramos de desenvolvimento de um mesmo item, ou seja, diferentes variantes ou variações de um item poderão ser desenvolvidas em paralelo.
- c) Estabelecer uma política de sincronização de mudanças que evite a perda de trabalho e o retrabalho.
- d) Fornecer um histórico de mudanças para cada item de configuração.

O controle de versão vai, assim, manter um histórico dos ICS. Se, por exemplo, foi feita uma alteração nos métodos de uma classe para modificar uma funcionalidade ou otimizar um procedimento qualquer e depois se descobriu que ela não deveria ter sido feita ou que foi feita de maneira inadequada e deve ser desfeita, então o controle de versão vai permitir desfazer a mudança (*undo*) e fazer o artefato retornar ao seu estado anterior. Dessa forma, diferentes modificações podem ser tentadas em um artefato, seja código, diagrama, seja texto, sem maiores riscos, já que

³Disponível em: <pt.wikipedia.org/wiki/Ger%C3%Aancia_de_configura%C3%A7%C3%A3o_de_software> . Acesso em: 21 jan. 2013.

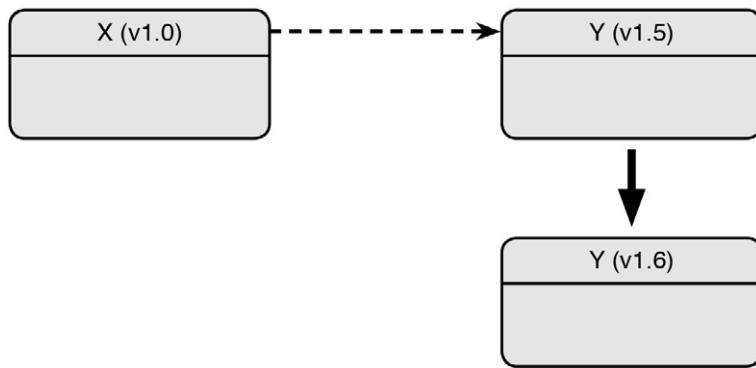


Figura 10.1 Uma classe que depende de uma versão desatualizada de outra classe.

qualquer modificação pode ser desfeita posteriormente. Tal característica é fundamental em qualquer projeto de desenvolvimento de software não trivial.

Não apenas os itens de configuração devem ser controlados pelo sistema de gerenciamento de configuração, mas também os relacionamentos entre eles (Seção 10.1.2). Alguns itens poderão ser compatíveis apenas com determinadas versões de outros itens ou, ainda, necessitar de conexões com outros itens a partir de determinada versão.

Apenas o teste de integração vai poder dizer se dois ICS são compatíveis entre si. Por exemplo, se uma classe X versão 1.0 dependia de métodos da classe Y versão 1.5, então, se a classe Y for atualizada para a versão 1.6, talvez ela não seja mais compatível com a classe X versão 1.0 (Figura 10.1).

Apenas depois que a integração entre Y 1.6 e X 1.0 passar nos testes é que essa nova dependência poderá ser aprovada. Até esse ponto, a classe X 1.0 continuará a depender de Y 1.5 (com uma anotação no sistema de versões de que Y 1.5 está desatualizada e que uma nova integração é necessária).

10.2.1 REPOSITÓRIO

Todos os arquivos referentes às realizações dos itens de configuração ficam em um local chamado *repositório*, sob a guarda do sistema de controle de versão. O repositório pode ser entendido como um local (diretório) em que as diferentes versões de cada item são mantidas e identificadas. Esse controle poderá ser manual, mas o ideal é que seja automático.

Um bom sistema de controle de versões deverá ser capaz de otimizar o espaço de armazenamento do repositório. Em geral, tal sistema deverá ser capaz de armazenar apenas o artefato original e, dali para a frente, armazenar apenas as modificações que forem feitas, e não novas cópias completas do mesmo artefato. Dessa forma, uma versão qualquer será produzida pelo sistema a partir da versão original, na qual o sistema aplicará sequencialmente todas as modificações até chegar à versão desejada.

Para otimizar a velocidade de acesso, uma vez que normalmente a última versão é a que será efetivamente utilizada na maioria das vezes, o sistema pode armazenar em *cache* uma cópia completa dessa última versão, sem que haja necessidade de gerá-la a partir da original e das modificações, como as anteriores. Quando essa versão deixar de ser a atual, a cópia pode ser apagada, ficando no repositório apenas as modificações que permitem gerá-la a partir das versões anteriores e a cópia da nova versão atual.

10.2.2 POLÍTICAS DE COMPARTILHAMENTO DE ITENS

Os desenvolvedores não trabalham diretamente sobre os itens do repositório, mas sobre cópias desses itens. Assim, um desenvolvedor deve solicitar acesso a determinado item no repositório, obtendo uma cópia dele. Esse desenvolvedor vai fazer as alterações nessa cópia do item e, posteriormente, salvar no repositório uma nova versão do item.

Deve-se decidir o que fazer caso mais de um desenvolvedor esteja trabalhando sobre o mesmo item (concorrência). Nesse caso, existem duas políticas usuais:

- a) Política *trava-modifica-destrava* (*exclusive lock*), na qual um desenvolvedor que copia um item para modificá-lo deve travar o item no repositório de forma que nenhum outro desenvolvedor poderá alterá-lo

até que a modificação seja concluída e a nova versão seja salva. Essa política tem como vantagem o fato de evitar colisões, isto é, situações em que dois desenvolvedores alteram um item e as alterações do primeiro a salvar acabam sendo perdidas porque o segundo vai salvar suas próprias modificações sobre uma versão anterior às modificações do primeiro. Mas a desvantagem dessa política reside no fato de que, muitas vezes, os desenvolvedores até poderiam trabalhar simultaneamente sobre um mesmo item, pois vão alterar partes diferentes dele, mas mesmo assim terão de esperar, perdendo tempo ou realizando outras tarefas menos importantes.

- b) Política *copia-modifica-resolve* (*optimistic merges*), na qual dois desenvolvedores ou mais podem trabalhar simultaneamente sobre um mesmo item e, no momento de salvar a nova versão, resolver entre si possíveis interferências. Na prática, os conflitos são raros e causados por falta de comunicação entre os desenvolvedores, e a mesclagem das versões pode ser feita automaticamente pelo sistema de controle de versões.

No caso da política copia-modifica-resolve, na maioria das vezes as alterações de um item vão ocorrer em locais diferentes e o próprio sistema poderá resolvê-las. Nas poucas vezes em que as alterações ocorrerem na mesma parte do artefato, os desenvolvedores deverão decidir como tratá-las. Esse tipo de conflito, porém, normalmente ocorre somente quando há uma divisão de trabalho inadequada.

Quando um sistema de controle de versões é usado, normalmente o tratamento de conflitos na política copia-modifica-resolve ocorre da seguinte forma:

- a) O desenvolvedor A pega uma cópia do item X para modificar (por exemplo, versão 1.1).
- b) O desenvolvedor B pega outra cópia do mesmo item X para modificar (versão 1.1).
- c) O desenvolvedor A salva (*commit*) uma nova versão de X com suas alterações (gerando a versão 1.2).
- d) Quando o desenvolvedor B vai salvar sua cópia (baseada na versão 1.1), o sistema avisa que a versão do item está desatualizada (porque já existe a versão 1.2) e que o desenvolvedor B deve fazer uma mesclagem (*merge*) de sua cópia com a versão 1.2. Um bom sistema vai mostrar exatamente quais linhas mudaram da versão 1.1 para 1.2 e também se o *merge* das duas versões implica alterar as mesmas linhas.

Dessa forma, o desenvolvedor B deverá avaliar as alterações do desenvolvedor A e verificar se existe conflito entre estas e as que ele próprio produziu. Se necessário, os dois desenvolvedores devem conversar para resolver possíveis conflitos. Ao final do processo, o desenvolvedor B salvará a versão 1.3 do item.

10.2.3 ENVIO DE VERSÕES

Normalmente, o *envio de versões* (*commit*) é feito a critério do desenvolvedor. Porém, é importante que uma nova versão de qualquer artefato só seja enviada ao repositório se estiver minimamente estável, isto é, razoavelmente livre de defeitos.

No caso de artefatos de código, isso pode ser garantido com a realização de testes de unidade, os quais devem inclusive acompanhar o código como parte do item de configuração.

10.3 Controle de Mudança

O *controle de mudança* ou *gerência de mudança* é uma parte importante da gerência de configuração que permite saber por que determinada versão de um ICS foi sucedida por outra.

Um típico controle de mudança de um sistema de software vai indicar quais funcionalidades foram adicionadas, removidas ou modificadas, quais defeitos foram corrigidos e, eventualmente, quais pendências ainda restam para uma versão futura. Por exemplo, um arquivo de controle de mudança de um sistema poderia conter a seguinte informação:

- a) Mudanças da versão 2.2 para a versão 2.3:
 - Correção do defeito D345.
 - Correção do defeito D346.
 - Adicionada a funcionalidade do requisito R43.
 - Aprimorada a usabilidade da interface I12.

- b)** Pendências para uma versão posterior:
- Defeito D347.
 - Melhorar a usabilidade da interface I13.

As modificações de uma versão para outra podem ser tanto aquelas que já estavam no plano de desenvolvimento do software, como a adição de funcionalidades referentes aos requisitos ou casos de uso próprios dos ciclos iterativos, ou a inclusão de mudanças solicitadas pelo usuário, quando este não fica totalmente satisfeito com as funcionalidades implementadas durante os testes de aceitação, ou ainda a inclusão de mudanças referentes a características que não foram incorporadas em um ciclo iterativo por falta de tempo e foram retiradas do escopo do ciclo. Na fase de produção (evolução e manutenção do software), o gerenciamento de mudança fará o controle das atividades de manutenção corretiva, adaptativa e perfectiva (Seção 14.2).

Além desse tipo de controle, é possível haver um controle automatizado das mudanças, pois o sistema gerenciador de versões é capaz de comparar duas versões de um artefato e indicar exatamente quais elementos foram alterados e por quem. Dessa forma, quando uma nova versão de um sistema é gerada a partir de um conjunto de itens de configuração, é possível também gerar automaticamente um descritivo das mudanças feitas. Porém, essa mudança será unicamente sintática, costumando ser necessário adicionar um descritivo que indique a motivação da mudança. Por exemplo, não basta informar que a linha X teve seu código alterado para tal sequência de caracteres; é necessário informar *por que* isso foi feito.

10.4 Auditoria de Configuração

A auditoria de configuração é uma atividade que tem como objetivo verificar se os itens de configuração presentes em uma versão ou *baseline* são realmente aqueles que deveriam estar ali. Em relação à *baseline* ou versão, a auditoria deve ainda verificar se todos os itens necessários estão realmente presentes no repositório.

A auditoria também pode ter como finalidade verificar a consistência da documentação fornecida ao usuário com a configuração de sistema entregue.

Uma auditoria de configuração pode ser executada seguindo os passos abaixo:

- a) Preparar um relatório que liste cada item a ser verificado na *baseline* e o procedimento de teste a ser efetuado.
- b) Efetuar os testes e anotar os itens que passaram no teste.
- c) Se houver algum item que não passou no teste, anotar o fato no documento de *descobertas da auditoria* para encaminhar ao setor responsável para providências.

10.5 Ferramentas para Controle de Versão

O controle de versões pode até ser feito manualmente, com a utilização de diretórios e padrões de nomeação de arquivos, mas esse tipo de controle não é muito efetivo quando se deseja obter relatórios de versões ou quando se realiza o desenvolvimento de variantes. Além disso, essa forma de controle consome espaço de armazenamento, pois cada versão é armazenada integralmente no diretório.

Assim, o mais adequado para uma empresa de desenvolvimento de software seria o uso de um sistema automatizado de controle de versões. Existem algumas soluções livres, como:

- a) CVS⁴, ou *Concurrent Version System*, que é uma solução de código aberto baseada em arquitetura cliente-servidor. O sistema foi iniciado por Dick Grune em 1986.
- b) Mercurial⁵, que é um sistema de controle de versões baseado em linha de comando, desenvolvido por Matt Mackall.

⁴Disponível em: <savannah.nongnu.org/projects/cvs/>. Acesso em: 21 jan. 2013.

⁵Disponível em: <mercurial.selenic.com/>. Acesso em: 21 jan. 2013.

- c) Git⁶, que é um sistema de controle de versão que enfatiza o aspecto de *performance* e foi criado por Linus Torvalds para o desenvolvimento do *kernel* do Linux.
- d) SVN⁷, ou *Subversion*, que foi projetado para substituir e suprir algumas limitações do CVS.

Além disso, existem versões comerciais de software para controle de versões:

- a) *SourceSafe*⁸, da Microsoft.
- b) *Serena PVCS Version Manager*⁹.
- c) *ClearCase*¹⁰, da IBM, que é uma das ferramentas disponibilizadas para o RUP.

O SVN vem de fato substituindo o CVS, especialmente na comunidade de software livre.

⁶Disponível em: <git-scm.com/> . Acesso em: 21 jan. 2013.

⁷Disponível em: <subversion.apache.org/> . Acesso em: 21 jan. 2013.

⁸Disponível em: <msdn.microsoft.com/pt-br/library/ms181038%28v=vs.80%29.aspx> . Acesso em: 21 jan. 2013.

⁹Disponível em: <www.serena.com/products/pvcs-pro/> . Acesso em: 21 jan. 2013.

¹⁰Disponível em: <www-01.ibm.com/software/awdtools/clearcase/> . Acesso em: 21 jan. 2013.

QUALIDADE

A terceira parte do livro aborda os seguintes itens do SWEBOK: “Qualidade de Software”, “Teste de Software” e “Manutenção de Software”.

O assunto “qualidade” foi dividido em duas partes neste livro: *qualidade do produto* (Capítulo 10) e *qualidade do processo* (Capítulo 11).

A área de teste de software (Capítulo 13) também pode ser vista como uma das formas de garantir a qualidade, pois os testes procuram identificar defeitos do software.

Finalmente, a *manutenção* (ou *evolução*) de software (Capítulo 14) foi incluída nesta parte porque trata do processo de manter e evoluir a qualidade do software após sua entrega ao cliente final.



Qualidade de Produto

Este capítulo apresenta os conceitos de *qualidade de produto*, iniciando com o modelo de qualidade da recém-aprovada Norma ISO/IEC 25010:2011 (Seção 11.1), que define os atributos de qualidade internos, externos e de uso de produtos de software. Em seguida, o capítulo apresenta um método para construir outros modelos de qualidade a partir de critérios definidos (Seção 11.2). Na sequência, apresenta informações sobre como instalar um *programa de melhoria de qualidade de produto* (Seção 11.3) e como fazer a *gestão da qualidade* (Seção 11.4). Finalmente, são apresentados os conceitos de *medição da qualidade* (Seção 11.5), *requisitos de qualidade* (Seção 11.6) e o método *GQM* (Seção 11.7), usado para avaliação da qualidade de produtos de software.

Qualidade de software é uma área dentro da Engenharia de Software que visa garantir bons produtos a partir de bons processos. Pode-se falar, então, de dois aspectos da qualidade: a *qualidade do produto* em si e a *qualidade do processo*. Embora não exista uma garantia de que um bom processo vá produzir um bom produto, em geral admite-se que a mesma equipe com um bom processo vá produzir produtos melhores do que se não tivesse processo algum.

Qualidade de software é um assunto amplo e de definição difusa. Existem várias dimensões de qualidade, e nem sempre é simples avaliar objetivamente cada uma delas. Pressman (2005) define dois tipos de qualidade para o produto de software:

- a) *Qualidade de projeto*, que avalia quão bem o produto foi projetado.
- b) *Qualidade de conformação*, que avalia quão bem o produto atende aos requisitos.

Em relação à qualidade, o SWEBOK¹ faz uma distinção entre técnicas estáticas e dinâmicas. As técnicas estáticas são apresentadas neste capítulo como “qualidade de software” e as técnicas dinâmicas são relacionadas ao teste do software (Capítulo 13).

11.1 Modelo de Qualidade SquaRE – ISO/IEC 25010:2011

A Norma NBR ISO/IEC 9126-1:2003² define *qualidade de software* como “a totalidade de características de um produto de software que lhe confere a capacidade de satisfazer necessidades explícitas e implícitas”. A definição, propositalmente, não especifica os possuidores de tais necessidades, visto que se aplica a quaisquer atores envolvidos

¹Disponível em: <www.computer.org/portal/web/swebok>. Acesso em: 21 jan. 2013.

²Disponível em: <www.abntcatalogo.com.br/norma.aspx?ID=2815>. Acesso em: 21 jan. 2013.

TABELA 11.1 Fases genéricas do ciclo de vida da Norma ISO/IEC 15288 e suas equivalentes SQuaRE

| Fases da 15288 | Agrupamento de fases de SQuaRE |
|--|--|
| Processo de definição de requisitos dos interessados | Requisitos de qualidade do produto de software |
| Processo de análise dos requisitos | Desenvolvimento do produto |
| Processo de <i>design</i> arquitetural | |
| Processo de implementação | |
| Processo de integração | |
| Processo de verificação | |
| Processo de transição | |
| Processo de validação | |
| Processo de operação | Uso do produto |
| Processo de manutenção | |
| Processo de aposentadoria | - |

com a produção, encomenda, uso ou pessoas afetadas pelas consequências do software ou de seu processo de produção.

Essa norma, referência de atributos de qualidade por vários anos, foi substituída em julho de 2011 pela ISO/IEC 25010, parte da nova família de Normas ISO/IEC 25000: *Software Engineering: Software Product Quality Requirements and Evaluation (SQuaRE)*³.

Uma das motivações para a criação de uma nova norma está no fato de que as antigas aplicavam-se apenas ao processo de desenvolvimento e uso do produto de software, mas pouco tinham a dizer em relação à definição do produto.

Para a definição de uma segunda geração de normas de qualidade, foi então utilizado um modelo genérico de processo de desenvolvimento baseado na Norma ISO/IEC 15288 – *System Life Cycle Processes*⁴ (Tabela 11.1).

O lado esquerdo da tabela apresenta as fases originais da Norma 15288, e o lado direito apresenta o agrupamento dessas fases para efeito da aplicação das normas de qualidade SQuaRE. Apenas o processo de aposentadoria de software ainda não é contemplado pelo novo conjunto de normas.

O modelo de qualidade SQuaRE avalia quatro tipos de indicadores de qualidade:

- a) *Medidas de qualidade do processo*: avaliam a qualidade do processo usado para desenvolver os produtos e, consequentemente, a maturidade da empresa em termos de processos de engenharia de software. Mais detalhes sobre qualidade de processo podem ser encontrados no Capítulo 12.
- b) *Medidas de qualidade internas*: avaliam aspectos internos da qualidade do software que normalmente só são percebidos pelos desenvolvedores (por exemplo, capacidade de manutenção).
- c) *Medidas de qualidade externas*: avaliam aspectos externos da qualidade do software que podem ser avaliados pela equipe de desenvolvimento do ponto de vista do usuário (por exemplo, eficiência).
- d) *Medidas de qualidade do software em uso*: avaliam aspectos da qualidade do software em seu ambiente final que só podem ser medidos pelos usuários finais (por exemplo, satisfação dos usuários).

Assim, as *qualidades internas* permitem que a equipe de desenvolvimento atinja seus objetivos de forma eficiente. As *qualidades externas* e *de uso* permitem que o usuário final do sistema atinja seus objetivos. As qualidades internas, portanto, nem sempre são importantes para o usuário final, pois ele não as percebe diretamente, mas pode ser afetado indiretamente por elas (no tempo de manutenção ou evolução do software, por exemplo).

Defende-se que a qualidade de processo influencia a qualidade interna, que influencia a qualidade externa, que, por sua vez, influencia a qualidade de uso do software (Figura 11.1).

³Disponível em: <www.iso.org/iso/catalogue_detail.htm?csnumber=35683>. Acesso em: 21 jan. 2013.

⁴Disponível em: <www.iso.org/iso/catalogue_detail?csnumber=43564>. Acesso em: 21 jan. 2013.

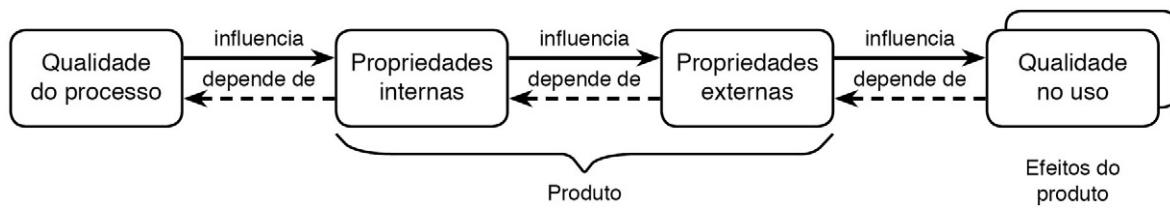


Figura 11.1 Abordagem conceitual para qualidade de acordo com a ISO/IEC 25010:2011.

Na figura, o bloco “Qualidade no uso” é mostrado como um bloco múltiplo, já que, em diferentes contextos de uso, o mesmo produto pode ter diferentes avaliações de sua qualidade no uso.

O modelo SQuaRE (Suryn & Abran, 2003)⁵ é formado por um conjunto de normas dividido da seguinte forma:

- ISO/IEC 2500n – Divisão Gestão da Qualidade.
- ISO/IEC 2501n – Divisão Modelo de Qualidade.
- ISO/IEC 2502n – Divisão Medição da Qualidade.
- ISO/IEC 2503n – Divisão Requisitos de Qualidade.
- ISO/IEC 2504n – Divisão Avaliação da Qualidade.

A divisão de *gestão da qualidade* do modelo SQuaRE apresenta os modelos comuns, padrões básicos, termos e definições usados por toda a série de normas SQuaRE. Essa divisão inclui duas unidades:

- Guia do SQuaRE*, que apresenta a estrutura, terminologia, visão geral do documento, público-alvo, modelos de referência e partes associadas da série.
- Planejamento e Gerenciamento*, que apresenta os requisitos para planejar e gerenciar o processo de avaliação da qualidade de produtos de software.

O *modelo de qualidade* apresenta as características e subcaracterísticas de qualidade interna, externa e de uso. Os padrões na área de medidas de qualidade são derivados das Normas 9126 e 14598, cobrindo as definições matemáticas e o detalhamento da aplicação de medidas práticas de qualidade interna, externa e de uso. O documento inclui o seguinte:

- Modelo de referência e guia de medição*: apresenta uma introdução e explicação sobre a aplicação das medidas de qualidade.
- Medidas primitivas*: conjunto de medições básicas usadas para a definição das demais.
- Medidas internas*: conjunto de medidas quantitativas em termos de características e subcaracterísticas internas.
- Medidas externas*: conjunto de medidas quantitativas em termos de características e subcaracterísticas externas.
- Medidas de uso*: conjunto de medidas quantitativas em termos de características e subcaracterísticas de uso do software.

A divisão de *requisitos de qualidade* contém o padrão para suportar a especificação de requisitos de qualidade, tanto para a fase de elicitação dos requisitos de qualidade do software quanto como entrada para o processo de avaliação da qualidade do software.

A divisão de *avaliação da qualidade* provê as ferramentas para a avaliação da qualidade de um sistema de software tanto por desenvolvedores, compradores ou avaliadores independentes. Os seguintes documentos são disponibilizados:

- Guia e visão geral da avaliação da qualidade*: apresenta um *framework* para a avaliação da qualidade de um produto de software.
- Processo para desenvolvedores*: recomendações práticas para a avaliação da qualidade de um produto quando esta é feita paralelamente ao seu desenvolvimento.

⁵Disponível em: <profs.etsmtl.ca/wsurn/research/SQE-Publ/SQuaRE-second%20generation%20of%20standards%20for%20SW%20Quality%20%28IASTED03%29.pdf>. Acesso em: 21 jan. 2013.

- c) *Processo para compradores*: recomendações práticas para a avaliação de produtos comprados em prateleira (COTS), feitos por encomenda, ou ainda para avaliação de modificações em sistemas existentes.
- d) *Processo para avaliadores*: recomendações práticas para a avaliação do software por terceiros, enfatizando a participação de vários agentes que precisam compreender e aceitar essa avaliação.
- e) *Documentação para o módulo de avaliação*: define a estrutura e o conteúdo da documentação usada no processo de avaliação.

O modelo de qualidade da ISO/IEC 25010 define um conjunto de oito características internas e externas de produto de software, subdivididas em subcaracterísticas e mais cinco características de software *em uso*, algumas das quais também são subdivididas em subcaracterísticas.

O modelo resultante é mostrado na Tabela 11.2. As características internas e externas do software são agregadas nas chamadas *características do produto*, pois podem ser avaliadas no ambiente de desenvolvimento. Já as características do software *em uso* só podem ser avaliadas no contexto de uso do sistema. As características e subcaracterísticas da tabela são apresentadas em português e em inglês, pois a tradução pode nem sempre apresentar o espírito do termo na sua língua original.

O conjunto de características e subcaracterísticas dessa norma modificou-se bastante ao longo do tempo, enquanto ela era elaborada. Assim, é possível encontrar versões diferentes de características e subcaracterísticas na literatura, pois estas podem ter se baseado em versões intermediárias da norma. A lista apresentada aqui corresponde à versão definitiva, publicada em 2011⁶. A seguir, as características do modelo de qualidade são brevemente apresentadas.

11.1.1 ADEQUAÇÃO FUNCIONAL

A *adequação funcional* mede o grau em que o produto disponibiliza funções que satisfazem às necessidades estabelecidas e implicadas quando o produto é usado sob condições especificadas. Suas subcaracterísticas são:

- a) *Completude funcional*: o software efetivamente possibilita executar as funções que são apropriadas, ou seja, as entradas e saídas de dados necessárias para o usuário atingir seus objetivos são possíveis? Um software no qual faltem algumas funções necessárias não apresenta a qualidade de completude funcional.
- b) *Corretude funcional*: também denominada *acurácia*, essa subcaracterística avalia o quanto o software gera dados e consultas corretos e precisos de acordo com sua definição. Um software que apresenta dados incorretos ou com grau de imprecisão acima de um limite definido como tolerável não apresenta a qualidade de corretude funcional.
- c) *Funcionalidade apropriada*: essa subcaracterística indica em qual grau as funções do sistema facilitam a realização de tarefas e objetivos para os quais o sistema foi especificado.

11.1.2 CONFIABILIDADE

Um software *confiável* é aquele que, ao longo do tempo, se mantém com um comportamento consistente com o esperado. A confiabilidade tem relação com a minimização da quantidade de defeitos do software e com a forma como ele funciona perante situações anômalas. As subcaracterísticas da confiabilidade são:

- a) *Maturidade*: a maturidade é a medida da frequência com que um software apresenta defeitos. Um software mais maduro é aquele que apresenta menos defeitos ao longo de um período fixo de tempo. Espera-se que a maturidade de um sistema aumente com o tempo, mas processos de manutenção mal gerenciados, especialmente aqueles que deixam de realizar testes de regressão (Seção 13.2.6) ou refatoração, podem fazer que a maturidade de um sistema diminua com o passar do tempo, ou seja, em vez de reduzir a frequência dos erros, eles podem aumentar.
- b) *Disponibilidade*: essa subcaracterística avalia o quanto o software está operacional e disponível para uso quando necessário.
- c) *Tolerância a falhas*: a subcaracterística de tolerância a falhas tem relação com a forma como o software reage quando em situação anômala. Idealmente, requisitos de tolerância a falhas deveriam ser definidos durante o projeto do software. Um software que consegue continuar funcionando mesmo quando ocorrem

⁶Disponível em: <www.meti.go.jp/policy/it_policy/softseibi/metrics/20110324product_metrics2010%28en%29.pdf>. Acesso em: 21 jan. 2013.

TABELA 11.2 Modelo de qualidade da ISO 25010:2011

| Tipo | Características | Subcaracterísticas | Termo em inglês |
|----------------------------|--|---|---|
| Características do produto | Adequação funcional (<i>functional suitability</i>) | Completude funcional Corretude funcional (acurácia) Funcionalidade apropriada | <i>Functional completeness</i> <i>Functional correctness (accuracy)</i> <i>Functional appropriateness</i> |
| | Confiabilidade (<i>reliability</i>) | Maturidade Disponibilidade Tolerância a falhas Recuperabilidade | <i>Maturity</i> <i>Availability</i> <i>Fault tolerance</i> <i>Recoverability</i> |
| | Usabilidade (<i>usability</i>) | Apropriação reconhecível Inteligibilidade Operabilidade Proteção contra erro de usuário Estética de interface com usuário Acessibilidade | <i>Appropriateness recognisability</i> <i>Learnability</i> <i>Operability</i> <i>User error protection</i> <i>User interface aesthetics</i> <i>Accessibility</i> |
| | Eficiência de desempenho (<i>performance efficiency</i>) | Comportamento em relação ao tempo Utilização de recursos Capacidade | <i>Time behavior</i> <i>Resource utilization</i> <i>Capacity</i> |
| | Segurança (<i>security</i>) | Confidencialidade Integridade Não repúdio Rastreabilidade de uso Autenticidade | <i>Confidentiality</i> <i>Integrity</i> <i>Non-repudiation</i> <i>Accountability</i> <i>Authenticity</i> |
| | Compatibilidade (<i>compatibility</i>) | Coexistência Interoperabilidade | <i>Co-existence</i> <i>Interoperability</i> |
| | Capacidade de manutenção (<i>maintainability</i>) | Modularidade Reusabilidade Analizabilidade Modificabilidade Testabilidade | <i>Modularity</i> <i>Reusability</i> <i>Analyzability</i> <i>Modifiability</i> <i>Testability</i> |
| | Portabilidade (<i>portability</i>) | Adaptabilidade Instalabilidade Substituibilidade | <i>Adaptability</i> <i>Instalability</i> <i>Replaceability</i> |
| Características do uso | Efetividade (<i>effectiveness</i>) | Efetividade | <i>Effectiveness</i> |
| | Eficiência (<i>efficiency</i>) | Eficiência | <i>Efficiency</i> |
| | Satisfação (<i>satisfaction</i>) | Utilidade Prazer Conforto Confiança | <i>Usefulness</i> <i>Pleasure</i> <i>Comfort</i> <i>Trust</i> |
| | Uso sem riscos (<i>freedom from risk</i>) | Mitigação de risco econômico Mitigação de risco a saúde e segurança Mitigação de risco ambiental | <i>Economic risk mitigation</i> <i>Health and safety risk mitigation</i> <i>Environmental risk mitigation</i> |
| | Cobertura de contexto (<i>context coverage</i>) | Completude de contexto Flexibilidade | <i>Context completeness</i> <i>Flexibility</i> |

falhas tem boa avaliação em relação a essa qualidade. Deve-se deixar claro, porém, que falhas e defeitos são coisas diferentes. A maturidade tem a ver com a minimização de *erros* que são oriundos de *defeitos* e, portanto, indesejáveis em qualquer sistema. Uma *falha*, porém, é uma situação que pode ocorrer mesmo que o software não apresente nenhum defeito. Por exemplo, uma linha de comunicação pode ser temporariamente interrompida, um dispositivo de armazenamento pode ser danificado, um processador pode estar danificado etc. Essas falhas são imprevisíveis e, na maioria das vezes, inevitáveis. A qualidade de tolerância a falhas, então, tem relação com a maneira como o software reage a essas situações externas indesejadas.

- d) *Recuperabilidade*: a recuperabilidade de um sistema está relacionada à sua capacidade de recuperar dados e colocar-se novamente em operação após uma situação de desastre. Idealmente, deveria haver requisitos de recuperabilidade definidos para a maioria dos projetos de software, pois situações negativas, como uma falha generalizada ou perda de dados, nem sempre são previstas em um projeto, a não ser que sejam explicitamente recomendadas. Em relação à tolerância a falhas, pode-se dizer que a recuperabilidade trata de situações mais críticas em que o problema ocorrido não é apenas temporário, como uma falha de comunicação, ele é mais definitivo, como a perda completa de um disco de dados.

11.1.3 USABILIDADE

A *usabilidade* avalia o grau no qual o produto tem atributos que permitem que seja entendido, aprendido, usado e que seja atraente ao usuário, quando usado sob condições especificadas. Suas subcaracterísticas são:

- a) *Apropriação reconhecível*: mede o grau em que os usuários reconhecem que o produto é apropriado para suas necessidades.
- b) *Inteligibilidade*: tem relação com o grau de facilidade que um usuário tem para entender os conceitos-chave do software e, assim, tornar-se competente no seu uso.
- c) *Operabilidade*: avalia o grau no qual o produto é fácil de usar e controlar.
- d) *Proteção contra erro de usuário*: avalia o grau em que o produto foi projetado para evitar que o usuário cometa erros.
- e) *Estética de interface com usuário*: avalia o grau em que a interface com o usuário proporciona prazer e uma interação satisfatória.
- f) *Acessibilidade*: avalia o grau em que o produto foi projetado para atender a usuários com necessidades especiais.

11.1.4 EFICIÊNCIA DE DESEMPENHO

A *eficiência de desempenho* trata da otimização do uso de recursos de tempo e espaço. Espera-se que um sistema seja o mais eficiente possível de acordo com o tipo de problema que ele soluciona.

Existem problemas para os quais as soluções computacionais demandam quantidades de tempo e memória que as tornam intratáveis. Nesses problemas, a eficiência de tempo pode ser conseguida com sacrifício, por exemplo, da acurácia, como no caso de algoritmos que buscam soluções aproximadas para problemas intratáveis em tempo razoável.

A característica de eficiência de desempenho divide-se, então, em duas subcaracterísticas:

- a) *Comportamento em relação ao tempo*: essa qualidade mede o tempo que o software leva para processar suas funções. Existe para todos os problemas algorítmicos um limite mínimo de complexidade que pode ser demonstrado formalmente. Um sistema eficiente em termos de tempo, portanto, é aquele cujo tempo de processamento se aproxima desse mínimo.
- b) *Utilização de recursos*: normalmente associada a espaço de armazenamento ou memória, a eficiência de recursos também pode ser associada a outros recursos necessários, como, por exemplo, banda de transmissão de rede. Em algumas situações pode-se obter eficiência de tempo com sacrifício da eficiência de recursos e vice-versa.
- c) *Capacidade*: avalia o grau em que os limites máximos do produto atendem aos requisitos. A *capacidade* de um produto, portanto, é relativa aos seus requisitos. Por exemplo, pode-se avaliar que um produto capaz de processar 20 transações simultaneamente terá excelente capacidade se os requisitos exigirem 2 ou 3 transações simultâneas, mas o mesmo produto terá capacidade ruim caso os requisitos exijam 40 ou 50 transações simultâneas.

11.1.5 SEGURANÇA

A característica de *segurança* avalia o grau em que as funções e os dados são protegidos de acesso não autorizado e o grau em que são disponibilizados para acesso autorizado. Deve-se tomar cuidado para não confundir a qualidade de *segurança* (*security*) relacionada à segurança dos dados e funções com a qualidade de *uso seguro* (*safety*), que é uma qualidade do software em uso, relacionada à segurança das pessoas, instalações e meio ambiente.

As subcaracterísticas de segurança são:

- a) *Confidencialidade*: avalia o grau em que as informações e funções do sistema são acessíveis por quem tem a devida autorização para isso.
- b) *Integridade*: avalia o grau em que os dados e funções do sistema são protegidos contra acesso por pessoas ou sistemas não autorizados.
- c) *Não repúdio*: avalia o grau em que o sistema permite constatar que ações ou acessos foram efetivamente feitos, de forma que não possam ser negados posteriormente.
- d) *Rastreabilidade de uso*: avalia o grau em que as ações realizadas por uma pessoa ou sistema podem ser rastreadas de forma a comprovar que foram efetivamente realizadas por essa pessoa ou sistema.
- e) *Autenticidade*: avalia o grau em que a identidade de uma pessoa ou recurso é efetivamente aquela que se diz ser. Por exemplo, não é desejável que um usuário possa se passar por outro ou que um recurso (conjunto de dados) que se pensa ser uma coisa na verdade seja outra.

A diferença entre confidencialidade e integridade é sutil. No primeiro caso, garante-se que quem deve ter acesso o terá; no segundo, garante-se que quem não deve ter acesso não o terá.

11.1.6 COMPATIBILIDADE

A *compatibilidade* avalia o grau em que dois ou mais sistemas ou componentes podem trocar informação e/ou realizar suas funções requeridas enquanto compartilham o mesmo ambiente de hardware e software. Suas subcaracterísticas são:

- a) *Coexistência*: avalia o grau no qual o produto pode desempenhar as funções requeridas eficientemente enquanto compartilha ambiente e recursos comuns com outros produtos, sem impacto negativo nos demais produtos.
- b) *Interoperabilidade*: avalia o grau no qual o software é capaz de interagir com outros sistemas com os quais se espera que ele interaja. Um software incapaz de se comunicar adequadamente com outros sistemas-chave não apresenta a qualidade de interoperabilidade.

11.1.7 CAPACIDADE DE MANUTENÇÃO

A *capacidade de manutenção* é uma característica interna do software percebida diretamente apenas pelos desenvolvedores, embora os clientes possam ser afetados por ela na medida do tempo gasto pelos desenvolvedores para executar atividades de manutenção ou evolução do software. A manutenibilidade, portanto, mede a facilidade de se realizarem alterações no software para sua evolução ou de detectar e corrigir erros.

A capacidade de manutenção se subdivide nas seguintes subcaracterísticas:

- a) *Modularidade*: avalia o grau em que o sistema é subdividido em partes lógicas coesas, de forma que mudanças em uma dessas partes tenham impacto mínimo nas outras.
- b) *Reusabilidade*: avalia o grau em que partes do sistema podem ser usadas para construir outros sistemas.
- c) *Analysabilidade*: um sistema é analisável quando permite encontrar defeitos (depurar) facilmente quando ocorrem erros ou falhas. Por exemplo, sistemas que, quando falham, travam o computador podem ter nível de analysabilidade baixo, porque é difícil encontrar um defeito em um sistema travado. Já sistemas que, ao falharem, apresentam mensagens relacionadas com exceções internas ocorridas são mais facilmente analisáveis.
- d) *Modificabilidade*: a modificabilidade tem relação com a facilidade que o sistema oferece para que erros sejam corrigidos quando detectados, sem que as modificações introduzam novos defeitos ou degradem sua organização interna. Boas práticas de programação, arquitetura bem definida, refatoração quando necessário, aplicação de padrões de projeto e padrões de programação e testes automatizados são exemplos de disciplinas que podem colaborar para que sistemas tenham melhor modificabilidade.
- e) *Testabilidade*: a testabilidade mede a facilidade de realizar testes de regressão (Seção 13.2.6). Essa qualidade não diz tanto respeito ao software em si quanto ao processo estabelecido para permitir que ele seja testado. Porém, algumas características internas do software, como a complexidade ciclomática ou a coesão modular, podem afetar significativamente a testabilidade.

11.1.8 PORTABILIDADE

A *portabilidade* avalia o grau em que o software pode ser efetiva e eficientemente transferido de um ambiente de hardware ou software para outro. Suas subcaracterísticas são:

- a) *Adaptabilidade*: avalia o quanto é fácil adaptar o software a outros ambientes sem que seja necessário aplicar ações ou meios além daqueles fornecidos com o próprio software.
- b) *Instalabilidade*: avalia a facilidade de instalar o software.
- c) *Substituibilidade*: avalia o grau em que o sistema pode substituir outro no mesmo ambiente e com os mesmos objetivos.

11.1.9 QUALIDADES DO SOFTWARE EM USO

As características de qualidade do software em uso são fatores externos que só podem ser plenamente avaliados quando o software está efetivamente em seu ambiente de uso final, ou seja, é muito difícil avaliá-las em um ambiente de desenvolvimento. As características são as seguintes:

- a) *Efetividade*: é a capacidade que o produto de software tem para fazer que o cliente atinja seus objetivos de negócio de forma correta e completa, no ambiente real de uso.
- b) *Eficiência*: avalia o retorno que o produto dá ao cliente, ou seja, a razão entre o que o cliente investiu e investe no sistema em relação ao que recebe em troca. Essa medida nem sempre é financeira.
- c) *Satisfação*: é a capacidade do produto de satisfazer aos usuários durante seu uso no ambiente final. Essa característica subdivide-se nas seguintes subcaracterísticas:
 - *Utilidade*: avalia o grau no qual o usuário é satisfeito com a obtenção percebida de metas pragmáticas, incluindo os resultados e as consequências do uso do software.
 - *Prazer*: avalia o grau em que o usuário sente prazer em usar o sistema para satisfazer seus objetivos.
 - *Conforto*: avalia o conforto físico e mental do usuário ao usar o sistema.
 - *Confiança*: avalia o grau em que o usuário ou outros interessados confiam que o sistema faça o que é esperado dele.
- d) *Uso sem riscos*: é a capacidade do produto de estar dentro de níveis aceitáveis de segurança relativa a riscos envolvendo pessoas, negócios e meio ambiente. Essa característica subdivide-se em:
 - *Mitigação de risco econômico*: avalia o grau no qual o produto minimiza riscos financeiros potenciais, incluindo danos à propriedade e à reputação de pessoas.
 - *Mitigação de risco à saúde e à segurança*: avalia o grau no qual o produto minimiza riscos físicos às pessoas em seu contexto de uso.
 - *Mitigação de risco ambiental*: avalia o grau no qual o produto minimiza riscos ambientais ou à propriedade em seu contexto de uso.
- e) *Cobertura de contexto*: avalia o grau no qual o produto ou sistema pode ser usado com efetividade, eficiência, sem riscos e com satisfação tanto no contexto inicialmente especificado quanto em outros contextos. Essa característica subdivide-se em:
 - *Completure de contexto*: avalia o grau no qual o produto ou sistema pode ser usado com efetividade, eficiência, sem riscos e com satisfação em todos os contextos de uso especificados.
 - *Flexibilidade*: avalia o grau no qual o produto ou sistema pode ser usado com efetividade, eficiência, sem riscos e com satisfação em contextos diferentes daqueles inicialmente especificados.

11.2 Modelo de Qualidade de Dromey

Modelos de qualidade são estruturas conceituais que definem quais são as características da qualidade e como elas se estruturam e se relacionam entre si. A Norma 9126 e sua sucessora, a 25010, são exemplos importantes de modelos de qualidade nos quais as características são decompostas hierarquicamente.

Uma das críticas a esse tipo de modelo é que não há um critério claro para decidir quais são as características de alto nível e quais são as subcaracterísticas (Kitchenham & Lawrence, 1996). Assim, Dromey (1998)⁷

⁷Disponível em: <se.dongguk.ac.kr/metrics/SPQ-Theory.pdf>. Acesso em: 21 jan. 2013.

desenvolveu um modelo que procura resolver esse e outros problemas apontados em relação aos modelos hierárquicos. Ele diz que é impossível construir características de qualidade de alto nível (como usabilidade) diretamente nos produtos. Em vez disso, é necessário construir componentes de software que exibam um conjunto harmonioso, completo e consistente de propriedades que resultem na manifestação dessas características de alto nível.

Assim, antes de criar uma hierarquia de características de qualidade, Dromey estabelece um método para determinar sistematicamente essas características, a partir do qual é possível avaliar se a decomposição é consistente e completa. O método se baseia nos seguintes princípios:

- a) Um *comportamento* pode ser decomposto, e assim definido, em termos de propriedades subordinadas, as quais podem ser tanto comportamentos quanto características do software.
- b) Um *uso* pode ser decomposto, e assim definido, em termos de propriedades subordinadas, que podem ser tanto usos quanto características do software.

O desafio, então, passa a ser diferenciar os comportamentos e usos subordinados hierarquicamente (ou seja, que são subtipos do comportamento ou usos de mais alto nível) das características de software, que não são subtipos, mas elementos agregados aos comportamentos ou usos.

Algumas propriedades, como modularidade e acurácia, são claramente características do software, enquanto outras, como tolerância a falhas, não são tão facilmente distinguíveis de comportamentos.

A sugestão de Dromey é que se considerem os *comportamentos* como gerais em relação ao sistema, ou seja, não sendo aplicáveis apenas a algumas de suas partes. Enquanto isso, as *características* de software podem ser aplicadas a componentes específicos.

A construção de um modelo de qualidade deve ainda considerar os seguintes princípios:

- a) Propriedades abstratas, chamadas de *atributos de qualidade*, podem ser associadas ao software.
- b) A *qualidade do software* pode ser caracterizada por um conjunto de atributos de qualidade de alto nível.
- c) Os *atributos de qualidade do software* correspondem tanto a um conjunto de comportamentos de software independentes de domínio quanto a um conjunto de usos de software independentes de domínio.
- d) Os *atributos de qualidade de um modelo de qualidade* devem ser suficientes para satisfazer às necessidades de todos os grupos de interesse associados ao software.
- e) Cada *atributo de qualidade de software de alto nível* é caracterizado por um conjunto de propriedades subordinadas, que podem ser tanto comportamentos quanto usos ou características do software.
- f) Cada *característica* do software é determinada ou composta por um conjunto de propriedades tangíveis, que serão chamadas de *propriedades que transferem qualidade (quality carrying)*.
- g) *Propriedades que transferem qualidade* podem incorporar tanto propriedades funcionais quanto não funcionais.

A Figura 11.2 apresenta, de forma esquemática, a composição do modelo de qualidade de Dromey, que, na verdade, pode ser considerado um metamodelo, já que modelos individuais podem ser instanciados a partir dele.

Dromey apresenta dois estudos de caso de aplicação desse modelo, um para a definição de *usabilidade* e outro para a definição de *confiabilidade*. No caso de usabilidade, a título de ilustração, ele começa definindo um conjunto de propriedades (não necessariamente completo) que podem caracterizar a usabilidade:

- a) *Apreensibilidade*: facilidade de aprender como usar.
- b) *Transparência*: facilidade de entender ou lembrar como se usa uma funcionalidade.
- c) *Operabilidade*: facilidade de aplicar uma funcionalidade eficientemente.
- d) *Responsividade*: capacidade de executar todas as funções em tempo razoável.
- e) *Customizabilidade*: capacidade de personalizar interfaces para as necessidades do usuário.
- f) *Disponibilização de outros idiomas*: capacidade de mudar a língua da interface.
- g) *Comandos sensíveis ao contexto*: mostrar comandos usados no contexto.
- h) *Imediatismo operacional*: permitir a execução de comandos de forma direta.
- i) *Teclas de atalho*: permite uso de teclas de atalho para as opções mais frequentes.
- j) *Consistência*: comandos consistentes com o ambiente.

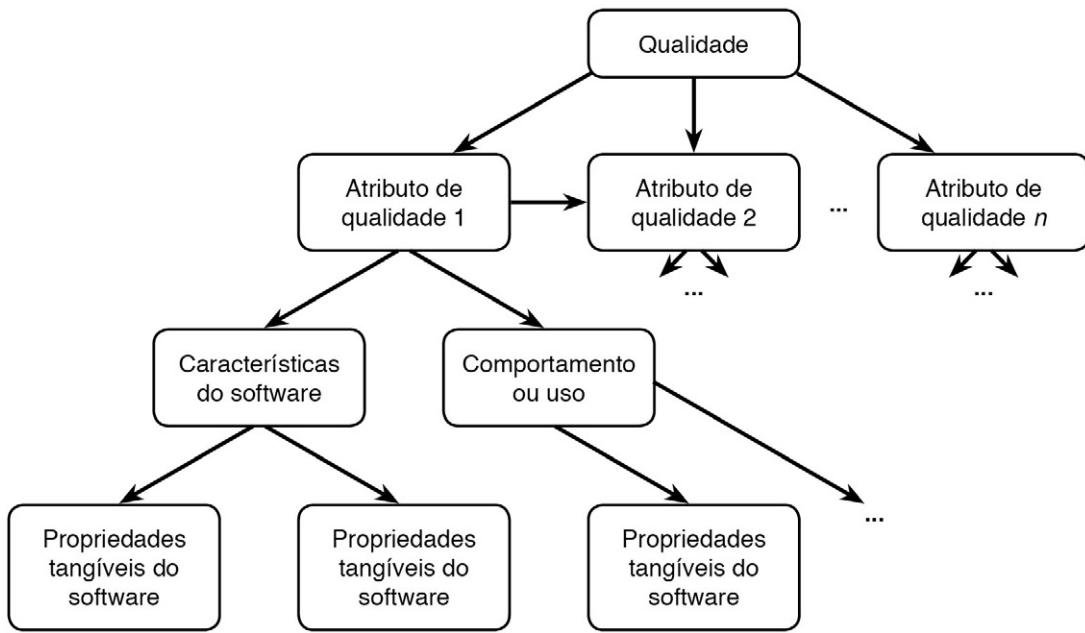


Figura 11.2 Modelo de qualidade de Dromey.

Inicialmente, essas características são classificadas em usos, comportamentos ou características do software, resultando no seguinte:

- Apreensibilidade: uso.
- Transparência: característica do software.
- Operabilidade: uso.
- Responsividade: comportamento (métrica).
- Customizabilidade: uso.
- Disponibilização de outros idiomas: característica do software.
- Comandos sensíveis ao contexto: característica do software.
- Imediatismo operacional: característica do software.
- Teclas de atalho: característica do software.
- Consistência: característica do software.

Comportamentos e usos são candidatos a serem determinantes de usabilidade de nível mais alto, embora alguns comportamentos ou usos possam ser subordinados a outros. O processo de definição da hierarquia consiste em perguntar, para cada duas propriedades, se uma é parte ou subtipo de outra. Dessa forma, uma hierarquia como a seguinte poderia ser formada a partir das características mostradas no exemplo:

- Usabilidade
 - Apreensibilidade
 - i. Consistência
 - ii. Transparência
 - iii. Comandos sensíveis ao contexto
 - Operabilidade
 - i. Customizabilidade
 - 1. Disponibilização de outros idiomas
 - ii. Responsividade
 - iii. Imediatismo operacional
 - iv. Teclas de atalho

O modelo é também fundamentado no axioma de que, se o software é composto por componentes, então suas propriedades contextuais intrínsecas tangíveis e a forma como eles são compostos vão determinar a qualidade do

TABELA 11.3 Propriedades de variáveis que impactam na qualidade de software

| Propriedade que transfere qualidade | Característica do software | Impacto na qualidade |
|-------------------------------------|----------------------------|--|
| Convenção de nomes | Analisabilidade | Capacidade de manutenção |
| Atribuída | Correção | Funcionalidade e confiabilidade |
| Precisa | Correção | Funcionalidade e confiabilidade |
| Propósito único | Correção | Funcionalidade e confiabilidade |
| Encapsulada | Modularidade | Capacidade de manutenção e reusabilidade |
| Utilizada | Consistência | Capacidade de manutenção |
| Autodescritiva | Analisabilidade | Capacidade de manutenção e reusabilidade |
| Comentada | Analisabilidade | Capacidade de manutenção e reusabilidade |

TABELA 11.4 Propriedades de expressões que impactam na qualidade de software

| Propriedade que transfere qualidade | Característica do software | Impacto na qualidade |
|-------------------------------------|----------------------------|----------------------------------|
| Computável | Correção | Funcionalidade e confiabilidade |
| Livre de efeitos colaterais | Correção | Funcionalidade e confiabilidade |
| Efetiva | Não redundância | Eficiência |
| Ajustável | Parametrização | Manutenibilidade e reusabilidade |

TABELA 11.5 Propriedades de comandos que impactam na qualidade de software

| Propriedade que transfere qualidade | Característica do software | Impacto na qualidade |
|-------------------------------------|----------------------------|----------------------|
| Completo | Corretude | Funcionalidade |
| Efetivo | Não redundância | Eficiência |

software. Ou seja, a qualidade do software começa na escolha dos nomes das variáveis, por exemplo, que são usadas para criar os métodos usados para criar as classes etc.

A Tabela 11.3 mostra algumas propriedades de variáveis que, por transferirem qualidade para certas características do software, têm impacto na qualidade. Essa tabela pode ser utilizada como um *checklist* para uma eventual inspeção de qualidade no código-fonte. Por exemplo, uma variável definida mas nunca atribuída é um potencial defeito no software que precisa ser corrigido.

Na sequência, as variáveis são compostas para criar expressões no software. Algumas propriedades de expressões podem definir qualidade e ter impacto. A Tabela 11.4 apresenta algumas qualidades que podem ser atribuídas a expressões (linhas de código) que afetam a qualidade do software.

No nível seguinte estão os comandos como atribuições e estruturas de controle, as quais usam expressões. As qualidades são identificadas na Tabela 11.5.

As comunicações com o hardware também são analisadas e algumas propriedades, indicadas na Tabela 11.6.

A análise das propriedades que transferem qualidade ligadas à conexão entre o software e a base de dados é apresentada na Tabela 11.7.

As propriedades referentes à comunicação de dados são apresentadas na Tabela 11.8.

Mais detalhes podem ser encontrados no trabalho de Dromey (1998)⁸.

⁸Disponível em: <se.dongguk.ac.kr/metrics/SPQ-Theory.pdf>. Acesso em: 21 jan. 2013.

TABELA 11.6 Propriedades de interfaces com hardware que impactam na qualidade de software

| Propriedade que transfere qualidade | Característica do software | Impacto na qualidade |
|---|-------------------------------------|---|
| Inicialização no <i>startup</i> , reinício ou recuperação de erro | Sobrevivência | Confiabilidade |
| Checagem periódica de <i>status</i> operacional | Sobrevivência e tolerância a falhas | Confiabilidade |
| Checagem periódica de <i>status</i> operacional parametrizada | Flexibilidade | Capacidade de manutenção |
| Relatório e resolução quando o dispositivo não está operacional | Visibilidade e sobrevivência | Funcionalidade e confiabilidade |
| Intervalo de valores permitidos do dispositivo é documentado | Tolerância a falhas | Confiabilidade |
| Todos os estados operacionais do dispositivo são resolvidos e completos | Corretude e tolerância a falhas | Funcionalidade e confiabilidade |
| Todas as interações para cada dispositivo de hardware estão encapsuladas e parametrizadas | Modularidade e flexibilidade | Capacidade de manutenção, reúso e portabilidade |
| Detalhes arquiteturais de dispositivos estão encapsulados e parametrizados | Modularidade e flexibilidade | Capacidade de manutenção, reúso e portabilidade |

TABELA 11.7 Propriedades de interfaces com a base de dados que impactam na qualidade de software

| Propriedade que transfere qualidade | Característica do software | Impacto na qualidade |
|--|---|--|
| Chamadas à base de dados não devem depender de conhecimento sobre seu esquema de gerenciamento | Independência de aplicação | Capacidade de manutenção, portabilidade e reusabilidade |
| Chamadas à base de dados devem ser processadas fora da unidade que faz a chamada | Independência de aplicação | Capacidade de manutenção, portabilidade e reusabilidade |
| A aplicação deve ser independente de detalhes da estrutura da base de dados | Independência de representação e independência de sistema | Capacidade de manutenção, portabilidade e reusabilidade |
| O <i>design</i> de gerenciamento da base de dados deve fornecer uma forma comum e controlada para adicionar, recuperar e alterar dados | Consistência, interoperabilidade e segurança | Capacidade de manutenção, reusabilidade e funcionalidade |
| Itens da base de dados não devem ser referenciados por mais de um nome | Consistência e eficiência de armazenamento | Capacidade de manutenção, eficiência e confiabilidade |
| Modificações na base de dados devem ser feitas por transações atômicas, consistentes, íntegras e duráveis | Tolerância a falhas e consistência | Confiabilidade e capacidade de manutenção |
| A integridade da base de dados deve ser recuperada a partir de condições de erro | Tolerância a falhas | Confiabilidade |
| O acesso à base de dados é controlado | Acessibilidade e segurança | Funcionalidade |
| O <i>design</i> do sistema o protege contra acesso não autorizado | Acessibilidade e segurança | Funcionalidade |

TABELA 11.8 Propriedades de comunicação de dados que impactam na qualidade de software

| Propriedade que transfere qualidade | Característica do software | Impacto na qualidade |
|--|---------------------------------------|---|
| Mensagens contêm rótulos indicando o tipo de dados | Interoperabilidade | Funcionalidade |
| Intervalos válidos para todos os dados nas mensagens são indicados | Tolerância a falhas e analisabilidade | Confiabilidade e capacidade de manutenção |
| Propósito e formato de cada item de dados nas mensagens são definidos | Analysabilidade | Capacidade de manutenção |
| Glossário técnico para rótulos de mensagens é fornecido | Analysabilidade | Capacidade de manutenção |
| Formato comum especificado é usado para posicionamento, empacotamento de dados e transmissão de blocos | Interoperabilidade e analysabilidade | Capacidade de manutenção e funcionalidade |
| Representações de dados em mensagens atendem a padrões especificados em contrato | Interoperabilidade e analysabilidade | Capacidade de manutenção e funcionalidade |

11.3 Instalação de um Programa de Melhoria de Qualidade

Para que a gestão da qualidade seja eficaz, deve existir um *programa de melhoria de qualidade* definido na empresa. A instalação inicial do programa implica estabelecer uma anistia geral na empresa (Belchior, 1997), ou seja, não se buscam culpados para o que aconteceu até o momento. Busca-se promover uma melhoria geral conjunta.

Para que o programa de melhoria de qualidade funcione, é necessário que ele seja acordado e conhecido por todos os envolvidos e, também, que se torne parte da cultura da empresa. Planos apenas colocados no papel que são abandonados na primeira dificuldade logo são esquecidos.

Assim, os planos devem ser consistentes, factíveis, gradualmente implementados e, principalmente, levados a sério por todos. Planos inconsistentes, impossíveis de executar e que caem do céu prontos e acabados dificilmente são levados a sério.

Deming (*apud* Vasconcelos, Rouiller, Machado & Medeiros, 2006)⁹ estabelece 14 princípios fundamentais para o sucesso de um programa de melhoria de qualidade, entre os quais destacam-se os seguintes:

- a) Melhorar constantemente o sistema de produção e os serviços de forma a maximizar o binômio qualidade/produtividade.
- b) Institucionalizar os novos métodos de treinamento no trabalho.
- c) Institucionalizar e fortalecer os papéis de liderança.
- d) Eliminar os medos.
- e) Quebrar as barreiras entre departamentos.
- f) Eliminar *slogans*, exortações e metas de produtividade, pois isso pode levar à queda na qualidade.
- g) Eliminar cotas-padrão arbitrárias e gerenciamento por objetivos.
- h) Institucionalizar um vigoroso programa de educação e automelhoria.
- i) Colocar todos para trabalhar pela modificação em prol da qualidade.

A Seção 12.5 apresenta, com detalhes, um modelo de implantação e melhoria contínua de processos visando à qualidade.

11.4 Gestão da Qualidade

A gestão da qualidade do produto de software pode ser considerada uma atividade de gerenciamento que pode ser efetuada pelo gerente de projeto, mas preferencialmente deveria ser realizada por um gerente ou equipe especializados. Consiste no planejamento e na execução das ações necessárias para que o produto satisfaça aos requisitos de qualidade estabelecidos (Seção 11.6).

Crosby (1979) define um modelo de maturidade organizacional em relação à qualidade baseado em cinco estágios:

- a) *Desconhecimento*: quando a empresa não sabe sequer que tem problemas com qualidade. Não há compreensão de que a qualidade seja um objetivo ou processo de negócio, ferramentas não são usadas ou conhecidas, e inspeções de qualidade não são realizadas.
- b) *Despertar*: a empresa reconhece que tem problemas com a qualidade e que precisa começar a lidar com eles, mas ainda vê isso como um mal necessário, não como fonte de lucro.
- c) *Alinhamento*: o gerenciamento da qualidade se torna uma ferramenta institucional e os problemas vão sendo priorizados e resolvidos à medida que surgem.
- d) *Sabedoria*: a prevenção de problemas, e não apenas sua correção, torna-se rotina na empresa. Problemas são identificados antes que surjam, e todos os processos e rotinas estão abertos a mudanças visando à melhoria da qualidade.
- e) *Certeza*: a gestão da qualidade é uma constante e parte essencial do funcionamento da empresa. Quase todos os problemas são prevenidos e eliminados antes de surgirem.

⁹Disponível em: <www.cin.ufpe.br/~if720/downloads/Mod.01.MPS_Engenharia%26QualidadeSoftware_V.28.09.06.pdf>. Acesso em: 21 jan. 2013.

Segundo Crosby, o custo relativo da qualidade vai diminuindo gradativamente à medida que sobem os níveis de maturidade.

Duas técnicas de controle de qualidade conhecidas (Belchior, 1997) são o *walkthrough* e as *inspeções*. Ambas baseiam-se em um processo de verificação sistemática e cuidadosa dos produtos do trabalho por terceiros para detectar defeitos. Outra técnica de garantia de qualidade é o teste sistemático de software, abordado em detalhes no Capítulo 13.

Além das técnicas de inspeção e teste, que podem ser aplicadas com qualquer ciclo de vida, pode-se mencionar também o modelo *cleanroom*. Trata-se de um ciclo de vida à parte que visa, a partir de especificações formais, produzir produtos de software isentos de defeitos desde sua origem, não necessitando de testes de unidade e integração. As subseções seguintes vão apresentar mais alguns detalhes sobre as técnicas *walkthrough* e de *inspeção*, bem como uma breve introdução ao modelo *cleanroom*.

11.4.1 WALKTHROUGH

O *walkthrough* (Yourdon, 1985) é uma forma de avaliação do produto que utiliza uma equipe de especialistas na qual cada um faz uma análise prévia do produto. Depois, três a cinco pessoas reúnem-se por cerca de duas horas para trocar impressões sobre o produto e sugerir melhorias.

Além dos analistas, a reunião de *walkthrough* deve contar preferencialmente com desenvolvedores e usuários, que poderão apresentar rapidamente respostas a eventuais dúvidas, como “este requisito devia ter sido implementado dessa forma mesmo?”.

Ao término da reunião, os participantes votam pela *aceitação do produto*, pela *aceitação com modificações parciais* ou pela *rejeição*. Sempre que houver modificações recomendadas, o produto deverá passar por um novo *walkthrough*.

Os papéis em uma reunião *walkthrough* são os seguintes (entre outros)¹⁰:

- a) *Apresentador*: geralmente é o autor do artefato, que o descreve, bem como as razões para ele ser dessa forma. Antes da reunião, ele entrega as especificações do artefato ao coordenador, que as distribui à equipe.
- b) *Coordenador*: é o moderador da reunião. Seu trabalho é manter todos focados nas tarefas e não se envolver em discussões. O ideal é que esse papel seja executado por alguém de fora da equipe.
- c) *Secretário*: é o responsável por tomar nota das discussões e decisões. Suas notas deverão ser aprovadas pelos participantes ao final da reunião.
- d) *Oráculo de manutenção*: é o inspetor de garantia de qualidade cujo trabalho é certificar-se de que o código produzido seja compreensível e manutenível, de acordo com os padrões da empresa.
- e) *Guardião dos padrões*: seu trabalho é certificar-se de que o código produzido esteja de acordo com os padrões de programação estabelecidos previamente pela equipe. Se não houver padrões estabelecidos, possivelmente muito tempo da reunião será perdido com discussões irrelevantes.
- f) *Representante do usuário*: pode estar presente em algumas reuniões, especialmente naquelas que discutem requisitos, para garantir que o cliente realmente receba o produto que espera.
- g) Outros desenvolvedores poderão participar e contribuir com a discussão apresentando outros pontos de vista.

É importante mencionar que a reunião deve seguir estritamente o planejamento inicial, mantendo a discussão produtiva e objetiva. O objetivo principal é avaliar os defeitos, e não os desenvolvedores. Além disso, a reunião não será usada para corrigir defeitos, apenas encontrá-los. O processo de reparação vai ocorrer depois.

Erros triviais, como erros ortográficos em janelas, não necessitam de discussão. Apenas os erros mais graves. Em relação à psicologia das reuniões, os seguintes perfis devem ser observados:

- a) *Programadores gênios*: especialmente se forem aqueles gênios arrogantes, impacientes e de mente estreita, podem causar problemas. Devem ser valorizados, pois são capazes de detectar defeitos com facilidade (e alimentam seu ego com isso). O coordenador da sessão deve ter humildade e controle para não iniciar discussões com eles, nem deixar que outros o façam, pois isso tornará o trabalho improductivo.

¹⁰Disponível em: <www.hep.wisc.edu/~jnb/structured_walkthroughs.html>. Acesso em: 21 jan. 2013.

- b) *Pessoas defensivas e inseguras*: deve-se ter cuidado com essas pessoas, pois frequentemente se sentirão atingidas pessoalmente pelas críticas feitas ao seu código. Também é preciso tomar muito cuidado para que seja mantida a discussão sobre o produto, e não sobre os programadores. A reunião de *walkthrough* não é o momento para tentar resolver a vida deles.
- c) *Conservadores*: também poderão causar problemas algumas vezes, pois buscam se manter fiéis às tradições estabelecidas. Deve-se dar atenção às suas opiniões, porque a área de programação é muito sujeita a modismos, mas deve-se também procurar evitar que iniciem discussões improdutivas.
- d) *Alienados*: não estão interessados no mundo real e primam mais pelo processo do que pelo produto, podendo tornar-se um incômodo sério. O coordenador sempre deve ter em mente que o processo só é útil quando ajuda a produzir da melhor forma possível. O processo não é uma religião que se deva seguir cegamente. As regras existem porque há objetivos a alcançar, e não porque foram ditadas por alguma divindade da computação, mas, muitas vezes, os alienados não percebem isso.

Enfim, muitos problemas interpessoais poderão surgir nas primeiras reuniões. Por isso, é necessário que o coordenador seja experimentado e competente na condução das reuniões para que, com o tempo, a equipe aprenda a se manter estritamente focada em seu objetivo, que é a *detecção de defeitos nos programas*.

11.4.2 INSPEÇÕES FAGAN

As *inspeções Fagan* (Fagan, 1976)¹¹ consistem em um processo estruturado para tentar encontrar defeitos no código, diagramas ou especificações. Uma inspeção Fagan parte do princípio de que toda atividade que tenha critérios de entrada e saída bem definidos pode ser avaliada de forma a verificar se efetivamente produz a saída especificada.

Como as atividades de processos de software (Seção 2.3) devem ser sempre definidas em termos de artefatos de entrada e saída, elas se prestam bem a serem avaliadas por inspeções Fagan.

Assim, os artefatos de entrada e saída equivalem aos critérios de entrada e saída para as inspeções, e qualquer desvio encontrado nos artefatos de saída é considerado um *defeito*. Defeitos podem ser classificados em diferentes tipos, como *graves* e *triviais*. Um *defeito grave* é caracterizado pelo não funcionamento do produto, como uma função faltando, por exemplo. Um *defeito trivial* é uma característica errada que não afeta a capacidade de funcionamento do software, como um erro ortográfico em uma janela de sistema, por exemplo.

Normalmente, os papéis na equipe de inspeção são os seguintes:

- a) *Autor*: o programador, *designer* ou analista, ou seja, a pessoa que produziu o artefato.
- b) *Narrador*: analisa, interpreta, sumariza o artefato e seus critérios de aceitação.
- c) *Revisores*: revisam o artefato com o objetivo de detectar eventuais defeitos.
- d) *Moderador*: é o responsável pela sessão de inspeção e pelo andamento do processo.

O processo de inspeção, tipicamente, abrange as seguintes atividades (Fagan, 1986)¹²:

- a) *Planejamento*: inclui preparação dos materiais (artefatos), convite aos participantes e alocação do espaço de trabalho.
- b) *Visão geral*: inclui instrução prévia (apresentação) aos participantes sobre os materiais a serem inspecionados e a atribuição de papéis a esses participantes.
- c) *Preparação*: nessa atividade, os participantes analisam os artefatos sob inspeção e o material de suporte, anotando possíveis defeitos e questões para a reunião de inspeção.
- d) *Reunião de inspeção*: é o momento em que efetivamente se discutem os defeitos encontrados e se decide o que fazer com eles.
- e) *Retrabalho*: é a atividade em que o autor do artefato vai corrigir os defeitos apontados na reunião de inspeção.
- f) *Prosseguimento (follow-up)*: essa atividade considera que todos os defeitos foram corrigidos e o produto está aprovado para prosseguir para a fase seguinte, ou entrega.

¹¹Disponível em: <www.mfagan.com/pdfs/ibmfagan.pdf>. Acesso em: 21 jan. 2013.

¹²Disponível em: <www.mfagan.com/pdfs/aisi1986.pdf>. Acesso em: 21 jan. 2013.

É responsabilidade do moderador da inspeção verificar se todos os defeitos foram corrigidos e se o produto pode ir para *follow-up*. Caso ele considere que os defeitos não foram adequadamente corrigidos ou que novos defeitos foram introduzidos no processo de correção, deverá determinar que o produto retorne ao processo de inspeção.

Defeitos triviais podem simplesmente ir para retrabalho sem que sejam necessárias novas inspeções. Contudo, os defeitos graves devem ser novamente analisados pelo processo de inspeção, que deve ser reiniciado na sua primeira fase (planejamento).

As reuniões são importantes, pois produzem sinergia no grupo, incluindo a necessária troca de experiências e a afinação de discurso, o que funciona também como atividade de formação continuada de inspetores. Porém, em função dos custos com deslocamento de pessoas para reuniões desse tipo, cada vez mais têm sido utilizados meios eletrônicos para que elas possam acontecer de forma virtual. O mesmo ocorre com o material, disponibilizado eletronicamente para minimizar também o uso de papel (Genuchten, Cornelissen & Dijk, 1997).

11.4.3 MÉTODO CLEANROOM

O método *cleanroom* (Mills, Dyer & Linger, 1987)¹³ é uma maneira bastante formal de desenvolver software com foco na qualidade, que procura evitar que defeitos sejam introduzidos durante o desenvolvimento. O nome foi inspirado nas salas de fábricas de circuitos integrados (“salas limpas”). O método estabelece que um circuito não precisa ser limpo depois de fabricado, porque já é produzido em um ambiente sem sujeira. Da mesma forma, espera-se que um software produzido em um ambiente *limpo* possa ser isento de defeitos.

O principal objetivo do método é produzir software que apresente taxa zero de defeitos durante seu uso (Linger & Trammell, 1996)¹⁴.

Os princípios básicos do método *cleanroom* são os seguintes:

- a) *Desenvolvimento baseado em métodos formais*: usam-se estruturas de *caixas* (ver a seguir) para especificar o software. Uma equipe de revisão verifica se o produto satisfaz à especificação.
- b) *Desenvolvimento incremental sob controle estatístico de qualidade*: utiliza-se desenvolvimento baseado em ciclos iterativos, nos quais funcionalidades vão sendo agregadas gradualmente, e o controle estatístico é feito por uma equipe independente, cuja função é garantir que a qualidade permaneça em nível aceitável.
- c) *Programação estruturada*: apenas um conjunto limitado de estruturas de abstração é permitido. O desenvolvimento de programas é um processo de refinamento passo a passo que utiliza essas estruturas e transformações que garantem a preservação da corretude em relação à especificação, até chegar ao código.
- d) *Verificação estática*: o software desenvolvido é verificado estaticamente, usando inspeções de código rigorosas. Não há teste de unidade nem de integração.
- e) *Testes baseados em medição estatística*: um conjunto de testes baseados em especificações formais e estatisticamente representativo é selecionado e aplicado. Os testes de sistema são realizados diretamente.

O método *cleanroom* é definido em termos de 14 processos e 20 artefatos (produtos do trabalho), definidos por Linger e Trammell (1996).

O princípio básico do método é que programas podem ser vistos como regras para funções ou relações matemáticas. Um programa faz transformações em uma entrada, produzindo uma saída, o que pode ser especificado por uma função de mapeamento. Assim, programas podem ser projetados como decomposições de suas especificações funcionais e verificados formalmente.

As estruturas de caixas são três estruturas matemáticas usadas pelo método, as quais mapeiam *estímulos* (entradas) e *históricos* de estímulos (entradas prévias) em *respostas* (saídas):

- a) *Black box*: define o comportamento esperado de uma função do sistema em todos os seus contextos de uso: *estímulo corrente + histórico de estímulos → resposta*.
- b) *State box*: pode ser derivada de uma *black box* e define o histórico de estímulos como um estado: *estímulo corrente + estado atual → resposta + novo estado*.

¹³Disponível em: <ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1695817>. Acesso em: 21 jan. 2013.

¹⁴Disponível em: <www.sei.cmu.edu/reports/96tr022.pdf>. Acesso em: 21 jan. 2013.

- c) *Clear box*: é idêntica a uma *state box*, mas sua especificação é feita por procedimentos em linguagem de programação, ou seja, é a realização de uma *state box*.

Em orientação a objetos, a *black box* pode ser entendida como o comportamento especificado para um objeto, a *state box* como o encapsulamento de dados do objeto, e a *clear box* como seus métodos.

Um dos maiores problemas para a adoção desse método é a dificuldade de se obterem desenvolvedores suficientemente preparados em lógica e matemática para atuar com especificação formal de sistemas¹⁵.

11.5 Medição da Qualidade

A Seção 9.5 já apresentou algumas questões referentes a métricas e medições do ponto de vista do processo de gerência de projetos de software. Aqui será aprofundado o aspecto de medição da qualidade do produto de software.

Kitshenham e Lawrence (1996) indicam que a qualidade de um produto, do ponto de vista da satisfação do usuário, será resultado de três fatores:

- Funcionalidade*, cuja medida será estar *presente* ou *ausente*.
- Comportamento*, isto é, as qualidades não funcionais, que normalmente são mensuráveis em um dado *intervalo*.
- Restrições*, que determinam como o usuário pode usar o produto.

Quando os usuários pensam em qualidade de software, em geral lembram-se da característica de confiabilidade (Seção 11.1.2), isto é, do tempo em que o produto funciona sem apresentar defeitos. Porém, caso o software já seja relativamente confiável, outras qualidades entrarão com mais ênfase nas expectativas do usuário, como usabilidade e eficiência.

Gilb (1987) sugere que essas características podem ser medidas de forma objetiva. Por exemplo, o tempo de aprendizagem de um sistema pode ser medido como o tempo médio que os usuários levam para aprender a executar um conjunto predeterminado de tarefas.

A técnica de Gilb pode ser generalizada para outras características. A ideia é quebrar a característica de qualidade em outras menores até que se encontrem aquelas que possuam um *procedimento operacional objetivo* para serem avaliadas.

Do ponto de vista do desenvolvedor, a qualidade pode ser medida a partir de duas variáveis principais: a quantidade de defeitos e os custos com retrabalho ao longo do desenvolvimento.

A contagem de defeitos deve ser sempre relacionada com o momento em que os defeitos são introduzidos e, principalmente, encontrados. Por exemplo, encontrar um defeito durante os testes de unidade ou integração não é tão sério quanto encontrar um defeito em um produto já instalado no cliente.

A contagem de defeitos nas diferentes fases poderá dar uma boa medida da eficácia dos processos da empresa, bem como permitir a avaliação de mudanças nesses mesmos processos ou nas ferramentas de desenvolvimento. Se a quantidade de defeitos diminuir ou se os defeitos começarem a ser identificados mais cedo, isso ocorrerá porque a mudança de processo foi salutar em relação a essa métrica.

O número de defeitos em um sistema não tem uma relação necessariamente linear com os custos de retrabalho. Por vezes, defeitos são muito simples de depurar e corrigir. Outras vezes, um único defeito pode ter um impacto catastrófico no projeto, exigindo grandes mudanças estruturais e consumo de tempo e recursos para sua correção (como o caso do *bug* do ano 2000¹⁶). A contagem de tempo com retrabalho é de difícil implementação, mas uma opção é definir “retrabalho” como um dos tipos de ação nas folhas de tempo (Seção 9.4.1). Assim, pode-se verificar quanto esforço efetivamente foi empregado refazendo o que já havia sido aprovado.

Poderá ser útil distinguir o retrabalho causado por defeitos do software, o causado por erros nos requisitos ou, ainda, o causado pela necessidade de aprimorar aspectos não funcionais do software, como sua eficiência. Normalmente, apenas o retrabalho causado por defeito ou erro nos requisitos será efetivamente contabilizado como uma atividade não produtiva, pois a melhoria ou o aprimoramento de outras qualidades do software será um investimento.

¹⁵Disponível em: <www.cs.st-andrews.ac.uk/~ifs/Books/SE9/Web/Cleanroom/experience.htm>. Acesso em: 21 jan. 2013.

¹⁶Disponível em: <en.wikipedia.org/wiki/Year_2000_problem>. Acesso em: 21 jan. 2013.

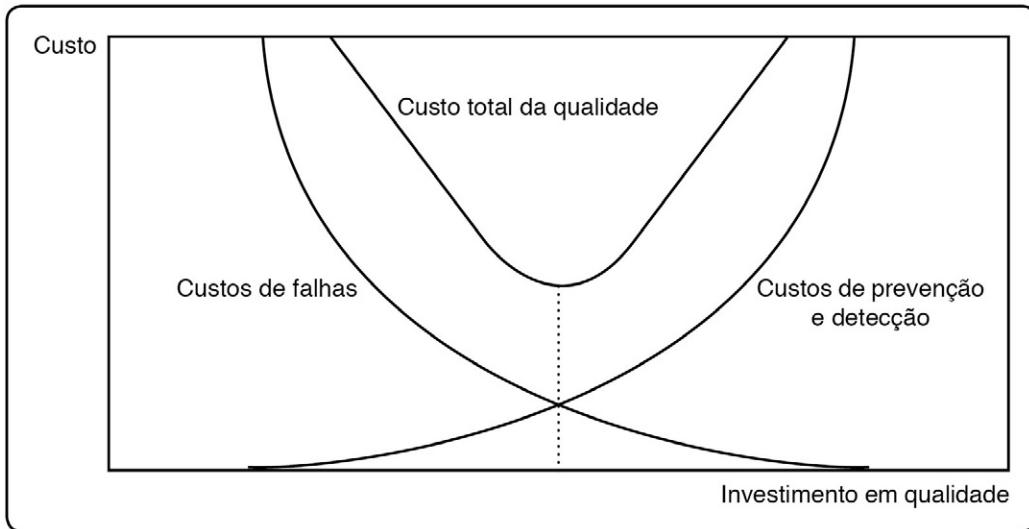


Figura 11.3 Relação entre investimento em qualidade e economia relacionada a falhas¹⁸.

11.6 Requisitos de Qualidade

Os requisitos de qualidade de software podem ser catalogados, mas cada produto terá um conjunto de requisitos diferente, pois qualidade também tem custo. Algumas subcaracterísticas de qualidade são sempre desejáveis e podem ser obtidas a partir de um bom processo de desenvolvimento, como um software livre de defeitos, por exemplo. Contudo, outras qualidades (como portabilidade, por exemplo) poderão ser eletivas e o custo de sua inclusão no software poderá não ser justificável. Belchior (1997)¹⁷ observa que qualidade não é sinônimo de perfeição, mas algo factível, relativo, dinâmico e evolutivo que se amolda aos objetivos a serem atingidos.

Os requisitos de qualidade devem fazer parte da própria especificação do produto. Normalmente são requisitos suplementares, ou seja, definidos para o software como um todo, e não para uma função individual. Mas também podem ser requisitos não funcionais quando se aplicam a uma ou a poucas funções.

Como os requisitos de qualidade são suplementares ou não funcionais, é de se esperar que possam ser classificados em diferentes graus de obrigatoriedade. Pode-se usar aqui o padrão MOSCOW (*Must, Should, Could e Would*) para determinar o grau de necessidade que determinado requisito de qualidade seja cumprido. Kerzner (1998) indica que existe um ponto ótimo para o investimento em qualidade que baixa os custos com falhas o suficiente para compensar o investimento (Figura 11.3).

Se, de um lado, as medidas de qualidade mais fundamentais (como software livre de defeitos) podem ficar subentendidas, as medidas de qualidade eletivas (como portabilidade) somente serão incorporadas ao produto se forem explicitamente solicitadas nos requisitos.

O ideal é que cada requisito de qualidade seja definido por uma especificação objetiva ou, melhor ainda, uma métrica que possa ser usada para medir o produto final e confirmar se atende ou não ao requisito.

Por exemplo, se o requisito de qualidade for “O software deve ser fácil de usar”, como avaliar se o produto final atende a essa especificação? Esse requisito está colocado de maneira subjetiva, ou seja, duas pessoas poderão ter opiniões diferentes sobre o fato de determinado produto de software ser ou não fácil de usar. Dessa forma, não há como avaliar se o requisito foi atendido, mas o problema, nesse caso, é que o requisito em si não é objetivo.

Melhor seria estabelecer um requisito como “Todas as janelas de sistema devem ter acesso a uma tela de ajuda acessível por F1”. Dessa forma, o produto final pode ser inspecionado e o requisito de qualidade, conforme especificado, pode ser avaliado como satisfeito ou não.

¹⁷Disponível em: <www.boente.eti.br/fuzzy/tese-fuzzy-belchior.pdf>. Acesso em: 21 jan. 2013.

¹⁸Kerzner (1998).

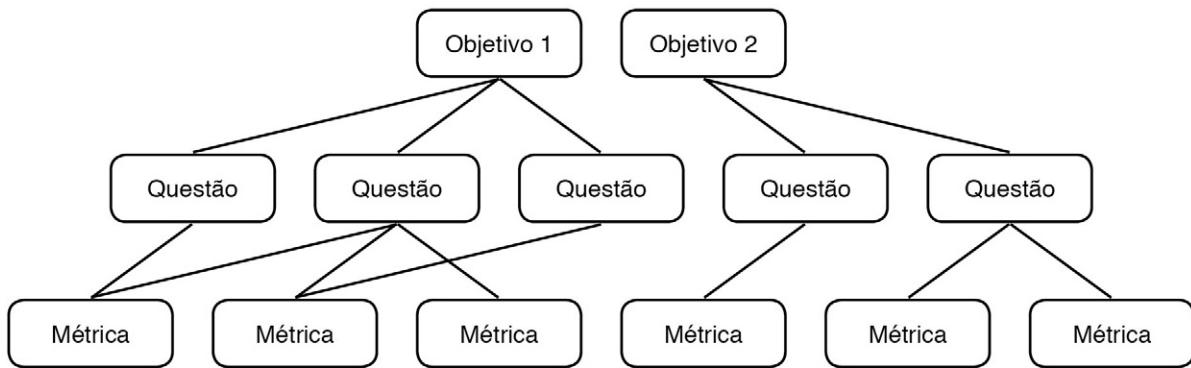


Figura 11.4 Estrutura hierárquica do modelo GQM.

11.7 GQM (*Goal/Question/Metric*) e Avaliação da Qualidade

O método GQM (Basili, Caldiera & Rombach, 1994)¹⁹, um acrônimo para *Goal/Question/Metric*, é uma abordagem para a avaliação de qualidade de software. O GQM define um modelo de mensuração em três níveis:

- Nível conceitual (Goal/objetivo)*: um objetivo é definido para um objeto por uma variedade de razões, com respeito a vários modelos de qualidade, a partir de vários pontos de vista e relativo a um ambiente em particular (os *objetos* a serem medidos podem ser produtos, processos ou recursos).
- Nível operacional (Question/questão)*: um conjunto de questões é usado para definir modelos de objetos de estudo e, então, focar no objeto que caracteriza a avaliação ou a obtenção de um objetivo específico.
- Nível quantitativo (Metric/métrica)*: um conjunto de dados baseados nos modelos é associado a cada questão para respondê-la de forma quantitativa (os dados podem ser *objetivos*, se dependerem apenas do objeto avaliado, ou *subjetivos*, se dependerem de uma interpretação do avaliador).

Um guia completo de aplicação de GQM também pode ser consultado na internet (Solingen & Berghout, 1999)²⁰. Outra publicação com exemplos práticos de aplicação de GQM é o trabalho de Wangenheim (2000).

Em GQM, a definição do processo de avaliação é feita de forma *top-down*, ou seja, dos objetivos para as métricas, enquanto a interpretação dos resultados é feita de forma *bottom-up*, ou seja, das métricas para os objetivos. A Figura 11.4 apresenta o modelo GQM esquematicamente. Observe que as métricas não são exclusivas para as questões.

Santos e Pretz (2009)²¹ apresentam um estudo de caso em que GQM é usado para avaliar um projeto de desenvolvimento de software. Cada risco importante do sistema é analisado como um objetivo, para o qual são definidas questões e métricas. Inicialmente, os autores associam os riscos identificados às características e subcaracterísticas de qualidade da Norma 9126 (Tabela 11.9), na época ainda em vigência.

Assim, para cada risco identificado e possivelmente para cada subcaracterística de qualidade associada ao risco, um objetivo é estabelecido. Para cada objetivo, uma ou mais questões são colocadas, e, para cada questão, uma ou mais métricas são definidas (Tabela 11.10).

Como se pode ver na tabela, o *objetivo* é especificado de acordo com um padrão estabelecido pelo próprio GQM, que sugere que objetivos sejam estabelecidos a partir de diferentes dimensões:

- Propósito*: verbo que representa o objetivo, como “avaliar”.
- Questão*: adjetivo referente ao objeto, como “a maturidade de”.
- Objeto*: objeto em avaliação, como “o software”.
- Ponto de vista*: para quem a avaliação é feita, como “do ponto de vista do cliente”.

¹⁹Disponível em: <[ftp://ftp.cs.umd.edu/pub/sel/papers/gqm.pdf](http://ftp.cs.umd.edu/pub/sel/papers/gqm.pdf)>. Acesso em: 21 jan. 2013.

²⁰Disponível em: <www.iteva.rug.nl/gqm/GQM%20Guide%20non%20printable.pdf>. Acesso em: 21 jan. 2013.

²¹Disponível em: <tconline.feevale.br/tc/files/6163_38.pdf>. Acesso em: 21 jan. 2013.

TABELA 11.9 Associação de riscos às características e subcaracterísticas de qualidade²²

| Sistema Exemplo | | NBR ISO/IEC 9126 | |
|---|--|------------------|------------------------|
| Requisito | Risco | Característica | Subcaracterística |
| Envio e recepção de nova versão à base centralizada | R001 – Indisponibilidade do sistema para o usuário. | Confiabilidade | Tolerância a falhas |
| | R002 – Insuficiência dos recursos envolvidos com a produção do sistema, causando indisponibilidade. | | Maturidade |
| | | Eficiência | Recuperabilidade |
| Envio e recepção de informações dos sistemas relacionados | R003 – Interceptação de informações sigilosas no tráfego de rede utilizado pelo sistema. | Funcionalidade | Utilização de recursos |
| | R004 – Acesso liberado dos usuários da aplicação às informações inseridas no banco local, instalado na estação de trabalho do usuário. | | |

TABELA 11.10 Exemplo de aplicação do modelo GQM²³

| Risco R001 – Indisponibilidade do sistema para o usuário | | | | |
|--|--|--|--|---|
| Característica | Subcaracterística | Objetivo | Questão | Métrica |
| Confiabilidade | Maturidade | Avaliar a capacidade de prevenção de falhas do sistema do ponto de vista do usuário. | Quantas falhas foram detectadas durante um período definido de experimentação? | Número de falhas detectadas/número de casos de testes |
| | | | Quantos padrões de defeitos são mantidos sob controle para evitar falhas críticas e sérias? | Número de ocorrências de falhas sérias e críticas evitadas conforme os casos de testes de indução de falhas/número de casos de testes de indução de falhas executados |
| | Tolerância a falhas e recuperabilidade | Avaliar a disponibilidade do sistema do ponto de vista do usuário. | Quão disponível é o sistema para uso durante um período de tempo específico? | Tempo de operação/(tempo de operação + tempo de reparo) Total de casos em que o sistema estava disponível e foi utilizado com sucesso pelo usuário / número total de casos em que o usuário tentou usar o software durante um período de tempo |
| | | | Qual é o tempo médio em que o sistema fica indisponível quando uma falha ocorre, antes da inicialização? | Tempo ocioso total (indisponível)/número de quedas do sistema |
| | | | Qual o tempo médio que o sistema leva para completar a recuperação desde o início? | Soma de todos os tempos de recuperação do sistema inativo em cada oportunidade/número total de casos em que o sistema entrou em recuperação |
| | | | | |

²²Santos e Pretz (2009).²³Santos e Pretz (2009).

TABELA 11.10 Exemplo de aplicação do modelo GQM22 (*Continuação*)

Risco R002 – Insuficiência dos recursos envolvidos com a produção do sistema, causando indisponibilidade

| Característica | Subcaracterística | Objetivo | Questão | Métrica |
|----------------|------------------------|--|--|--|
| Eficiência | Utilização de recursos | Avaliar a eficiência na utilização de recursos de produção do ponto de vista do usuário. | Qual é o limite absoluto de transmissões necessárias para cumprir uma função? | Número máximo de mensagens de erro e falhas relacionadas à transmissão do primeiro ao último item avaliado/máximo requerido de mensagens de erro e falhas relacionadas à transmissão |
| | | | O sistema é capaz de desempenhar tarefas dentro da capacidade de transmissão esperada? | Capacidade de transmissão / capacidade de transmissão específica projetada para ser usada pelo software durante sua execução |

Risco R003 – Interceptação de informações sigilosas no tráfego de rede utilizado pelo sistema

| Característica | Subcaracterística | Objetivo | Questão | Métrica |
|----------------|---------------------|--|---|---|
| Funcionalidade | Segurança de acesso | Avaliar a integridade dos dados do sistema do ponto de vista do usuário. | Qual é a frequência de eventos de corrupção de dados? | Número de vezes que o maior evento de corrupção de dados ocorreu/número de casos de testes executados que causaram eventos de corrupção de dados (número de vezes que o menor evento de corrupção de dados ocorreu/número de casos de testes executados que causaram eventos de corrupção de dados) |

Risco R004 – Acesso liberado dos usuários da aplicação às informações inseridas no banco local, instalado na estação de trabalho do usuário

| Característica | Subcaracterística | Objetivo | Questão | Métrica |
|----------------|---------------------|---|---|--|
| Funcionalidade | Segurança de acesso | Avaliar o controle de acesso ao sistema do ponto de vista do usuário. | Quão completa é a trilha de auditoria sobre o acesso do usuário ao sistema e dados? | Número de acessos do usuário ao sistema e dados gravados no log de acesso/número de acessos do usuário ao sistema e dados realizados durante a avaliação |
| | | | Quão controlável é o acesso ao sistema? | Número (tipos diferentes) de operações ilegais detectadas/ número (tipos diferentes) de operações ilegais especificadas |

Os autores acrescentam, ainda, as técnicas de teste, que permitirão avaliar a questão de acordo com a métrica definida. Essa informação foi omitida na tabela, mas pode ser consultada em Santos e Pretz (2009).

Qualidade de Processo

Este capítulo vai tratar da qualidade do processo de desenvolvimento de software, iniciando pela recém-evoluída Norma ISO/IEC 90003 (Seção 12.1), que define várias orientações sobre como avaliar e melhorar processos. A Norma 15504, ou SPICE (Seção 12.2), é um modelo de avaliação compatível com o CMMI (Seção 12.3) do SEI e o modelo brasileiro MPS.BR (Seção 12.4). Na sequência, o capítulo aborda o guia de melhoria de processos SEI-IDEAL (Seção 12.5) e discute os *fatores humanos* (Seção 12.6) relacionados à mudança de processos de trabalho na empresa. Por fim, apresenta brevemente o conceito ainda em evolução de *Linhas de Processo de Software* (Seção 12.7).

Já foi comentado que a qualidade de produtos de software pode ser fortemente afetada pela qualidade do processo usado para desenvolvê-los. Também foram vistos alguns modelos de processo, como UP, métodos ágeis, espiral, prototipação etc. Cada um desses modelos apresenta vantagens e desvantagens, e cada um deles pode ser mais bem aplicado em determinadas situações do que outros.

Deve-se, porém, diferenciar a questão do modelo teórico em si da questão relacionada à *implementação* do modelo em uma empresa específica. Ou seja, um modelo pode ser intrinsecamente adequado, mas a empresa pode estar usando-o de forma inadequada.

Em função dessa observação, foram definidos modelos de avaliação de qualidade da implementação de processos nas empresas. Esses modelos não prescrevem este ou aquele ciclo de vida, mas avaliam quão bem uma empresa está aplicando e gerenciando seu processo de desenvolvimento com o modelo de processo escolhido.

12.1 ISO/IEC 90003

A ISO 90003:2004^{1,2} é a versão mais atual da antiga Norma ISO 9000-3:1997, que era um guia para aplicação da ISO 9001 à indústria de software. Assim, cada aspecto da 9001 tem um correspondente na 90003, especialmente detalhado para sua aplicação na indústria de software.

¹Disponível em: <www.praxiom.com/iso-90003.htm>. Acesso em: 21 jan. 2013.

²Disponível em: <www.iso.org/iso/catalogue_detail?csnumber=35867>. Acesso em: 21 jan. 2013.

A Norma NBR ISO 9000-3 (Kehoe & Jarvis, 1996) é compreendida como uma das primeiras tentativas de melhorar o processo de produção de software. Ela apresentava padrões para gerenciamento e garantia de qualidade aplicáveis a companhias de qualquer tamanho. A ISO 9000-3 foi publicada em 1991 e também consistia em uma diretriz para a aplicação da norma ISO 9001 ao processo de desenvolvimento, fornecimento e manutenção de software.

Um dos problemas com a 9000-3 é que ela não tratava a melhoria contínua do processo, apenas indicava os processos que as empresas deveriam ter e manter. Essa deficiência foi corrigida com a 90003.

A Norma 90003 relaciona-se com um conjunto de normas ISO que dizem respeito aos aspectos de qualidade de processo de software³. Ela considera que o processo de produção de software é variado, podendo ser dirigido por diferentes modelos, mas ao mesmo tempo considera que determinadas fases ou disciplinas existirão com maior ou menor ênfase ou, ainda, com diferentes formas de organização em qualquer processo de produção.

Existem duas formas definidas para a aplicação das normas da família 9000:

- a) *Gestão da qualidade*: nessa forma, a qualidade é vista como uma filosofia que deve ser impregnada em todos os setores da empresa, mas dirigida pela alta administração. Sua norma básica é a ISO 9004-1.
- b) *Garantia da qualidade*: nessa forma, procura-se aplicar qualidade a processos e produtos de forma a assegurar ao cliente que a empresa fornecedora tem capacidade de atender aos requisitos com qualidade. Suas normas básicas são a ISO 9001, 9002 e 9003.

A série 9000 é conhecida como um conjunto de normas que enfatiza a documentação de processos e procedimentos. Ela estabelece quatro níveis de documentação cada vez mais detalhados e complexos:

- a) *Nível 1*: neste nível genérico é exigido basicamente um manual geral de qualidade explicando a política e o sistema de qualidade, bem como a estrutura organizacional da empresa e os papéis ou responsabilidades.
- b) *Nível 2*: neste nível, os processos são documentados pelos *manuais de procedimentos*. Eles devem abranger todas as atividades ligadas ao desenvolvimento e fornecimento de software, independentemente do ciclo de vida adotado, estabelecendo como as atividades devem ser executadas, quais suas dependências e quais os perfis de responsáveis (Capítulo 2).
- c) *Nível 3*: neste nível devem ser detalhadas as instruções sobre como proceder para o eficaz funcionamento do sistema de qualidade, abrangendo as atividades de teste, inspeção, especificações, modelo e requisitos de qualidade etc. (Zahran, 1997).
- d) *Nível 4*: neste nível devem ser mantidos os registros de qualidade, ou seja, resultados de testes e inspeções que comprovam que as atividades do sistema de qualidade documentado no nível 3 são efetivamente executadas.

A documentação relacionada ao sistema de qualidade também pode ser classificada segundo outros critérios:

- a) *Documentação da qualidade*: todos os documentos que estabelecem processos, políticas e regras sobre como executar as atividades relacionadas à qualidade.
- b) *Registros da qualidade*: resultados dos processos de avaliação da qualidade que indicam que os documentos da qualidade não são apenas letra morta, mas que são efetivamente usados na empresa.

Espera-se que os documentos de qualidade sejam estabelecidos e, à medida que amadurecem, alcancem certa estabilidade, embora nunca devam estagnar. Já os registros da qualidade são criados diariamente para cada projeto em desenvolvimento na empresa.

As subseções seguintes apresentam, de forma abrangente, o espírito da ISO 90003.

12.1.1 REQUISITOS E ORIENTAÇÕES SISTÊMICOS

Inicialmente, deve-se estabelecer o sistema de gerenciamento da qualidade para produtos de software, o que inclui o seguinte:

- a) *Desenvolver um sistema de qualidade para produtos de software e serviços relacionados*: devem-se identificar e descrever os processos que fazem parte do sistema de qualidade (desenvolvimento, planejamento do

³Disponível em: <www.lyfreitas.com/pdf/ISO%209000-3.pdf>. Acesso em: 21 jan. 2013.

desenvolvimento, planejamento da qualidade, operação e manutenção). A descrição dos processos deve ser uma estrutura sequencial em que os processos podem interagir uns com os outros.

- b)** *Implementar um sistema de qualidade para produtos e serviços de software:* devem-se usar, para isso, os processos estabelecidos no sistema de qualidade (ver **item a)**). A efetividade dos processos deve ser gerenciada e receber suporte.
- c)** *Melhorar o sistema de gerenciamento de qualidade orientado a software:* deve-se monitorar, mensurar e melhorar a efetividade dos processos.

Paralelamente, o sistema de qualidade orientado ao software deverá ser *documentado*:

- a)** *Desenvolver os documentos do sistema de gerenciamento da qualidade:* isso inclui os documentos que instauram ou implementam o sistema de qualidade, os que descrevem os processos de software e os que descrevem os modelos de ciclo de vida adotados.
- b)** *Preparar o manual do sistema de gerenciamento da qualidade:* devem-se documentar os procedimentos e o escopo do sistema de qualidade, descrever como os processos interagem e justificar as reduções ou exclusões do escopo (ou seja, justificar por que alguns processos relacionados a software ficaram de fora, se for o caso).
- c)** *Controlar os documentos do sistema de gerenciamento da qualidade:* devem-se aprovar os documentos formalmente antes de distribuí-los, fornecer as versões corretas atualizadas dos documentos sempre que necessário, prevenir a todo custo o uso de versões obsoletas e preservar a usabilidade dos documentos de qualidade.
- d)** *Manter os registros do sistema de gerenciamento da qualidade:* deve-se provar que os requisitos foram obtidos e que as operações são efetivas, e estabelecer uma abordagem de retenção de registros.

12.1.2 REQUISITOS E ORIENTAÇÕES DE GERENCIAMENTO

Do ponto de vista do gerenciamento, várias recomendações são feitas pela norma. Inicialmente, o gerente deve dar *suporte à qualidade* através das seguintes atividades:

- a)** *Promover a importância da qualidade:* deve-se promover a necessidade de satisfazer aos requisitos do cliente e aos requisitos do produto de software.
- b)** *Desenvolver um sistema de gerenciamento da qualidade:* deve-se dar suporte ao desenvolvimento de um sistema de qualidade, formular a política de qualidade da organização, definir os objetivos de qualidade da organização e fornecer os recursos necessários.
- c)** *Implementar o sistema de gerenciamento da qualidade:* devem-se fornecer os recursos para implementar o sistema de gerenciamento da qualidade e encorajar as pessoas a satisfazer os requisitos do sistema de gerenciamento da qualidade.
- d)** *Aperfeiçoar o sistema de gerenciamento da qualidade:* devem-se realizar revisões do gerenciamento da qualidade e fornecer recursos para aperfeiçoar o sistema de qualidade.

Outro objetivo gerencial a ser perseguido é o *foco no cliente*. Esse objetivo é realizado a partir das seguintes ações:

- a)** *Identificar os requisitos do cliente:* certifique-se de que as pessoas conseguirão identificar os requisitos do cliente.
- b)** *Satisfazer os requisitos do cliente:* certifique-se de que sua organização será capaz de satisfazer aos requisitos do cliente.
- c)** *Melhorar a satisfação do cliente:* certifique-se de que sua organização será capaz de aprimorar a satisfação do cliente.

Além disso, o gerente também deverá *estabelecer a política de qualidade*, o que inclui:

- a)** *Definir a política de qualidade:* garanta que a sua política serve aos propósitos da organização, enfatizando a necessidade de satisfazer aos requisitos e facilitando o desenvolvimento de objetivos da qualidade.
- b)** *Gerenciar a política de qualidade:* sua política deve ser comunicada a toda a organização e deve ser revisada para garantir que continue adequada.

O *planejamento da qualidade* deve ser realizado da seguinte forma:

- a) *Formular os objetivos de qualidade*: deve-se definir um conjunto de objetivos para todas as áreas funcionais e todos os níveis organizacionais.
- b) *Planejar o sistema de gerenciamento da qualidade*: planeje o desenvolvimento do sistema de gerenciamento da qualidade. Realize o planejamento do gerenciamento da qualidade para produtos de software. Planeje como a efetividade do sistema de qualidade será aperfeiçoada.

O gerente também deve *controlar seu sistema de qualidade*, através das seguintes atividades:

- a) *Definir responsabilidades e autoridade*: elas devem ser documentadas e comunicadas.
- b) *Indicar um representante da gerência*: o sistema de gerenciamento da qualidade deve ser supervisionado, seu *status*, relatado, e sua manutenção, suportada.
- c) *Dar suporte às comunicações internas*: deve-se garantir que os processos de comunicação estejam estabelecidos e que eles ocorram entre todas as partes da organização.

Por fim, o gerente também deve realizar *revisões do processo de gerenciamento*:

- a) *Revisar o sistema de gerenciamento da qualidade*: devem-se planejar revisões regulares do sistema de gerenciamento da qualidade. A efetividade do sistema de gerenciamento da qualidade deve ser avaliada. Deve ser mantido um registro das revisões do sistema de qualidade.
- b) *Examinar as entradas para as revisões de gerenciamento*: podem ser examinados resultados de auditoria, oportunidades de aperfeiçoamento e realimentação dos clientes, dados de conformidade do produto de software, informação sobre o desempenho de processo, ações corretivas e preventivas e revisões de gerenciamento de qualidade feitas.
- c) *Gerar saídas a partir das revisões de gerenciamento*: devem-se gerar ações para aprimorar a efetividade do sistema de qualidade e dos produtos de software, bem como para tratar das necessidades de recursos.

12.1.3 REQUISITOS E ORIENTAÇÕES RELACIONADOS A RECURSOS

Com relação aos diversos tipos de recursos necessários, a norma chama a atenção para alguns objetivos listados nesta seção. Inicialmente, é necessário *fornecer recursos de qualidade*:

- a) *Identificar os requisitos dos recursos para o sistema de gerenciamento da qualidade*: devem-se identificar os recursos necessários para dar suporte ao sistema de qualidade, para satisfazer aos requisitos do cliente e aos requisitos regulatórios.
- b) *Fornecer recursos para o sistema de gerenciamento da qualidade*: devem-se fornecer os recursos necessários para dar suporte, implementar e aperfeiçoar o sistema de gerenciamento da qualidade, e também para satisfazer aos requisitos do cliente e aos requisitos regulatórios.

Da mesma forma, também é necessário fornecer *pessoal de qualidade*:

- a) *Usar pessoal competente*: é necessário certificar-se de que o pessoal tenha a experiência, a educação, o treinamento e as habilidades corretas.
- b) *Dar suporte à competência com a definição dos níveis de competência aceitáveis*: deve-se identificar as necessidades de conhecimento e treinamento da organização, seja do pessoal de desenvolvimento de software, seja do pessoal de gerenciamento de projetos. Deve-se criar e avaliar os programas de treinamento da organização e manter um registro de competências.

Também deve ser fornecida uma *infraestrutura de qualidade*:

- a) *Identificar as necessidades de infraestrutura*: deve-se identificar a infraestrutura (hardware, software e instalações) necessária para o desenvolvimento de software, assim como as ferramentas necessárias para gerenciar, desenvolver, suportar, proteger e controlar o software.
- b) *Fornecer e manter a infraestrutura necessária*: deve-se fornecer e manter a infraestrutura (hardware, software e instalações) necessária para o desenvolvimento de software, assim como as ferramentas necessárias para gerenciar, desenvolver, suportar, proteger e controlar o software.

Por fim, deve-se fornecer um ambiente de qualidade:

- a) Identificar as necessidades do ambiente de trabalho.
- b) Implementar as necessidades do ambiente de trabalho.
- c) Gerenciar as necessidades do ambiente de trabalho.

O ambiente de trabalho refere-se a todas as condições e fatores que influenciam o trabalho. Em geral, isso inclui fatores e condições físicas, sociais, psicológicas e ambientais. O ambiente de trabalho inclui iluminação, temperatura, ruído e condições ergonômicas, e também práticas de supervisão e programas de reconhecimento e recompensa.

12.1.4 REQUISITOS E ORIENTAÇÕES PARA REALIZAÇÃO DE PROJETOS

Os processos de realização são os processos de engenharia que criam o produto de software. A norma estabelece que *o planejamento da realização de produtos de software* seja controlado da seguinte forma:

- a) *Planejar os processos de realização de produto de software*: devem-se identificar os requisitos e objetivos de qualidade do produto de software, bem como os requisitos e as necessidades de realização do produto, os requisitos de gerenciamento de riscos e os requisitos de manutenção de registros.
- b) *Desenvolver os processos de realização do produto*: devem-se desenvolver os documentos de realização do produto, o sistema de manutenção de registro da realização do produto e os métodos para controlar a qualidade durante a realização do produto.
- c) *Usar modelos de ciclo de vida para planejar o trabalho*: as tarefas, as atividades e os processos devem ser planejados usando modelos de ciclo de vida adequados. Os modelos devem ser selecionados e usados para executar os projetos de software. Os métodos de desenvolvimento de software adequados também devem ser selecionados.
- d) *Executar o planejamento da qualidade de software*: deve-se planejar como o sistema de gerenciamento da qualidade será aplicado ao desenvolvimento dos produtos de software e também a cada projeto de software.

Também devem ser controlados os processos relacionados ao cliente:

- a) *Identificar os requisitos do produto de software*: devem-se identificar os requisitos que os clientes querem ver satisfeitos, os requisitos ditados pelo uso do produto ou impostos por agências externas, e também aqueles que interessam à organização desenvolvedora. Devem-se estabelecer os métodos que podem ser usados para identificar os requisitos de software, bem como para autorizar e rastrear mudanças nesses requisitos.
- b) *Revisar os requisitos do produto de software*: deve-se poder revisar os requisitos relacionados aos contratos, engenharia, capacidade de manutenção e qualidade do software. Esse objetivo comporta três subobjetivos:
 - *Identificar os interesses da organização desenvolvedora no produto de software*: antes de concordar em fornecer um produto de software, é preciso identificar os padrões e procedimentos de *design* e desenvolvimento que devem ser usados, os itens que devem ser fornecidos pelo cliente e os métodos que deverão ser usados para avaliar a adequação dos itens que se espera que o cliente forneça.
 - *Avaliar riscos relacionados aos requisitos do produto*: antes de concordar em atender aos requisitos do produto, devem-se avaliar os riscos, os aspectos de segurança e confidencialidade, a experiência e a capacidade da organização desenvolvedora, a experiência e a capacidade dos fornecedores. Também devem-se avaliar criticamente os problemas detectados.
 - *Indicar alguém para representar o cliente*: deve-se solicitar ao cliente que indique alguém para dar suporte às atividades de desenvolvimento de software e gerenciar todas as responsabilidades contratuais. É necessário certificar-se de que os representantes do cliente tenham autoridade para garantir que o pessoal do cliente vai cooperar com a equipe de desenvolvimento.
- c) *Comunicar-se com seus clientes de software*: devem-se desenvolver e implementar os processos para controlar as comunicações com os clientes. É necessário certificar-se de que os processos controlam efetivamente os recados do/para o cliente. Esse objetivo também tem três subobjetivos:
 - *Comunicar-se consistentemente com seus clientes de software*: é necessário certificar-se de que os métodos de comunicação com o cliente são consistentes com os acordos contratuais.

- *Comunicar-se com os clientes durante o desenvolvimento:* devem-se programar revisões envolvendo tanto o cliente quanto a organização desenvolvedora (revisões conjuntas) para discutir informações sobre o desenvolvimento do produto de software e também indagações, contratos, termos aditivos e progresso.
- *Comunicar-se durante a operação e a manutenção do sistema:* é necessário comunicar-se com seus clientes durante o processo de operação e manutenção do software.

O design e desenvolvimento do software devem ser controlados:

- a) *Planejar o design e o desenvolvimento do produto:* devem ser definidos os estágios de *design* e desenvolvimento do produto de software, bem como procedimentos para controlá-los. Devem ser esclarecidas as responsabilidades e autoridades, e também gerenciadas as interações entre grupos de *design* e desenvolvimento. Os planos de *design* e desenvolvimento devem ser atualizados à medida que mudanças ocorrerem. Devem-se documentar as saídas de planejamento à medida que as mudanças ocorrerem. Esse objetivo comporta dois subobjetivos:
 - *Planejar o design e desenvolvimento do software:* devem ser identificadas as atividades que vão ser realizadas, incluindo as entradas, saídas, atividades de gerenciamento, serviços de suporte e treinamento de equipe necessários para cada atividade. Devem ser identificados os recursos de que o projeto vai necessitar, assim como as atividades de verificação e validação, regras e convenções de *design* e desenvolvimento, ferramentas e técnicas para desenvolvimento de software.
 - *Planejar atividades de revisão, verificação e validação:* devem-se usar procedimentos de manutenção ou acordos de serviços para planejar a revisão, a verificação e a validação da operação e a manutenção do software.
- b) *Definir as entradas para o design e o desenvolvimento do software:* devem-se especificar as entradas para o *design* e desenvolvimento do produto e suas definições. Devem-se avaliar as definições de entradas, e revisar as definições de entradas antes que elas sejam aprovadas. As entradas de *design* e desenvolvimento do software devem ser derivadas a partir dos requisitos funcionais, de *performance*, de qualidade, de segurança e de confidencialidade.
- c) *Gerar as saídas de design e desenvolvimento do software:* devem ser criadas as saídas de *design* e desenvolvimento do software. Antes da liberação, as saídas devem ser aprovadas. A saídas devem ser usadas para controlar a qualidade do produto de software. Deve-se manter um registro das saídas.
- d) *Realizar revisões de design e desenvolvimento do software:* devem-se realizar revisões ao longo do processo de *design* e desenvolvimento do produto de software. É necessário estabelecer procedimentos para especificar como os problemas identificados durante as revisões devem ser gerenciados, estabelecendo-se procedimentos que especifiquem como as deficiências do produto ou não conformidades e como as deficiências de processo ou não conformidades devem ser gerenciados. Além disso, é preciso formular orientações que todos os participantes deverão seguir durante as revisões de *design* e desenvolvimento do software.
- e) *Realizar verificações no design e no desenvolvimento do software:* devem ser realizadas atividades de verificação ao longo do processo de *design* e desenvolvimento do software, e registrados os resultados dessas atividades.
- f) *Realizar validações de design e desenvolvimento de software:* se necessário, as validações devem ser conduzidas em avaliações clínicas de dispositivos ligados à área médica e devem ser feitas avaliações de *performance* de dispositivos. Validações devem ser registradas. Esse objetivo comporta dois subobjetivos:
 - *Realizar atividades de validação do software:* a operação do produto de software deve ser validada antes que se peça ao cliente que o aceite formalmente, para confirmar que o novo produto satisfaz aos requisitos especificados no contrato e que satisfaz aos requisitos que definem seu uso pretendido. Essas validações devem ser registradas.
 - *Realizar atividades de teste de software:* os itens individuais de software e o produto completo devem ser testados de forma a garantir que eles satisfaçam requisitos operacionais.
- g) *Gerenciar mudanças no design e no desenvolvimento do software:* devem ser identificadas as mudanças no *design* e no desenvolvimento com o uso de um sistema de gerenciamento de configuração. Mudanças devem ser registradas, realizadas, verificadas, validadas e aprovadas antes de serem implementadas.

A produção e o fornecimento de serviços devem ser gerenciados:

- a) *Controlar a produção e o fornecimento de serviços:* devem ser controlados produção e serviços, versões e *releases*, atividades de replicação, entrega e instalação, operação e manutenção do software.

- b) *Validar a produção e o fornecimento de serviços:* devem-se validar os processos de produção e serviços sempre que as saídas não puderem ser mensuradas, monitoradas ou verificadas (processos especiais). Devem ser desenvolvidos métodos e procedimentos para provar que processos especiais podem produzir os resultados planejados. Devem ser mantidos registros que provem que processos especiais podem produzir os resultados planejados.
- c) *Identificar e rastrear os produtos:* os produtos de software devem ser identificados e rastreados utilizando-se um sistema de gerenciamento de configuração. Deve ser estabelecido um processo de rastreamento de software.
- d) *Proteger propriedades fornecidas por clientes:* devem-se identificar, verificar e salvaguardar propriedades fornecidas por clientes.
- e) *Preservar os produtos e componentes:* os produtos e componentes devem ser preservados durante o processamento interno e a entrega final, a partir de procedimentos estabelecidos.

Devem-se controlar os dispositivos de monitoramento:

- a) *Identificar as necessidades de medição e monitoramento:* devem-se identificar as necessidades de requisitos de medição e monitoramento, assim como as medições e o monitoramento que podem ser realizados para garantir que os produtos atendam aos requisitos especificados.
- b) *Selecionar dispositivos de medição e monitoramento:* devem ser selecionados dispositivos que sejam capazes de satisfazer às necessidades de medição e monitoramento da organização, assim como dispositivos que possam realizar o monitoramento e a medição de que a organização necessita.
- c) *Calibragem dos dispositivos de medição e monitoramento:* os dispositivos de medição e monitoramento devem ser calibrados pela comparação com dispositivos definidos por padrões nacionais ou internacionais. O status da calibração dos dispositivos deve ser identificado e registrado.
- d) *Proteger os dispositivos de medição e monitoramento:* o sistema de gerenciamento de configuração deve ser usado para controlar os dispositivos de medição e monitoramento, de forma que sejam protegidos contra ajustes não autorizados, danos ou deterioração.
- e) *Validar os dispositivos de medição e monitoramento:* os dispositivos de medição e monitoramento devem ser validados antes de serem usados e revalidados quando necessário.
- f) *Usar os dispositivos de medição e monitoramento:* os dispositivos devem ser usados para garantir que os produtos atendam aos requisitos.

12.1.5 REQUISITOS E ORIENTAÇÕES PARA AÇÕES CORRETIVAS

Devem ser realizados *processos corretivos*:

- a) *Planejar processos corretivos:* deve-se planejar como o monitoramento, a medição e os processos analíticos serão usados para demonstrar conformidade. Também deve ser planejado como eles serão usados para manter e melhorar continuamente a efetividade do sistema de gerenciamento de qualidade.
- b) *Implementar processos corretivos:* deve-se usar o monitoramento, a medição e os processos analíticos para demonstrar conformidade, manter o sistema de qualidade e melhorar continuamente a efetividade do sistema de qualidade.

A *qualidade* deve ser monitorada e medida:

- a) *Monitorar e medir a satisfação do cliente:* devem ser identificados métodos que possam ser usados para monitorar e medir a satisfação do cliente. Informações sobre a satisfação do cliente devem ser usadas como uma medida de desempenho do sistema de gerenciamento da qualidade. Isso pode ser feito pela análise de chamadas ao atendimento *help desk* e pelo estudo de métricas de qualidade de uso derivadas de *feedback* direto e indireto do cliente.
- b) *Planejar e realizar auditorias internas regulares:* um programa de auditoria interna deve ser definido e desenvolvido. Projetos de auditoria interna devem ser planejados, e também deve-se planejar como os projetos de software serão auditados. Auditorias internas devem ser realizadas regularmente. Problemas descobertos pela auditoria devem ser resolvidos, e deve-se dispor de meios para verificar se os problemas foram efetivamente resolvidos.

- c) *Monitorar e medir processos de qualidade:* devem-se usar métodos adequados para monitorar e medir processos. Ações devem ser realizadas quando processos falham em obter os resultados planejados.
- d) *Monitorar e medir características do produto:* deve-se monitorar e medir quanto bem os produtos de software atendem aos requisitos de qualidade. Deve ser mantido o registro das atividades de monitoramento e medição dos produtos de software.

A não conformidade de produtos de software deve ser controlada:

- a) *Estabelecer um procedimento de não conformidade de produtos de software:* deve-se definir como essa não conformidade deve ser identificada e controlada.
- b) *Identificar e controlar a não conformidade de produtos de software:* deve-se eliminar ou corrigir as não conformidades de produtos de software, prevenir a entrega e o uso de produtos de software com não conformidades, evitar o uso inapropriado de produtos de software com não conformidades. Pode-se permitir concessões apenas se os requisitos regulatórios forem satisfeitos.
- c) *Reverificar a não conformidade de produtos de software que foram corrigidos:* deve-se pelo menos provar que os produtos de software atendem aos requisitos depois de corrigidos.
- d) *Controlar produtos de software não conformados após entrega ou uso:* devem-se controlar eventos nos quais foram disponibilizados ou usados produtos não conformados. Deve-se desenvolver uma instrução de trabalho para controlar o processo de retrabalho dos produtos.
- e) *Manter o registro de produtos de software não conformados:* devem-se documentar as não conformidades do produto de software, descrever as ações realizadas para lidar com as não conformidades, manter o registro de concessões dadas para produtos de software.

Devem-se analisar as informações sobre a qualidade:

- a) *Definir necessidades de gerenciamento de informação sobre qualidade:* deve ser identificado o tipo de informação de que a organização necessita para o sistema de gerenciamento de qualidade, assim como definido o tipo de informação necessário para que se possam avaliar a adequação e a efetividade do sistema de gerenciamento da qualidade.
- b) *Coletar dados para o sistema de gerenciamento da qualidade:* devem-se monitorar e medir a adequação e a efetividade do sistema de gerenciamento da qualidade.
- c) *Fornecer informações para gerenciamento da qualidade:* devem ser fornecidas informações sobre clientes, fornecedores, produtos e processos, bem como desenvolvidos procedimentos para analisar a informação e manter o registro dos resultados analíticos.

As ações corretivas necessárias devem ser realizadas:

- a) *Manter o sistema de controle de qualidade:* devem ser usadas auditorias, dados de qualidade, política de qualidade, objetivos de qualidade, revisões de gerenciamento, ações corretivas e preventivas para ajudar a manter a efetividade do sistema de qualidade.
- b) *Corrigir as não conformidades reais:* as não conformidades devem ser revisadas e deve-se detectar o que as causou. Deve-se avaliar se é necessário realizar alguma ação corretiva. Devem-se desenvolver ações corretivas para prevenir a recorrência. Devem ser realizadas ações corretivas sempre que necessário e registrar os resultados obtidos por elas. Deve ser examinada a efetividade das ações corretivas.
- c) *Prevenir não conformidades potenciais:* devem ser detectadas as não conformidades potenciais e identificadas as suas causas. Os efeitos das ações preventivas devem ser estudados. Deve-se avaliar se é necessário realizar ações preventivas. Devem ser desenvolvidas ações preventivas para eliminar as causas identificadas. Devem ser tomadas ações preventivas sempre que necessário e registrados os resultados obtidos por elas. Deve ser examinada a efetividade das ações preventivas.

Um exemplo de *checklist* para avaliar o grau de incorporação dos processos relacionados a ações preventivas na empresa pode ser encontrado na internet⁴.

⁴O conjunto completo de *checklists* pode ser adquirido no site <www.praxiom.com/iso-90003-sample.pdf>. Acesso em: 21 jan. 2013.

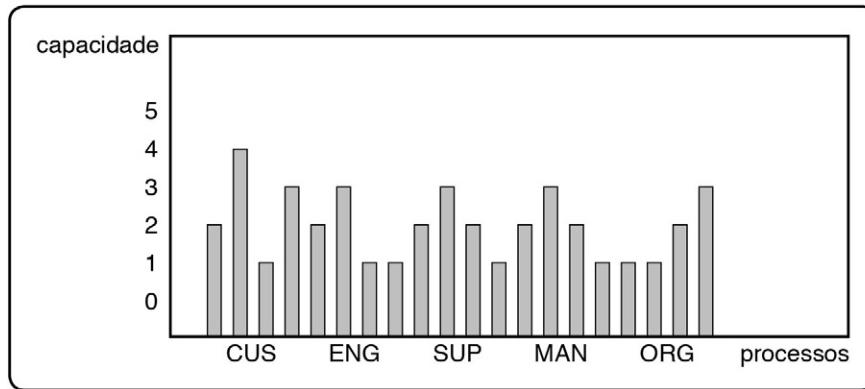


Figura 12.1 As duas dimensões de avaliação do SPICE.

12.2 ISO/IEC 15504 – SPICE

A Norma ISO/IEC 15504, também conhecida como SPICE (Zahran, 1997) ou *Software Process Improvement and Capability dEtermination*, foi criada como uma complementação para a ISO/IEC 12207 (definição de processos do ciclo de vida de desenvolvimento de software) e tem como objetivo orientar a avaliação e a autoavaliação da capacidade de empresas em processos e, a partir dessa avaliação, permitir a melhoria dos processos.

O aperfeiçoamento contínuo do sistema de qualidade é uma exigência da ISO 9001, mas a Norma 9001 não mostra o caminho para atingir esse objetivo. Por alguns anos, a ISO 9000-3 (atualmente substituída pela 90003) foi usada como referência para a indústria de software, mas também não abordava a questão de níveis de capacidade e melhoria contínua. Assim, o SPICE foi, por algum tempo, o modelo de referência preferido para as empresas que desejavam melhorar seus processos (Emam, Melo & Drouin, 1997), sendo usado como complemento à norma 9000-3.

A Norma 15504 se estrutura em duas dimensões (Figura 12.1):

- a) *Dimensão de processos*: quais processos são avaliados.
- b) *Dimensão de capacidade*: qual é a capacidade da empresa avaliada em cada um desses processos.

Existem cinco grandes categorias dentro da dimensão de processos. Esses processos são fortemente alinhados com as definições da Norma ISO/IEC 12207. Dependendo da referência bibliográfica, o número de processos pode variar, pois a lista tem evoluído com o passar do tempo. A lista a seguir é baseada em Emam, Melo & Drouin (1997):

- a) CUS: relação cliente/fornecedor, que inclui:
 - CUS.1 – Aquisição de software.
 - CUS.2 – Gerenciamento das necessidades do cliente.
 - CUS.3 – Fornecimento de software.
 - CUS.4 – Operação de software.
 - CUS.5 – Fornecimento de serviços ao usuário.
- b) ENG: processos de engenharia, que inclui:
 - ENG.1 – Desenvolvimento dos requisitos do sistema e do projeto.
 - ENG.2 – Desenvolvimento dos requisitos do software.
 - ENG.3 – Desenvolvimento do projeto do software.
 - ENG.4 – Implementação do projeto do software.
 - ENG.5 – Integração e teste do software.
 - ENG.6 – Integração e teste do sistema.
 - ENG.7 – Manutenção do sistema e do software.
- c) SUP: processos de suporte, que inclui:
 - SUP.1 – Desenvolvimento de documentação.
 - SUP.2 – Gerenciamento de configuração.
 - SUP.3 – Assegurar qualidade.

- SUP.4 – Verificar o produto do trabalho.
 - SUP.5 – Validar o produto do trabalho.
 - SUP.6 – Revisar conjuntamente.
 - SUP.7 – Realizar auditorias.
 - SUP.8 – Resolver problemas.
- d) MAN: processos de gerência, que inclui:
- MAN.1 – Gerenciamento do projeto.
 - MAN.2 – Gerenciamento da qualidade.
 - MAN.3 – Gerenciamento de riscos.
 - MAN.4 – Gerenciamento de subcontratados.
- e) ORG: processos de organização, que incluem:
- ORG.1 – Engenharia de negócio.
 - ORG.2 – Definição dos processos.
 - ORG.3 – Melhoria dos processos.
 - ORG.4 – Fornecimento de recursos humanos capacitados.
 - ORG.5 – Fornecimento de infraestrutura de engenharia de software.

Cada processo incluído em cada uma das cinco categorias é definido por um objetivo e tem saídas definidas. Por exemplo, o processo CUS.1, aquisição de software, tem como objetivo obter um *produto* ou *serviço* que satisfaça à necessidade do cliente e como saída desejada um *contrato* que expresse claramente as expectativas, responsabilidades e obrigações do cliente e do fornecedor.

Já os níveis de capacidade são seis:

- a) 0 – *incompleto*: esse nível representa uma falha geral em se atingir aos objetivos de um processo ou à ausência dele. Não há produtos e saídas facilmente identificáveis para o processo sendo avaliado. Pode ser atribuído quando o processo não é implementado, ou quando é implementado, mas falha em atingir seus objetivos.
- b) 1 – *processo realizado*: nesse nível, o propósito do processo geralmente é obtido, mas não necessariamente de forma planejada ou rastreável.
- c) 2 – *processo gerenciado*: nesse nível, os projetos entregam produtos com qualidade aceitável dentro dos prazos e orçamento definidos. A execução dos projetos de acordo com a definição dos processos é realizada e rastreável.
- d) 3 – *processo estabelecido*: nesse nível, a própria gerência dos projetos deve ser realizada de acordo com um processo estabelecido, no qual bons princípios de engenharia de software são empregados. Esse nível, ao contrário do anterior, necessita de um gerenciamento planejado e da utilização de um processo padrão.
- e) 4 – *processo previsível*: nesse nível, os projetos são realizados de forma consistente dentro de limites de controle. Medidas de *performance* detalhadas são coletadas e analisadas, o que leva à compreensão quantitativa da capacidade do processo e à melhor habilidade de prever *performances* futuras. Nesse caso, a *performance* é gerenciada de forma objetiva e a qualidade do trabalho é quantitativamente conhecida.
- f) 5 – *processo otimizado*: nesse nível, a realização do processo é otimizada para satisfazer às necessidades correntes e futuras do negócio, e os processos satisfazem repetidamente essas necessidades. Metas quantitativas de eficiência e efetividade para processos são estabelecidas. O monitoramento contínuo e eficaz permite a melhoria contínua do processo a partir da análise de resultados. Administrar um processo envolve a incorporação constante de novas ideias e tecnologias, bem como a modificação de processos ineficientes ou não efetivos.

A avaliação dos níveis de capacidade é demonstrada em função de um conjunto de atributos de processos. Cada nível tem seus próprios atributos, e os atributos são avaliados de acordo com uma escala de obtenção, que fornece uma medida da capacidade da empresa no processo sendo avaliado. Os valores possíveis na escala de obtenção dos atributos são os seguintes:

- a) *N – not achieved* (0%-15%): não há evidência de que o atributo tenha sido obtido.
- b) *P – partially achieved* (>15%-50%): o atributo foi parcialmente obtido.
- c) *L – largely achieved* (>50%-85%): o atributo foi amplamente obtido.
- d) *F – fully achieved* (>85%-100%): o atributo foi totalmente obtido.

Para que uma empresa tenha determinado processo avaliado em um nível N, é necessário que ela obtenha pelo menos escala L nos atributos do nível N e escala F nos atributos de todos os níveis anteriores.

O nível 0, ou *incompleto*, não tem atributos. Ele corresponde ao estado inicial de qualquer empresa que nunca tenha implementado processos sistemáticos.

No nível 1, ou *realizado*, o único atributo de processo é o seguinte:

- a) *PA1.1 – Atributo de realização do processo*: extensão na qual a execução dos projetos segue as práticas definidas no processo. Nesse caso, deve haver entradas e saídas bem definidas nas tarefas dos projetos.

No nível 2, ou *planejado e rastreado*, os atributos de processo são:

- a) *PA2.1 – Atributo de gerenciamento de performance*: extensão na qual a realização dos projetos é gerenciada para produzir os produtos de acordo com os prazos e recursos.
- b) *PA2.2 – Atributo de gerenciamento do produto do trabalho*: extensão na qual a realização dos projetos é gerenciada para produzir produtos que satisfaçam aos requisitos funcionais e não funcionais, dentro de padrões de qualidade definidos.

No nível 3, ou *estabelecido*, os atributos de processo são:

- a) *PA3.1 – Atributo de definição de processo*: extensão na qual a execução de um projeto usa uma definição de processo baseada em um processo padrão ou modelo de processo.
- b) *PA3.2 – Atributo de recursos de processo*: extensão na qual a execução de projetos usa recursos humanos capacitados e infraestrutura de processos que efetivamente contribuem para atingir os objetivos da organização.

No nível 4, ou *previsível*, os atributos de processo são:

- a) *PA4.1 – Atributo de medição de processo*: extensão na qual o processo conta com o suporte de medições que garantem que a implementação do processo contribua para o alcance das metas.
- b) *PA4.2 – Atributo de controle de processo*: extensão na qual a execução dos projetos é controlada através da coleta e da análise das medidas para controlar e corrigir, onde necessário, a *performance* do processo.

No nível 5, ou *otimizado*, os atributos de processo são:

- a) *PA5.1 – Atributo de mudança de processo*: extensão na qual mudanças na definição, gerenciamento e *performance* do processo são mais bem controladas para atingir as metas de negócio da organização.
- b) *PA5.2 – Atributo de melhoria contínua*: extensão na qual mudanças no processo são identificadas e implementadas para garantir a melhoria contínua no preenchimento das metas de negócio definidas para a organização.

12.2.1 PROCESSO DE AVALIAÇÃO

A própria Norma 15504 apresenta um guia para avaliação dos processos de uma empresa. Esse guia inclui um modelo e um processo de avaliação, bem como ferramentas para proceder a avaliação. Uma avaliação pode ser feita, basicamente, com dois objetivos:

- a) *Determinar capacidade*: uma organização que deseja terceirizar a produção de software pode querer saber ou avaliar a capacidade de potenciais fornecedores em diferentes áreas de processo.
- b) *Melhoria de processo*: uma organização que desenvolve software pode querer melhorar seus próprios processos. A norma possibilita avaliar o estado atual dos processos da empresa e, após as atividades de melhoria, avaliar se houve avanço.

O *modelo de avaliação* usado é a estrutura dos processos (CUS, ENG, SUP, MAN e ORG), conforme apresentado na Norma 12207. SPICE permite que outros modelos de processo sejam adotados. Porém, a adoção do modelo 12207 facilita a avaliação, pois existe bastante material disponível, e, com esse padrão, é possível realizar comparativos entre empresas.

O *processo de avaliação* é definido na norma e inclui as seguintes atividades:

- a) Iniciar a avaliação por parte do interessado.
- b) Selecionar o avaliador e sua equipe.

- c) Planejar a avaliação, selecionando os processos que serão avaliados de acordo com o modelo e a demanda do interessado.
- d) Reunião de pré-avaliação.
- e) Coletar dados.
- f) Validar dados.
- g) Atribuir nível de capacidade aos processos.
- h) Relatar os resultados da avaliação.

A consulta à norma, porém, não é suficiente para que o processo de avaliação ocorra, já que o método não é suficientemente detalhado. Para selecionar avaliadores, é necessário contratar pessoas com treinamento específico em avaliação SPICE, o que inclui, no mínimo, a conclusão de um curso de cinco dias e a experiência de ter avaliado uma empresa com sucesso sob a supervisão de um avaliador experiente. Existem também orientações sobre como um avaliador experiente deve avaliar o avaliador candidato para certificar sua capacidade (Van Loon, 2007).

O avaliador pode coletar dados de diversas maneiras, incluindo entrevistas, análise de dados estatísticos e registros de qualidade. Em geral, a falta de registros confiáveis não é positiva para a atribuição de níveis de capacidade em processos. Porém, mesmo quando os registros existem, o avaliador deve validá-los, isto é, certificar-se de que são corretos. Isso, em geral, pode ser feito através das entrevistas.

A partir de sua experiência e das diretrizes de avaliação, o avaliador vai, então, atribuir um nível de obtenção (F/L/P/N) a cada um dos atributos de processo, iniciando pelos atributos dos níveis mais baixos e subindo. À medida que os atributos são avaliados com F, o avaliador pode subir mais um nível. No momento em que encontrar atributos com nota inferior a F, ele para e atribui o nível de capacidade determinado pelos atributos (Tabela 12.1).

Na tabela, o processo CUS.1 foi avaliado no nível 3 porque possui L nos atributos do nível 3 e F nos atributos dos níveis 1 e 2. O processo CUS.2 foi avaliado no nível 0 porque não atingiu sequer L nos atributos do nível 1. O processo ORG.5 atingiu nível 2 porque, embora tenha F nos atributos dos níveis 1 e 2, não atingiu pelo menos L nos atributos do nível 3.

12.2.2 DETALHAMENTO DA DIMENSÃO DE PROCESSOS

Todos os processos dos cinco grupos SPICE são definidos a partir de um objetivo e uma saída esperada (Emam, Melo & Drouin, 1997).

Os processos que definem a relação com o cliente (CUS) são determinados da seguinte forma:

- a) CUS.1 – Aquisição de software
 - Propósito: obter um produto ou serviço que satisfaça à necessidade do cliente.
 - Saída esperada: um contrato que claramente expresse as expectativas, responsabilidades e obrigações do cliente e do fornecedor.

TABELA 12.1 Exemplo de avaliação SPICE

| | Processos | | | |
|------------------|-----------|-------|-----|-------|
| Atributo | CUS.1 | CUS.2 | ... | ORG.5 |
| PA5.2 | | | | |
| PA5.1 | | | | |
| PA4.2 | | | | |
| PA4.1 | | | | |
| PA3.2 | L | | | |
| PA3.1 | L | | | P |
| PA2.2 | F | | | F |
| PA2.1 | F | | | F |
| PA1.1 | F | P | | F |
| Nível atribuído: | 3 | 0 | ... | 2 |

b) CUS.2 – Gerenciamento das necessidades do cliente

- Propósito: gerenciar as mudanças nos requisitos. Estabelecer uma base de requisitos e gerenciar as mudanças sobre essa base.
- Saídas esperadas: canais de comunicação clara com o cliente, requisitos como base para o projeto, um mecanismo de monitoramento de mudanças nos requisitos e um mecanismo que garanta ao cliente fácil acesso ao *status* de suas solicitações.

c) CUS.3 – Fornecimento de software

- Propósito: empacotar, entregar e instalar o software na empresa do cliente.
- Saídas esperadas: requisitos para empacotamento, entrega e instalação determinados, empacotamento que facilite a instalação e a operação efetiva e eficiente, e software de qualidade instalado de acordo com os requisitos.

d) CUS.4 – Operação de software

- Propósito: dar suporte à operação correta e eficiente do software.
- Saídas esperadas: identificação e gerenciamento dos riscos operacionais para a introdução e operação do software, execução do software no seu ambiente operacional de acordo com procedimentos documentados, fornecimento de suporte operacional para resolver problemas operacionais, gerenciamento das solicitações do usuário e certificação de que as capacidades do software e do sistema hospedeiro sejam adequadas.

e) CUS.5 – Fornecimento de serviços ao usuário

- Propósito: estabelecer e manter um nível aceitável de serviço ao usuário para dar suporte ao uso efetivo do software.
- Saídas esperadas: identificação das necessidades de suporte ao usuário de forma continuada, verificação continuada da satisfação do usuário com os serviços fornecidos e o produto, satisfação das necessidades do usuário em termos de entrega de serviços apropriados.

Os processos de engenharia (ENG) são assim definidos:

a) ENG.1: Desenvolvimento dos requisitos do sistema e do projeto

- Propósito: estabelecer os requisitos funcionais e não funcionais do sistema e uma arquitetura que identifique quais requisitos devem ser alocados a quais elementos do sistema.
- Saídas esperadas: desenvolvimento dos requisitos do sistema, proposta de uma solução efetiva que identifique os principais elementos do sistema, alocação dos requisitos definidos aos principais elementos do sistema, desenvolvimento de estratégias de liberação que definam as prioridades de implementação dos requisitos do sistema e comunicação dos requisitos, solução proposta e suas relações com as partes afetadas.

b) ENG.2: Desenvolvimento dos requisitos do software

- Propósito: estabelecer os requisitos do componente “software” do sistema.
- Saídas esperadas: definição dos requisitos do software e suas interfaces, desenvolvimento de requisitos de software analisados, corretos e testáveis, compreensão do impacto dos requisitos do software no ambiente operacional, desenvolvimento de uma estratégia de liberação que defina as prioridades na implementação dos requisitos do software, aprovação dos requisitos e atualização, quando necessário, e comunicação dos requisitos com as partes afetadas.

c) ENG.3: Desenvolvimento do *design* do software

- Propósito: definir um *design* para o software que acomode os requisitos e que possa ser testado contra eles.
- Saídas esperadas: desenvolvimento de um *design* de arquitetura que descreva os grandes componentes do software e acomode os requisitos, definição das interfaces internas e externas de cada componente do software, desenvolvimento de um *design* detalhado que descreva as unidades de software que podem ser construídas e testadas e estabelecimento de mapeamento entre requisitos e *design* de software.

d) ENG.4: Implementar o projeto do software

- Propósito: produzir unidades de software executáveis e verificar se elas refletem o projeto do software.
- Saídas esperadas: definição de critérios de verificação para todas as unidades de software contra os requisitos, produção de unidades de software definidas pelo *design* e verificação das unidades de software contra o *design*.

e) ENG.5: Integrar e testar o software

- Propósito: integrar as unidades de software e produzir software que satisfaça aos requisitos.
- Saídas esperadas: estratégia de integração das unidades de software consistente com o plano de liberação, critérios de aceitação para agregados que verifiquem a satisfação dos requisitos alocados às unidades, verificação dos agregados de software e do software integrado usando critérios de aceitação definidos, registro dos resultados dos testes e desenvolvimento de uma estratégia de regressão para retestar agregados ou o software integrado se forem feitas mudanças nos componentes.

f) ENG.6: Integrar e testar o sistema

- Propósito: integrar os componentes do software com outros componentes, como operações manuais ou hardware, produzindo um sistema completo que satisfaça aos requisitos do sistema.
- Saídas esperadas: desenvolvimento de um plano de integração para construir unidades agregadas do sistema de acordo com a estratégia de liberação, definição de critérios de aceitação para cada agregado contra os requisitos do sistema alocados às unidades, verificação dos agregados usando os critérios de aceitação definidos, construção de um sistema integrado demonstrando atendimento aos requisitos, registro dos resultados dos testes e desenvolvimento de uma estratégia de regressão para retestar agregados ou o sistema integrado, se for necessário modificar os componentes.

g) ENG.7: Manter o sistema e o software

- Propósito: gerenciar a modificação, a migração e a aposentadoria de componentes do sistema em resposta aos requisitos do usuário, preservando a integridade das operações organizacionais.
- Saídas esperadas: definição do impacto no sistema existente das operações, interfaces e organização, atualização de especificações, documentos de projeto e planos de teste, desenvolvimento de componentes do sistema modificados com documentação associada e testes para demonstrar que os requisitos não são comprometidos, migração do sistema e software atualizado para o ambiente do usuário e software, e sistemas retirados de uso de forma controlada, minimizando a perturbação aos usuários.

Os *processos de suporte* (SUP) são assim definidos:

a) SUP.1: Desenvolvimento de documentação

- Propósito: desenvolver e manter documentos que registrem a informação produzida.
- Saídas esperadas: identificação de todos os documentos a serem produzidos, especificação do conteúdo e do propósito dos documentos, identificação dos padrões a serem aplicados aos documentos, desenvolvimento dos documentos publicados de acordo com os padrões e manutenção dos documentos de acordo com critérios especificados.

b) SUP.2: Gerenciamento de configuração

- Propósito: estabelecer e manter a integridade de todos os produtos do trabalho.
- Saídas esperadas: identificação, definição e embasamento de todos os itens relevantes gerados pelo processo ou produto, controle de modificações e versões, registro e relatório de *status* dos itens e modificações requeridas, garantia de completude e consistência dos itens, controle do armazenamento, manipulação e entrega dos itens.

c) SUP.3: Assegurar qualidade

- Propósito: garantir que os produtos do trabalho estejam de acordo com padrões, procedimentos e requisitos aplicáveis.
- Saídas esperadas: identificação, planejamento e cronograma de atividades de garantia de qualidade nas atividades do processo ou projeto, identificação de padrões, métodos, procedimentos e ferramentas de qualidade para realizar as atividades que visam assegurar qualidade, habilidade dos responsáveis pela qualidade ao realizar seus deveres, independentemente da gerência, e realização das atividades que visam assegurar qualidade de acordo com os planos e cronogramas.

d) SUP.4: Verificar o produto do trabalho

- Propósito: garantir que cada produto do trabalho reflita os requisitos de sua construção.
- Saídas esperadas: identificação de critérios para verificação de todos os produtos do trabalho requeridos, realização das atividades de verificação requeridas, localização e remoção eficiente de defeitos dos produtos produzidos pelo projeto.

e) SUP.5: Validar o produto do trabalho

- Propósito: garantir que os requisitos especificados para o uso pretendido do produto do trabalho sejam efetivados.
- Saídas esperadas: identificação de critérios para validação, realização das atividades de validação e fornecimento de evidência de que os produtos do trabalho são adequados para o uso pretendido.

f) SUP.6: Revisar conjuntamente

- Propósito: manter um entendimento comum com o cliente do progresso contra os objetivos do contrato.
- Saídas esperadas: avaliação do *status* dos produtos através de revisões conjuntas, planejamento e cronograma de revisões conjuntas a serem feitas, rastreamento para concluir ações resultantes das revisões.

g) SUP.7: Realizar auditorias

- Propósito: confirmar, de forma independente, que os produtos e processos empregados estão de acordo com os requisitos específicos definidos.
- Saídas esperadas: avaliação da adequação do produto aos requisitos, planos e contrato, arranjo e condução de auditorias independentes.

h) SUP.8: Resolver problemas

- Propósito: garantir que todos os problemas descobertos sejam analisados e removidos.
- Saídas esperadas: fornecimento de um meio rápido, responsável e documentado para garantir que todos os problemas sejam analisados e removidos, fornecimento de um mecanismo para reconhecer e agir sobre tendências em problemas identificados.

Os processos referentes ao *gerenciamento de projetos* (MAN) são assim definidos:

a) MAN.1: Gerenciamento do projeto

- Propósito: definir os processos necessários para estabelecer, coordenar e gerenciar um projeto e seus recursos.
- Saídas esperadas: definição do escopo de trabalho do projeto, estimativa e planejamento das tarefas e recursos necessários, identificação e gerenciamento das interfaces entre os elementos do projeto e entre o projeto e outras unidades da organização, realização de ações corretivas quando os objetivos de um projeto não estão sendo atingidos.

b) MAN.2: Gerenciamento da qualidade

- Propósito: gerenciar a qualidade dos produtos e processos do projeto de forma a satisfazer o cliente.
- Saídas esperadas: estabelecimento de metas de qualidade baseadas nos requisitos de qualidade do cliente para vários *checkpoints* durante o desenvolvimento do projeto, definição e uso de métricas para medir os resultados das atividades nos *checkpoints*, identificação sistemática de boas práticas de engenharia de software e sua integração ao ciclo de vida, realização das atividades de qualidade e confirmação de sua *performance* e realização de ações corretivas quando os objetivos de qualidade não são atingidos.

c) MAN.3: Gerenciamento de riscos

- Propósito: identificar e reduzir continuamente os riscos do projeto.
- Saídas esperadas: determinação do escopo do gerenciamento de riscos, identificação dos riscos no projeto à medida que ele se desenvolve, análise dos riscos e determinação da prioridade de aplicação de recursos para gerenciar esses riscos, definição, implementação e verificação de estratégias de gerenciamento de riscos, definição e aplicação de técnicas de risco e realização de ações corretivas quando o progresso esperado não é obtido.

d) MAN.4: Gerenciamento de subcontratados

- Propósito: selecionar subcontratados qualificados e gerenciar sua *performance*.
- Saídas esperadas: estabelecimento de uma especificação do trabalho a ser realizado por subcontratação, qualificação de subcontratados em potencial através de uma verificação de sua capacidade, estabelecimento e gerenciamento de compromissos de/para com o subcontratado, troca regular de informações sobre progresso técnico com o subcontratado, verificação da adequação do subcontratado em relação aos padrões e procedimentos acordados, verificação da qualidade dos produtos e serviços do subcontratado.

Os *processos da organização* (ORG) são assim definidos:

a) ORG.1: Engenharia de negócio

- Propósito: fornecer aos indivíduos da organização e dos projetos uma visão e uma cultura que os ajudem a funcionar efetivamente.

- Saídas esperadas: definição de uma visão, missão e objetivos do negócio que seja divulgada a todos os empregados, que se torna um desafio a cada indivíduo para garantir que seu trabalho seja realizado de forma a contribuir com a visão do negócio.

b) ORG.2: Definição dos processos

- Propósito: estabelecer um conjunto de processos para o ciclo de vida de software, incluindo uma estratégia para sua adaptação.
- Saídas esperadas: definição de metas, identificação de atividades, papéis e responsabilidades atuais, identificação de entradas e saídas, definição de critérios para iniciar e finalizar o processo, definição de pontos de controle (revisões e decisões-chave), identificação de interfaces externas (outros processos) e de interfaces internas (entre atividades do processo), definição de registros de qualidade (para demonstrar conformidade) e de medidas do processo (para verificar se as metas foram alcançadas), documentação do processo padrão, estabelecimento de políticas, estabelecimento de expectativas de desempenho e divulgação do processo.

c) ORG.3 – Melhoria dos processos

- Propósito: aperfeiçoar continuamente a eficácia e a eficiência de processos da organização, à luz de seus objetivos e necessidades de negócio.
- Saídas esperadas: identificação de oportunidades de melhoria (análise de métricas e medidas, *benchmarks* e alterações nos requisitos), definição de escopo das atividades de melhoria, entendimento do processo (forças e fraquezas), identificação de melhorias, priorização de melhorias, definição de medidas de impacto, alteração (melhoria) dos processos, confirmação das melhorias (através de testes) e divulgação das melhorias.

d) ORG.4 – Fornecimento de recursos humanos capacitados

- Propósito: fornecer à organização indivíduos capacitados para exercer seus papéis como indivíduos e em grupo.
- Saídas esperadas: identificação de necessidades de treinamento, desenvolvimento ou aquisição de treinamento, treinamento de pessoal e manutenção de registros de treinamento.

e) ORG.5 – Fornecimento de infraestrutura de engenharia de software

- Propósito: manter, de forma estável e confiável, a infraestrutura necessária para apoiar a execução de outros processos.
- Saídas esperadas: determinação de estratégia organizacional para reúso, identificação de componentes reusáveis, desenvolvimento de componentes reusáveis, estabelecimento de uma biblioteca de reúso, certificação de componentes reusáveis (antes de colocar na biblioteca), integração de reúso no ciclo de vida, propagação cuidadosa das alterações, identificação de requisitos para ambiente, fornecimento de um ambiente de desenvolvimento de software, fornecimento de suporte aos desenvolvedores, manutenção do ambiente, fornecimento de espaço de trabalho produtivo, integridade de dados (resultantes do projeto de software) assegurada, *backups* providenciados, fornecimento de instalações de trabalho (salas, comunicação) e fornecimento de acesso remoto.

Atualmente, embora a SPICE ainda não esteja obsoleta, o modelo CMMI tem sido mais popular, possivelmente porque, ao contrário das normas ISO, a descrição completa de CMMI pode ser obtida gratuitamente.

12.3 CMMI – *Capability Maturity Model Integration*

O CMMI (*Capability Maturity Model Integration*) é uma abordagem para a melhoria de processos compatível com a Norma ISO 15504 (SPICE). Embora as duas abordagens sejam semelhantes, o CMMI não foi baseado em SPICE, como se poderia pensar. O modelo foi construído de forma independente, com participação da indústria, do governo norte-americano e do Instituto de Engenharia de Software (SEI) da Carnegie Mellon University (CMU).

CMMI é o sucessor do modelo CMM (*Capability Maturity Model*), que foi desenvolvido entre 1987 e 1997. Em 2002 foi lançada a versão 1.1 do CMMI, e, em novembro de 2010, a versão 1.3. Documentação extensiva e comprehensiva sobre o CMMI 1.3 pode ser obtida sem custo no site do SEI⁵.

⁵Disponível em: <www.sei.cmu.edu/cmmi/tools/cmmiv1-3/>. Acesso em: 21 jan. 2013.

TABELA 12.2 Níveis de capacidade e maturidade do CMMI⁹

| Nível | Capacidade | Maturidade |
|-------|------------|------------------------------|
| 0 | Incompleto | |
| 1 | Realizado | Inicial |
| 2 | Gerenciado | Gerenciado |
| 3 | Definido | Definido |
| 4 | | Quantitativamente gerenciado |
| 5 | | Em otimização |

A versão atual do CMMI possui três vertentes:

- a) CMMI-ACQ⁶, para aquisição de produtos e serviços.
- b) CMMI-DEV⁷, para o desenvolvimento de produtos e serviços.
- c) CMMI-SVC⁸, para estabelecimento, gerenciamento e oferta de serviços.

Os modelos CMMI podem ser usados como guias para desenvolver e melhorar processos da organização, e também como um *framework* para avaliar a maturidade dos processos da organização.

CMMI se originou na indústria de software, mas também tem sido adaptado a outras áreas, como a indústria de hardware, serviços e comércio em geral. O termo “software” sequer aparece nas definições de CMMI, o que torna o modelo bem mais abrangente do que seu predecessor, o CMM.

Existem duas representações do CMMI, a *representação contínua* e a *representação em estágios*. A *representação contínua* é projetada para permitir à empresa focar em processos específicos que deseja melhorar em função de suas prioridades. Já a *representação em estágios* é aplicada à organização como um todo e permite que se compare a maturidade de diferentes organizações.

A avaliação pela representação contínua mede a *capacidade* da empresa em relação a um ou mais processos. Já a avaliação em estágios mede a *maturidade* da empresa.

Existem quatro níveis de capacidade e cinco níveis de maturidade. A Tabela 12.2 apresenta esses níveis e suas correspondências. Note que não existe nível 0 de maturidade, nem existem níveis 4 e 5 de capacidade.

Existe um alinhamento explícito entre os seis níveis SPICE e os níveis de capacidade e maturidade CMMI, e sua interpretação é bastante semelhante.

12.3.1 PRÁTICAS E OBJETIVOS ESPECÍFICOS E GENÉRICOS

CMMI identifica dois tipos de objetivos para processos: *específicos* e *genéricos*.

Objetivos específicos são características únicas que devem ser apresentadas para que determinada área de processo seja satisfeita. Ou seja, cada objetivo específico aplica-se a uma única área de processo. Um exemplo de objetivo específico da área de gerenciamento de configuração é: “A integridade das *baselines* é estabelecida e mantida”.

Já os *objetivos genéricos (generic)* têm esse nome porque se aplicam a várias áreas de processo. Um objetivo genérico descreve características que devem estar presentes para institucionalizar processos que compõem uma área de processo. Um exemplo de objetivo genérico é: “O processo é institucionalizado como processo definido”.

Uma *prática específica* é uma atividade considerada importante para a obtenção de um objetivo específico, ou seja, a prática vai indicar o que deve ser feito, em termos de ações, para que um objetivo específico seja atingido.

Da mesma forma, o CMMI descreve *práticas genéricas*, que são atividades relacionadas com os objetivos genéricos.

O modelo ainda faz um grande detalhamento das práticas, apresentando *subpráticas*, que são ações específicas e localizadas que servem de guia para a implementação das práticas e o alcance dos objetivos.

⁶Disponível em: <www.sei.cmu.edu/library/abstracts/reports/10tr032.cfm>. Acesso em: 21 jan. 2013.

⁷Disponível em: <www.sei.cmu.edu/library/abstracts/reports/10tr033.cfm>. Acesso em: 21 jan. 2013.

⁸Disponível em: <www.sei.cmu.edu/library/abstracts/reports/10tr034.cfm>. Acesso em: 21 jan. 2013.

⁹SEI (2010).

12.3.2 NÍVEIS DE CAPACIDADE

Existem, portanto, quatro níveis de capacidade para processos no CMMI. Um nível de capacidade é atingido quando os objetivos genéricos daquele nível são atingidos. Os níveis são os seguintes:

- a) *Nível 0 – incompleto*: pode ser tanto um processo que não foi estabelecido quanto um processo que não é executado de forma adequada. Um ou mais dos objetivos específicos da área de processo não são satisfeitos, e não existem objetivos genéricos, já que não existe razão para institucionalizar um processo parcialmente realizado.
- b) *Nível 1 – realizado*: é um processo seguido, mas que ainda não foi institucionalizado. Por esse motivo, a empresa corre o risco de perder essa conquista, caso não avance para os níveis seguintes.
- c) *Nível 2 – gerenciado*: é realizado de acordo com um planejamento e uma política definidos. Usa recursos humanos capacitados e produz produtos de forma previsível. Envolve os interessados relevantes, é monitorado, revisado e controlado. A aderência dos projetos ao processo é avaliada. Esse nível garante que as práticas sejam mantidas mesmo em períodos de estresse.
- d) *Nível 3 – definido*: é gerado a partir de um conjunto de processos padrão da organização, de acordo com as regras de geração de processos definidas. Sua descrição é mantida e sua evolução pode contribuir para o patrimônio de processos da empresa. Enquanto no nível 2 os processos podem ser bem heterogêneos, no nível 3 eles apresentam maior padronização.

12.3.3 NÍVEIS DE MATURIDADE

Os cinco níveis de maturidade são usados na versão em estágios do CMMI e refletem a obtenção de objetivos genéricos de áreas de processos na organização, em vez de apenas objetivos específicos. Os níveis são os seguintes:

- a) *Nível 1 – inicial*: nesse nível os processos costumam ser *ad hoc* e caóticos, não havendo um ambiente de suporte ao controle dos processos. O sucesso da empresa depende mais das capacidades individuais de seus funcionários do que de processos bem estabelecidos. Pode haver uma tendência a abandonar os processos, se houver algum, em tempos de crise.
- b) *Nível 2 – gerenciado*: nesse nível os projetos são planejados e executados de acordo com uma política, e suas saídas são controladas e previsíveis. Existem práticas que são mantidas mesmo em tempos de estresse. O *status* dos produtos do trabalho é visível para a gerência.
- c) *Nível 3 – definido*: nesse nível os processos são bem caracterizados e gerados a partir de padrões da organização. Existe consistência entre os processos das diferentes áreas da organização. Processos nesse nível costumam ser definidos de forma mais rigorosa.
- d) *Nível 4 – quantitativamente gerenciado*: nesse nível a organização estabelece metas de qualidade quantitativas e usa essas medidas no gerenciamento de projetos. A qualidade dos processos e produtos é compreendida em termos estatísticos e gerenciada de forma que seja quantitativamente previsível.
- e) *Nível 5 – em otimização*: nesse nível a organização melhora continuamente seus processos, baseando-se nas medições quantitativas obtidas.

12.3.4 ÁREAS DE PROCESSO DO CMMI-DEV v1.3

Dependendo da vertente (aquisição, desenvolvimento ou serviços), diferentes áreas de processos são consideradas. As áreas-chave são os processos que devem ser realizados pela organização para atingir determinado nível de maturidade.

O CMMI-DEV é a vertente mais relevante para Engenharia de Software. Ela contém 22 áreas de processo: 16 pertencem ao núcleo de áreas de processo, ou seja, são áreas comuns a todos os modelos CMMI, 1 é compartilhada com outra área e 5 são específicas da área de desenvolvimento. Todas as práticas do modelo CMMI-DEV são focadas nas atividades da organização desenvolvedora. Cinco áreas de processo são focadas em práticas específicas de desenvolvimento: desenvolvimento de requisitos, solução técnica, integração de produto, verificação e validação (SEI-CMU, 2010).

As 22 áreas de processo do CMMI-DEV são apresentadas na Tabela 12.3.

TABELA 12.3 As 22 áreas de processo de CMMI-DEV

| Sigla | Área de processo | Categoria | Nível | Propósito |
|-------|---|--------------------------|-------|---|
| PMC | Monitoramento e controle de projeto | Gerenciamento de projeto | 2 | Fornecer um entendimento do progresso do projeto de forma que as ações corretivas apropriadas possam ser realizadas quando o desempenho do projeto se desviar significativamente do plano. |
| PP | Planejamento de projetos | Gerenciamento de projeto | 2 | Estabelecer e manter planos que definam as atividades de projetos. |
| REQM | Gerenciamento de requisitos | Gerenciamento de projeto | 2 | Gerenciar os requisitos dos produtos e componentes de produto do projeto e garantir um alinhamento entre esses requisitos e os planos de projeto e produtos de trabalho. |
| SAM | Gerenciamento de acordos com fornecedores | Gerenciamento de projeto | 2 | Gerenciar a aquisição de produtos e serviços de fornecedores. |
| CM | Gerenciamento de configuração | Supporte | 2 | Estabelecer e manter a integridade dos produtos de trabalho usando identificação de configuração, controle de configuração, relatório de status de configuração e auditorias de configuração. |
| MA | Medição e análise | Supporte | 2 | Desenvolver e sustentar uma capacidade de medição usada para dar suporte às necessidades de informação da gerência. |
| PPQA | Garantia de qualidade de processo e produto | Supporte | 2 | Fornecer à equipe e aos gerentes um entendimento objetivo sobre os processos e produtos de trabalho associados. |
| PI | Integração de produto | Engenharia | 3 | Construir o produto a partir de componentes, certificando-se de que o produto, à medida que for integrado, se comporte adequadamente (ou seja, possua a funcionalidade requerida e os atributos de qualidade) e entregar o produto. |
| RD | Desenvolvimento de requisitos | Engenharia | 3 | Eliciar, analisar e estabelecer os requisitos do cliente, do produto e dos componentes do produto. |
| TS | Solução técnica | Engenharia | 3 | Selecionar, projetar e implementar soluções para os requisitos. Soluções, designs e implementações englobam os produtos, componentes de produtos e processos do ciclo de vida do produto, tanto individualmente quanto de forma combinada, conforme o caso. |
| VAL | Validação | Engenharia | 3 | Demonstrar que um produto ou componente de produto satisfaz seu uso pretendido quando colocado no ambiente-alvo. |
| VER | Verificação | Engenharia | 3 | Garantir que produtos selecionados satisfaçam aos requisitos especificados para eles. |
| IPM | Gerenciamento integrado de projeto | Gerenciamento de projeto | 3 | Estabelecer e gerenciar o projeto e o envolvimento dos interessados relevantes de acordo com um processo definido e integrado, que é gerado a partir de um conjunto de processos padrão da organização. |
| OPD | Definição de processo organizacional | Gerenciamento de projeto | 3 | Estabelecer e manter um patrimônio útil de processos organizacionais, padrões ambientais de trabalho, regras e orientações para equipes. |
| OPF | Foco de processo organizacional | Gerenciamento de projeto | 3 | Planejar, implementar e implantar melhorias em processos organizacionais baseados em uma compreensão extensiva dos pontos fortes e fracos atuais dos processos executados na organização e do seu patrimônio de processos. |

(Continua)

TABELA 12.3 As 22 áreas de processo de CMMI-DEV (*Continuação*)

| Sigla | Área de processo | Categoria | Nível | Propósito |
|-------|--|--------------------------|-------|---|
| OT | Treinamento organizacional | Gerenciamento de projeto | 3 | Desenvolver habilidades e conhecimentos nas pessoas de forma que elas possam desempenhar seus papéis de forma efetiva e eficiente. |
| RSKM | Gerenciamento de riscos | Gerenciamento de projeto | 3 | Identificar problemas potenciais antes que eles ocorram, de forma que as atividades de tratamento de riscos possam ser planejadas e invocadas, na medida do necessário, ao longo do ciclo de vida do produto ou projeto, de forma a mitigar impactos adversos na obtenção dos objetivos. |
| DAR | Análise de decisão e resolução | Supporte | 3 | Analizar possíveis decisões usando um processo de avaliação formal que avalie alternativas identificadas com respeito a critérios estabelecidos. |
| OPP | Desempenho de processo organizacional | Gerenciamento de projeto | 4 | Estabelecer e manter um entendimento quantitativo sobre o desempenho de processos selecionados no conjunto de processos padrão da organização para obter qualidade e objetivos de desempenho de processo e também para fornecer dados de desempenho de processo, bases e modelos para gerenciar quantitativamente os projetos da organização. |
| QPM | Gerenciamento de projeto quantitativo | Gerenciamento de projeto | 4 | Gerenciar quantitativamente o projeto para obter os objetivos estabelecidos de desempenho e de qualidade do projeto. |
| OPM | Gerenciamento de desempenho organizacional | Gerenciamento de projeto | 5 | Gerenciar proativamente o desempenho da organização de forma a atingir seus objetivos de negócio. |
| CAR | Análise causal e resolução | Supporte | 5 | Identificar as causas de resultados selecionados e realizar ações para melhorar o desempenho do processo. |

Na tabela, pode-se ver que, das 22 áreas de processo, 5 são de suporte, 12, de gerenciamento de projeto, e 5, de engenharia. O nível colocado na quarta coluna indica o nível de maturidade no qual se espera que cada área de processo esteja definida. Veja mais detalhes na seção seguinte.

A documentação do CMMI apresenta descrições detalhadas de cada área de processo, bem como melhores práticas e exemplos. Na tabela, apenas o propósito da área, ou seja, sua definição de mais alto nível, foi mencionada.

12.3.5 NÍVEIS DE MATURIDADE CMMI

O CMMI não apresenta um nível de maturidade 0, como o SPICE. O nível 1 do CMMI, chamado de “inicial”, corresponde aos níveis 0 e 1 do SPICE, em que os projetos são imprevisíveis e mal controlados, e o comportamento da gerência é mais reativo do que proativo.

Os níveis 2 a 5 do CMMI são obtidos à medida que a empresa consegue atingir níveis de capacidade em processos necessários para cada nível, os quais são definidos a seguir.

Para o nível de maturidade 2, *gerenciado*, é necessário que as seguintes áreas de processo atinjam o nível de capacidade 2 ou 3:

- a) CM – Gerenciamento de Configuração.
- b) MA – Medição e Análise.
- c) PMC – Monitoramento e Controle de Projeto.
- d) PP – Planejamento de Projeto.
- e) PPQA – Garantia de Qualidade de Processo e Produto.
- f) REQM – Gerenciamento de Requisitos.
- g) SAM – Gerenciamento de Acordos com Fornecedores.

Para o nível de maturidade 3, *definido*, é necessário que as áreas de processo do nível 2 e as seguintes áreas atinjam nível de capacidade 3:

- a) DAR – Análise e Resolução de Decisões.
- b) IPM – Gerenciamento Integrado de Projetos.
- c) OPD – Definição de Processo Organizacional.
- d) OPF – Foco de Processo Organizacional.
- e) OT – Treinamento Organizacional.
- f) PI – Integração de Produtos.
- g) RD – Desenvolvimento de Requisitos.
- h) RSKM – Gerenciamento de Riscos.
- i) TS – Solução Técnica.
- j) VAL – Validação.
- k) VER – Verificação.

Para o nível de maturidade 4, *quantitativamente gerenciado*, é necessário que todas as áreas de processo anteriores e as áreas de processo seguintes atinjam nível de capacidade 3:

- a) QPM – Gerenciamento de Projeto Quantitativo.
- b) OPP – Desempenho de Processo Organizacional.

Para o nível de maturidade 5, *em otimização*, é necessário que todas as áreas de processo anteriores e as áreas de processo seguintes atinjam nível de capacidade 3:

- a) CAR – Análise Causal e Resolução.
- b) OID – Inovação e Implantação Organizacional.

Uma organização não é “certificada” em CMMI, mas “avaliada” (Ahern, Armstrong, Clouse, Ferguson, Hayes & Nidiffer, 2005). Dependendo do tipo de avaliação, a organização pode obter um nível de 1 a 5 ou um perfil de maturidade. Os motivos que levam uma organização a buscar a avaliação costumam ser os seguintes:

- a) Para determinar quão bem os processos da organização se comparam às melhores práticas propostas pelo CMMI e identificar as áreas nas quais melhorias podem ser feitas.
- b) Para informar clientes e fornecedores sobre a capacidade em processos da empresa.
- c) Para satisfazer cláusulas contratuais estabelecidas por clientes e fornecedores.

Embora não seja oficial, existe um *site* que mantém atualizado o registro das empresas brasileiras avaliadas pelo CMMI¹⁰.

12.4 MPS.BR

MR-MPS (*Modelo de Referência para Melhoria do Processo de Software*), ou MPS.BR, é um modelo de avaliação de empresas produtoras de software brasileiro criado através de uma parceria entre a SOFTEX, o governo federal e academia (pesquisadores em geral). O modelo brasileiro é independente, mas compatível com as Normas ISO 12207 e 15504 (SPICE), bem como com o CMMI.

A principal justificativa para a criação desse modelo foram os altos custos dos processos de avaliação ou certificação internacionais, que se tornam proibitivos para pequenas e médias empresas. Assim, o MPS.BR apresenta um custo significativamente mais baixo, por ter consultores e avaliadores residentes no Brasil e também pelo fato de que apresenta 7 níveis de maturidade em vez de apenas 5, como o CMMI. Isso faz que a escala de progressão na melhoria de processos tenha degraus mais suaves, especialmente nos níveis mais baixos, ou seja, é possível subir um nível com menos esforço do que seria necessário para subir um nível no CMMI.

O *site* da SOFTEX¹¹ disponibiliza gratuitamente acesso ao guia geral do modelo, bem como aos guias de aquisição de software, avaliação (especialmente para consultores, mas também útil para empresas que desejam se

¹⁰Disponível em: <www.blogcmmi.com.br/avaliacao/lista-de-empresas-cmmi-no-brasil>. Acesso em: 21 jan. 2013.

¹¹Disponível em: <www.softex.br/mpsbr/_guias/>. Acesso em: 21 jan. 2013.

autoavaliar) e implementação. O guia de implementação se divide em 11 partes, sendo as 7 primeiras relacionadas aos 7 níveis MPS.BR (um guia para cada um dos níveis), e as demais sendo guias de implementação em empresas que adquirem software, em fábricas de software¹², em fábricas de teste, e para avaliação e implementação do MPS.BR em conjunto com CMMI-DEV.

Os níveis de maturidade do MPS.BR são os seguintes:

- a) A – Em otimização.
- b) B – Gerenciado quantitativamente.
- c) C – Definido.
- d) D – Largamente definido.
- e) E – Parcialmente definido.
- f) F – Gerenciado.
- g) G – Parcialmente gerenciado.

Assim como em SPICE e CMMI, os níveis são cumulativos, isto é, para subir um nível devem-se satisfazer todos os critérios dos níveis anteriores e os do nível para o qual se deseja subir. Da mesma forma, os níveis são avaliados a partir de atributos de processo (AP), que são 9. Cada atributo de processo no MPS.BR é detalhado por um conjunto de *resultados esperados* (RAP), conforme mostrado a seguir (SOFTEX, 2011):

- a) AP 1.1 – *O processo é executado*: esse atributo evidencia o quanto o processo atinge o seu propósito.
 - RAP 1: O processo atinge os resultados definidos.
- b) AP 2.1 – *O processo é gerenciado*: esse atributo evidencia o quanto a execução do processo é gerenciada.
 - RAP 2: existe uma política organizacional estabelecida e mantida para o processo.
 - RAP 3: a execução do processo é planejada.
 - RAP 4 (para o nível G): a execução do processo é monitorada e ajustes são realizados.
 - RAP 4 (a partir do nível F): medidas são planejadas e coletadas para monitoramento da execução do processo, e ajustes são realizados.
 - RAP 5: as informações e os recursos necessários para a execução do processo são identificados e disponibilizados.
 - RAP 6 (até o nível F): as responsabilidades e a autoridade para executar o processo são definidas, atribuídas e comunicadas.
 - RAP 6 (a partir do nível E): os papéis requeridos, responsabilidades e autoridade para execução do processo definido são atribuídos e comunicados.
 - RAP 7: as pessoas que executam o processo são competentes em termos de formação, treinamento e experiência.
 - RAP 8: a comunicação entre as partes interessadas no processo é planejada e executada de forma a garantir o seu envolvimento.
 - RAP 9 (até o nível F): os resultados do processo são revistos com a gerência de alto nível para fornecer visibilidade sobre a sua situação na organização.
 - RAP 9 (a partir do nível E): métodos adequados para monitorar a eficácia e a adequação do processo são determinados, e os resultados do processo são revistos com a gerência de alto nível para fornecer visibilidade sobre a sua situação na organização.
 - RAP 10 (para o nível G): o processo planejado para o projeto é executado.
 - RAP 10 (a partir do nível F): a aderência dos processos executados às descrições de processo, padrões e procedimentos é avaliada objetivamente, e são tratadas as não conformidades.
- c) AP 2.2 – *Os produtos de trabalho do processo são gerenciados*: esse atributo evidencia o quanto os produtos de trabalho produzidos pelo processo são gerenciados apropriadamente.
 - RAP 11: os requisitos dos produtos de trabalho do processo são identificados.
 - RAP 12: os requisitos para documentação e controle dos produtos de trabalho são estabelecidos.

¹²Uma fábrica de software é uma estrutura organizacional especializada em produzir componentes ou sistemas de software completos a partir de especificações produzidas externamente. Um exemplo típico são organizações que repassam requisitos completos e validados para fábricas de software que se responsabilizam pela construção de um sistema que atenda a esses requisitos.

- RAP 13: os produtos de trabalho são colocados em níveis de controle apropriados.
 - RAP 14: os produtos de trabalho são avaliados objetivamente com relação aos padrões, procedimentos e requisitos aplicáveis, e são tratadas as não conformidades.
- d) AP 3.1 – O processo é definido:** esse atributo evidencia o quanto um processo padrão é mantido para apoiar a implementação do processo definido.
- RAP 15: um processo padrão é descrito, incluindo diretrizes para sua adaptação.
 - RAP 16: a sequência e a interação do processo padrão com outros processos são determinadas.
 - RAP 17: os papéis e competências requeridos para executar o processo são identificados como parte do processo padrão.
 - RAP 18: a infraestrutura e o ambiente de trabalho requeridos para executar o processo são identificados como parte do processo padrão.
- e) AP 3.2 – O processo está implementado:** esse atributo evidencia o quanto o processo padrão é efetivamente implementado como um processo definido para atingir seus resultados.
- RAP 19: um processo definido é implementado com base nas diretrizes para seleção e/ou adaptação do processo padrão.
 - RAP 20: a infraestrutura e o ambiente de trabalho requeridos para executar o processo definido são disponibilizados, gerenciados e mantidos.
 - RAP 21: dados apropriados são coletados e analisados, constituindo uma base para o entendimento do comportamento do processo, para demonstrar a adequação e a eficácia do processo e avaliar onde pode ser feita a melhoria contínua do processo.
- f) AP 4.1 – O processo é medido:** esse atributo evidencia o quanto os resultados de medição são usados para assegurar que a execução do processo atinja seus objetivos de desempenho e apoia o alcance dos objetivos de negócio definidos.
- RAP 22: as necessidades de informação dos usuários dos processos, requeridas para apoiar objetivos de negócio relevantes da organização, são identificadas.
 - RAP 23: os objetivos de medição organizacionais dos processos e/ou subprocessos são derivados das necessidades de informação dos usuários do processo.
 - RAP 24: os objetivos quantitativos organizacionais de qualidade e de desempenho dos processos e/ou subprocessos são definidos para apoiar os objetivos de negócio.
 - RAP 25: os processos e/ou subprocessos que serão objeto de análise de desempenho são selecionados a partir do conjunto de processos padrão da organização e das necessidades de informação dos usuários dos processos.
 - RAP 26: as medidas, bem como a frequência de realização de suas medições, são identificadas e definidas de acordo com os objetivos de medição do processo/subprocesso e os objetivos quantitativos de qualidade e de desempenho do processo.
 - RAP 27: os resultados das medições são coletados e analisados, utilizando técnicas estatísticas e outras técnicas quantitativas apropriadas, e são comunicados para monitorar o alcance dos objetivos quantitativos de qualidade e de desempenho do processo/subprocesso.
 - RAP 28: os resultados de medição são utilizados para caracterizar o desempenho do processo/subprocesso.
 - RAP 29: os modelos de desempenho do processo são estabelecidos e mantidos.
- g) AP 4.2 – O processo é controlado:** esse atributo evidencia o quanto o processo é controlado estatisticamente para produzir um processo estável, capaz e previsível dentro de limites estabelecidos.
- RAP 30: técnicas de análise e de controle para a gerência quantitativa dos processos/subprocessos são identificadas e aplicadas quando necessário.
 - RAP 31: limites de controle de variação são estabelecidos para o desempenho normal do processo.
 - RAP 32: dados de medição são analisados com relação a causas especiais de variação.
 - RAP 33: ações corretivas e preventivas são realizadas para tratar de causas especiais ou de outros tipos de variação.
 - RAP 34: limites de controle são restabelecidos quando necessário, seguindo as ações corretivas, de forma que os processos continuem estáveis, capazes e previsíveis.

- h) AP 5.1 – O processo é objeto de melhorias incrementais e inovações:** esse atributo evidencia o quanto as mudanças no processo são identificadas a partir da análise de defeitos, problemas, causas comuns de variação do desempenho e da investigação de enfoques inovadores para a definição e implementação do processo.
- RAP 35: objetivos de negócio da organização são mantidos com base no entendimento das estratégias de negócio e resultados de desempenho do processo.
 - RAP 36: objetivos de melhoria do processo são definidos com base no entendimento do desempenho do processo, de forma a verificar que os objetivos de negócio relevantes são atingíveis.
 - RAP 37: dados que influenciam o desempenho do processo são identificados, classificados e selecionados para a análise de causas.
 - RAP 38: dados selecionados são analisados para identificar causas-raiz e propor soluções aceitáveis para evitar ocorrências futuras de resultados similares ou incorporar melhores práticas ao processo.
 - RAP 39: dados adequados são analisados para identificar causas comuns de variação no desempenho do processo.
 - RAP 40: dados adequados são analisados para identificar oportunidades para aplicar melhores práticas e inovações com impacto no alcance dos objetivos de negócio.
 - RAP 41: oportunidades de melhoria derivadas de novas tecnologias e conceitos de processo são identificados, avaliados e selecionados com base no impacto no alcance dos objetivos de negócio.
 - RAP 42: uma estratégia de implementação para as melhorias selecionadas é estabelecida para alcançar os objetivos de melhoria do processo e para resolver problemas.
- i) AP 5.2 – O processo é otimizado continuamente:** esse atributo evidencia o quanto as mudanças na definição, gerência e desempenho do processo têm impacto efetivo para o alcance dos objetivos relevantes de melhoria do processo.
- RAP 43: o impacto de todas as mudanças propostas é avaliado com relação aos objetivos do processo definido e do processo padrão.
 - RAP 44: a implementação de todas as mudanças acordadas é gerenciada para assegurar que qualquer alteração no desempenho do processo seja entendida e para que sejam realizadas as ações pertinentes.
 - RAP 45: as ações implementadas para resolução de problemas e melhoria no processo são acompanhadas do uso de técnicas estatísticas e outras técnicas quantitativas para verificar se as mudanças no processo corrigiram o problema e melhoraram o seu desempenho.
 - RAP 46: dados da análise de causas e de resolução são armazenados para uso em situações similares.

A Tabela 12.4 mostra como são obtidos os diferentes níveis de maturidade MPS.BR. Para cada nível devem-se implementar os processos definidos no nível e os dos níveis anteriores, e obter nos processos do nível os atributos de processo estabelecidos na coluna da direita para o nível, bem como os estabelecidos para os níveis anteriores, ou seja, tanto a coluna de processos quanto a de atributos de processos são interpretadas de forma cumulativa.

É interessante observar que o modelo prevê que alguns processos possam ser excluídos da avaliação em função de características especiais da companhia que está sendo avaliada:

- a) AQU (Aquisição):** se a empresa não realiza aquisição, esse processo pode ser excluído.
- b) GPP (Gerência de portfólio de projetos):** se a única atividade da organização for evolução (manutenção) de produtos, então esse processo pode ser excluído.
- c) DRU (Desenvolvimento para reutilização):** se a empresa conseguir demonstrar formalmente que não existem oportunidades reais para reutilização, então esse processo pode ser excluído.

Organizações que fazem exclusivamente aquisição de software, fábricas de código e fábricas de teste têm seus próprios conjuntos de processos incluídos e excluídos determinados em seus guias específicos.

Além disso, para os níveis A e B, os resultados esperados RAP 22 até RAP 46, referentes aos atributos 4.1 a 5.2, só precisam ser observados para os processos críticos da organização selecionados a fim de serem gerenciados quantitativamente, podendo ser relaxados para outros processos aos quais não se aplicam.

Cada processo do MPS.BR é detalhadamente descrito no guia geral e, por sua semelhança com os processos CMMI e SPICE, oriundos da ISO 12207, eles não serão descritos aqui. Para ver mais detalhes, pode-se consultar o guia no site da SOFTEX¹³.

¹³Disponível em: <www.softex.br/mpsbr/_guias/guias/MPS.BR_Guia_Geral_2011.pdf>. Acesso em: 21 jan. 2013.

TABELA 12.4 Processos e atributos de processos que definem os níveis de maturidade do MPS.BR

| Nível | Processos | Atributos de processo |
|-------|--|-----------------------|
| G | GRE – Gerência de requisitos GPR – Gerência de projetos | AP 1.1 e AP 2.1 |
| F | MED – Medição GQA – Garantia de qualidade GPP – Gerência de portfólio de projetos GCO – Gerência de configuração AQU – Aquisição | AP 2.2 |
| E | GPR (evolução) – Gerência de requisitos GRU – Gerência de reutilização GRH – Gerência de recursos humanos DFP – Definição do processo organizacional AMP – Avaliação e melhoria do processo organizacional | AP 3.1 e AP 3.2 |
| D | VER – Verificação VAL – Validação PCP – Projeto e construção do portfólio ITP – Integração do produto DRE – Desenvolvimento de requisitos | |
| C | GRI – Gerência de riscos DRU – Desenvolvimento para reutilização GDE – Gerência de decisões | |
| B | GPR (evolução) – Gerência de projetos | AP 4.1 e AP 4.2 |
| A | | AP 5.1 e AP 5.2 |

Apenas a título de exemplo, porém, segue a definição do processo GPR (Gerência de Projeto) conforme o guia geral. O propósito desse processo é: “Estabelecer e manter planos que definem as atividades, recursos e responsabilidades do projeto, bem como prover informações sobre o andamento do projeto que permitam a realização de correções quando houver desvios significativos no desempenho do projeto. O propósito desse processo evolui à medida que a organização cresce em maturidade. Assim, a partir do nível E, alguns resultados evoluem e outros são incorporados, de forma que a gerência de projetos passe a ser realizada com base no processo definido para o projeto e nos planos integrados. No nível B, a gerência de projetos passa a ter um enfoque quantitativo, refletindo a alta maturidade que se espera da organização. Novamente, alguns resultados evoluem e outros são incorporados” (SOFTEX, 2011).

Assim como os atributos de processo, os processos também são definidos a partir de um conjunto de resultados esperados. No caso de GPR, os resultados esperados são:

- GPR 1: o escopo do trabalho para o projeto é definido.
- GPR 2: as tarefas e os produtos de trabalho do projeto são dimensionados utilizando-se métodos apropriados.
- GPR 3: o modelo e as fases do ciclo de vida do projeto são definidos.
- GPR 4 (até o nível F): o esforço e o custo para a execução das tarefas e os produtos de trabalho são estimados com base em dados históricos ou referências técnicas.
- GPR 4 (a partir do nível E): o planejamento e as estimativas das tarefas do projeto são feitos baseados no repositório de estimativas e no conjunto de ativos de processo organizacional.
- GPR 5: o orçamento e o cronograma do projeto, incluindo a definição de marcos e pontos de controle, são estabelecidos e mantidos.

- GPR 6: os riscos do projeto são identificados e o seu impacto, probabilidade de ocorrência e prioridade de tratamento são determinados e documentados.
- GPR 7: os recursos humanos para o projeto são planejados considerando-se o perfil e o conhecimento necessários para executá-lo.
- GPR 8 (até o nível F): os recursos e o ambiente de trabalho necessários para executar o projeto são planejados.
- GPR 8 (a partir do nível E): os recursos e o ambiente de trabalho necessários para executar os projetos são planejados a partir dos ambientes padrão de trabalho da organização.
- GPR 9: os dados relevantes do projeto são identificados e planejados quanto à forma de coleta, armazenamento e distribuição. Um mecanismo é estabelecido para acessá-los, incluindo, se pertinente, questões de privacidade e segurança.
- GPR 10: um plano geral para a execução do projeto é estabelecido com a integração de planos específicos.
- GPR 11: a viabilidade de atingir as metas do projeto é explicitamente avaliada considerando-se restrições e recursos disponíveis. Se necessário, ajustes são realizados.
- GPR 12: o plano do projeto é revisado com todos os interessados, e o compromisso com ele é obtido e mantido.
- GPR 13: o escopo, as tarefas, as estimativas, o orçamento e o cronograma do projeto são monitorados em relação ao planejado.
- GPR 14: os recursos materiais e humanos, bem como os dados relevantes do projeto, são monitorados em relação ao planejado.
- GPR 15: os riscos são monitorados em relação ao planejado.
- GPR 16: o envolvimento das partes interessadas no projeto é planejado, monitorado e mantido.
- GPR 17: as revisões são realizadas em marcos do projeto e conforme estabelecido no planejamento.
- GPR 18: os registros de problemas identificados e o resultado da análise de questões pertinentes, incluindo dependências críticas, são estabelecidos e tratados com as partes interessadas.
- GPR 19: ações para corrigir desvios em relação ao planejado e para prevenir a repetição dos problemas identificados são estabelecidas, implementadas e acompanhadas até sua conclusão.
- GPR 20 (a partir do nível E): equipes envolvidas no projeto são estabelecidas e mantidas a partir das regras e diretrizes para estruturação, formação e atuação.
- GPR 21 (a partir do nível E): experiências relacionadas aos processos contribuem para os ativos de processo organizacional.
- GPR 22 (a partir do nível E): um processo definido para o projeto é estabelecido de acordo com a estratégia para adaptação do processo da organização.
- GPR 22 (a partir do nível B): os objetivos de qualidade e de desempenho do processo definido para o projeto são estabelecidos e mantidos.
- GPR 23 (a partir do nível B): o processo definido para o projeto, que possibilita a ele atender seus objetivos de qualidade e de desempenho, é composto com base em técnicas estatísticas e outras técnicas quantitativas.
- GPR 24 (a partir do nível B): subprocessos e atributos críticos para avaliar o desempenho, que estão relacionados ao alcance dos objetivos de qualidade e de desempenho do processo do projeto, são selecionados.
- GPR 25 (a partir do nível B): medidas e técnicas analíticas são selecionadas para serem utilizadas na gerência quantitativa.
- GPR 26 (a partir do nível B): o desempenho dos subprocessos escolhidos para gerência quantitativa é monitorado utilizando-se técnicas estatísticas e outras técnicas quantitativas.
- GPR 27 (a partir do nível B): o projeto é gerenciado utilizando-se técnicas estatísticas e outras técnicas quantitativas para determinar se os objetivos de qualidade e de desempenho do processo serão atingidos.
- GPR 28 (a partir do nível B): questões que afetam os objetivos de qualidade e de desempenho do processo do projeto são alvo de análise de causa-raiz.

12.5 Melhoria de Processo de Software (SEI-IDEAL)

Os modelos de avaliação vistos até aqui são uma boa referência para as empresas que querem melhorar seus processos de desenvolvimento saberem aonde devem chegar. Contudo, esses modelos não explicam o caminho para chegar a esses objetivos.

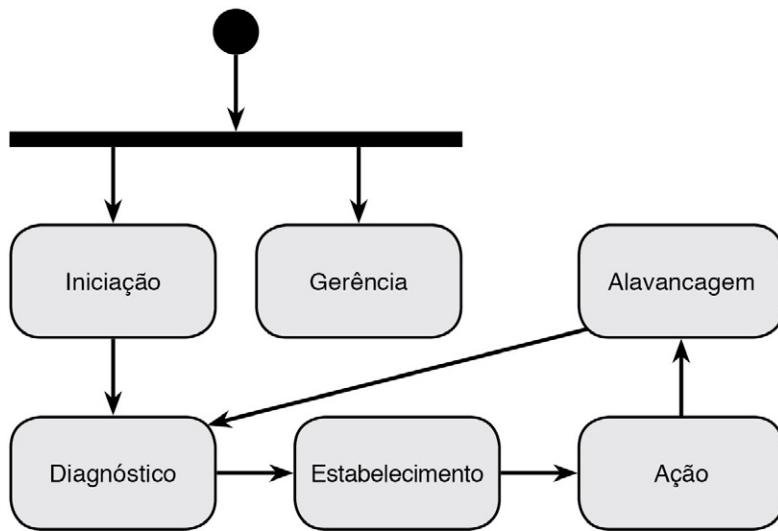


Figura 12.2 Ciclo de vida do modelo IDEAL.

Para trilhar esse caminho é necessário aplicar um modelo de melhoria de processo (SPI – *Software Process Improvement*). Uma boa referência é o modelo IDEAL (McFeeley, 1996)¹⁴, o qual será resumidamente apresentado nesta seção.

O modelo se baseia em cinco fases, das quais as quatro últimas podem ser executadas em ciclos. A ideia é que, a cada volta do ciclo, um novo degrau na melhoria de processos seja buscado e atingido (Figura 12.2). As fases são:

- Iniciação:** essa fase vai estabelecer os objetivos iniciais da iniciativa de melhoria de processo, buscar o comprometimento da alta gerência e os recursos para o trabalho e definir a equipe e a infraestrutura necessárias. Se não existir uma equipe de processo de engenharia de software, ela deverá ser criada nesse momento. As saídas importantes dessa fase são, portanto, a definição da *equipe de processo de engenharia de software (SEPG – Software Engineering Process Group)* e do *comitê diretor gerencial (MSP – Management Steering Group)*.
- Diagnóstico:** a fase de diagnóstico vai fazer a avaliação do estado atual dos processos da empresa para lançar as bases para o início do processo de melhoria continuada. O plano de melhoria, baseado nos objetivos estabelecidos na fase de iniciação, começa a ser executado, e os resultados das avaliações vão produzir atualizações nesse plano.
- Estabelecimento:** durante essa fase, os objetivos refinados na fase de diagnóstico serão priorizados e as estratégias para atingir os objetivos serão traçadas. O plano de melhoria será novamente refinado, sendo que os objetivos gerais estabelecidos nas fases anteriores agora serão transformados em objetivos mensuráveis, ou seja, será definida uma métrica, juntamente com seus mecanismos acessórios, para que se possa avaliar posteriormente se os objetivos foram mesmo atingidos.
- Ação:** nessa fase, os planos estabelecidos para atingir os objetivos são colocados em prática, inicialmente como projetos-piloto. Se aprovados nos projetos-piloto, então os novos processos poderão ser estabelecidos na organização.
- Alavancagem:** o objetivo dessa fase é capitalizar o patrimônio de informação obtido na iteração atual do ciclo de melhoria de processo para facilitar a execução do ciclo seguinte, que se reinicia na fase de diagnóstico. Nessa fase também é feita a avaliação da efetividade das atividades executadas no ciclo atual.

Uma sexta disciplina, que permeia todas as outras, é o *gerenciamento de melhoria de processo de software* (Figura 12.2).

O ciclo de vida da figura não tem propriamente um final, porque o processo de melhoria é contínuo, ou seja, idealmente, não acaba nunca.

¹⁴Disponível em: <www.sei.cmu.edu/reports/96hb001.pdf>. Acesso em: 21 jan. 2013.

As três atividades na parte superior da figura (iniciação, gerência e alavancagem) são consideradas atividades de nível estratégico, enquanto as outras três (diagnóstico, estabelecimento e ação) são consideradas atividades de nível tático.

12.5.1 INICIAÇÃO

Na fase de iniciação, o gerenciamento sênior da organização deve primeiramente compreender a necessidade de melhoria do processo de software, comprometer-se com um programa de melhoria e definir seu contexto.

A iniciação do projeto de melhoria é semelhante à de um projeto de software, devendo ser estabelecidos os requisitos iniciais e o escopo. Em geral, nessa fase será formada uma equipe de descoberta para estudar a empresa e descobrir os pontos em que a melhoria poderá ser aplicada. Essas descobertas deverão ser relatadas à gerência sênior.

Nessa fase são definidos o *comitê direutivo gerencial* (MSG) e a *equipe de processo de engenharia de software* (SEPG). O guia estabelece como objetivos para a fase de iniciação o seguinte:

- a) Construir consciência, habilidades e conhecimento iniciais para dar partida no processo de melhoria.
- b) Ganhar entendimento sobre quais são as necessidades de engajamento para o sucesso do plano.
- c) Determinar a capacidade de seguir em frente.
- d) Criar uma proposta para o programa de SPI, indicando as necessidades de melhoria, o escopo do programa e os requisitos em termos de recursos necessários.
- e) Recomendar cronograma e infraestrutura para gerenciar o programa.
- f) Planejar e obter compromissos para os próximos passos.

Visto que nessa fase inicial, possivelmente, a equipe terá pouco conhecimento sobre melhoria de processo, é de se esperar que muito estudo seja necessário para desenvolver as habilidades e os conhecimentos para o desenvolvimento do projeto.

Normalmente, as organizações iniciam um processo de SPI por conta de algum desastre ou de uma sucessão de falhas moderadas em seus projetos. Porém, o investimento em SPI só terá sucesso se efetivamente houver um compromisso da alta gerência. Caso essa necessidade não seja percebida, será melhor que o início do processo de melhoria seja postergado.

O guia estabelece como critérios de entrada para essa fase o seguinte:

- a) Compreende-se que existem necessidades de negócio críticas que demandam melhoria de processos de desenvolvimento de software.
- b) Os campeões de SPI da organização, ou seja, aqueles que vão levar o processo em frente, foram identificados.

Os critérios de saída dessa fase são os seguintes:

- a) A estrutura inicial de SPI foi estabelecida e está estabelecendo patrocínios e promovendo conceitos e atividades de SPI na empresa.
- b) A organização ligou a iniciativa de SPI com a sua estratégia de negócios.
- c) Um plano inicial de comunicação organizacional para a iniciativa de SPI foi completado.
- d) Foi estabelecido um programa de reconhecimento para demonstrar publicamente as recompensas pelos resultados em SPI.
- e) Um plano inicial específico para a organização foi criado para guiar a organização através das fases de iniciação, diagnóstico e estabelecimento do modelo IDEAL.

As atividades executadas nessa fase são exaustivamente detalhadas no modelo. Segue uma breve descrição de seus objetivos:

- a) 1.1 – *Iniciar*: identificar os departamentos que têm interesse em um programa de SPI. Avaliar e selecionar uma abordagem para conduzir o programa de SPI. Identificar as necessidades de negócio e as abordagens para SPI.
- b) 1.2 – *Identificar as necessidades de negócio e motivadores de melhoria*: identificar as necessidades-chave de negócio que motivam a necessidade de SPI. Ligar o programa de SPI a essas necessidades.
- c) 1.3 – *Construir uma proposta de SPI*: desenvolver e entregar a proposta de SPI para a gerência sênior.

- d) 1.4 – *Educar e construir suporte*: comunicar a necessidade de SPI para a organização. Comunicar a abordagem que a organização vai seguir para SPI. Introduzir e envolver os interessados-chave na comunicação e formar o programa de SPI.
- e) 1.5 – *Obter a aprovação da proposta de SPI e recursos iniciais*: obter a aprovação e recursos da gerência sênior, assim como a adesão de outros interessados-chave. Obter concordância para estabelecer o comitê diretivo gerencial e recursos para a formação da equipe de processo de engenharia de software. Obter compromisso de agenda da gerência sênior para as atividades subsequentes.
- f) 1.6 – *Estabelecer a infraestrutura de SPI*: estabelecer a infraestrutura. Iniciar as atividades de infraestrutura: facilitar o programa de SPI, dar apoio e monitorar as atividades da equipe de SPI, coordenar as atividades de SPI e fornecer patrocínio visível e efetivo para o programa de SPI.
- g) 1.7 – *Avaliar o clima organizacional para SPI*: identificar as principais barreiras organizacionais a um programa de SPI. Definir estratégias para reduzir essas barreiras. Definir estratégias para interagir com outros programas e iniciativas. Desenvolver estratégias para manter e melhorar o patrocínio ao programa de SPI. Atualizar o plano de comunicação de SPI. Desenvolver um programa para melhorar as habilidades dos agentes da mudança.
- h) 1.8 – *Definir as metas gerais de SPI*: definir objetivos de longo e curto prazos. Determinar quais métricas são necessárias para avaliar objetivamente a obtenção dos objetivos.
- i) 1.9 – *Definir os princípios orientadores gerais do programa de SPI*: para definir os princípios orientadores podem-se buscar as lições aprendidas na literatura ou em iniciativas passadas.
- j) 1.10 – *Iniciar o programa*: essa atividade faz a transição entre a fase inicial e a fase de diagnóstico, em que o plano é revisto, e sua continuidade, aprovada pela gerência superior.

O fluxo das atividades é basicamente sequencial, mas as atividades 1.2, 1.3 e 1.4 podem ser realizadas em paralelo, e as atividades 1.6 e 1.7, também.

12.5.2 DIAGNÓSTICO

Na fase de *diagnóstico*, a organização deverá entender e definir uma ou mais *baselines* de processos, ou seja, um conjunto formalmente aprovado de processos a serem tratados no ciclo atual de melhoria de processos.

As atividades de *baseline* realizadas na fase de diagnóstico vão produzir as informações necessárias para o planejamento de SPI e para a priorização das melhorias a serem feitas.

O plano de ação estratégica para SPI que será desenvolvido após as atividades de construção da *baseline* é crítico, pois vai conduzir várias ações de melhoria de processo ao longo dos anos seguintes, e por isso deve fornecer razões de negócios muito claras para conduzir o programa de SPI e deve estar claramente ligado aos planos de negócios da organização.

Para produzir a *baseline* de processos, recomenda-se utilizar um modelo de avaliação, como SPICE, CMMI ou MPS.BR.

De acordo com o guia IDEAL, os principais objetivos dessa fase são os seguintes:

- a) Compreender o funcionamento dos processos correntes e iterações da organização e como eles contribuem para os negócios da organização.
- b) Obter informação sobre os pontos fortes e oportunidades de melhoria correntes da organização, os quais serão entradas para o processo de planejamento de ações estratégicas de SPI.
- c) Conseguir o envolvimento da organização, desde a gerência sênior até os executores, para as tarefas de melhoria de processo, que farão o trabalho da organização ser mais efetivo.
- d) Detalhar o ponto de partida para medições de melhoria.

Os critérios de entrada para essa fase são:

- a) Há infraestrutura de SPI, particularmente o comitê diretivo gerencial e o grupo de processo de engenharia de software, que deverão estar definidos e operacionais.
- b) Recursos estão disponíveis para trabalhar nas *baselines*.
- c) O comitê diretivo gerencial decidiu que o plano de ação estratégica de SPI deve ser atualizado (caso não seja a primeira iteração).
- d) A visão, o plano de negócios e os objetivos de SPI da organização possuem sinergia.

Os critérios de saída da fase são os seguintes:

- a) O relatório de recomendações e achados da *baseline* foi entregue ao comitê diretivo gerencial e aceito.
- b) O rascunho do plano de ação estratégica de SPI foi iniciado.

As atividades previstas para essa fase são:

- a) 2.1 – *Determinar quais baselines são necessárias*: determinar quantas *baselines* e de que tipo devem ser tratadas.
- b) 2.2 – *Planejar para as baselines*: certificar-se de que todos os aspectos das atividades relacionadas às *baselines* foram considerados e planejados. Documentar as atividades necessárias para obter as *baselines*.
- c) 2.3 – *Conduzir as baselines*: obter informação real sobre os pontos fortes e os pontos fracos dos processos da organização que serão usados em atividades de desenvolvimento de software.
- d) 2.4 – *Apresentar achados*: fornecer aos participantes um retorno inicial sobre os resultados das atividades sobre as *baselines*. Construir suporte e consenso relacionados à validade dos achados.
- e) 2.5 – *Desenvolver achados finais e relatório de recomendações*: criar um conjunto de recomendações que tratem de cada um dos achados do esforço sobre as *baselines*.
- f) 2.6 – *Comunicar os achados e as recomendações para a organização*: ganhar suporte e patrocínio para SPI. Ganhar consenso nas áreas em que a SPI será tratada. Ganhar entradas adicionais relacionadas às soluções potenciais. Informar à organização quais são os próximos passos.

A execução dessas atividades acompanha a sequência estrita estabelecida.

12.5.3 ESTABELECIMENTO

Criar o plano de ação estratégica para o SPI é uma das atividades mais críticas e mais frequentemente negligenciadas. É nela que a equipe de gerenciamento desenvolve ou atualiza o plano de ação estratégica baseando-se na visão, no plano de negócios e nos esforços de melhoria passados, bem como nos achados dos esforços de *baseline*.

Nessa fase pode haver a grande tentação de começar imediatamente a fazer mudanças. Porém, a experiência mostra que sem um planejamento cuidadoso os esforços acabam falhando ou não atendendo às expectativas.

Segundo o guia IDEAL, os objetivos da fase de estabelecimento são os seguintes:

- a) Desenvolver/atualizar um plano de ação estratégica para SPI de longo prazo (de 3 a 5 anos) que engloba todas as atividades de melhoria de processo de software da organização e as integra com qualquer outra iniciativa de gerenciamento de qualidade que já tenha sido planejada ou esteja em progresso.
- b) Desenvolver/atualizar objetivos mensuráveis de longo prazo (3 a 5 anos) e curto prazo (1 ano) para o programa de SPI da organização.
- c) Integrar os achados de *baseline* e recomendações no plano de ação estratégica de SPI.
- d) Integrar o plano de ação estratégica de SPI com o plano de negócios, a missão e a visão da organização.

Os critérios de entrada da fase de estabelecimento são os seguintes:

- a) A infraestrutura de SPI está instalada e operacional.
- b) O comitê consultivo gerencial decidiu que o plano de ação estratégica de SPI precisa ser atualizado (caso não seja a primeira iteração).
- c) As atividades de *baseline* foram completadas.

Os critérios de saída são:

- a) O plano de ação estratégica de SPI está completo e aprovado.
- b) Existe sinergia entre a visão, os planos de negócio e o plano de ação estratégica de SPI na organização.

As atividades dessa fase são:

- a) 3.1 – *Selecionar e obter treinamento em um processo de planejamento estratégico*: selecionar um processo de planejamento. Treinar o comitê diretivo gerencial e a equipe de processo de engenharia de software nos processos e métodos.

- b) 3.2 – Revisar a visão da organização:** revisar e, se possível, modificar a visão corrente da organização. Gerar uma nova visão, se ela não existia antes ou se aquela que existia antes não era adequada. Identificar objetivos e motivações para o programa de SPI.
- c) 3.3 – Revisar o plano de negócios da organização:** revisar e, se possível, modificar o plano de negócios corrente da organização. Gerar um novo plano de negócios, se não existir um ou se o plano existente não for adequado. Identificar objetivos de outras iniciativas (possivelmente concorrentes).
- d) 3.4 – Determinar os problemas-chave do negócio:** determinar os problemas-chave do negócio que levam à necessidade de melhoria do processo de software.
- e) 3.5 – Revisar esforços de melhoria passados:** revisar os esforços de melhoria e/ou mudança passados, identificando práticas de sucesso a alavancar e práticas de insucesso a evitar.
- f) 3.6 – Descrever as motivações para a melhoria:** as motivações definidas, assim como os produtos das demais atividades dessa fase, serão documentadas no plano de ação estratégica de SPI.
- g) 3.7 – Identificar esforços de melhoria atuais e futuros (planejados):** identificar todos os esforços de melhoria antecipados e/ou existentes na organização, tanto os dirigidos internamente quanto os dirigidos externamente (como as iniciativas corporativas).
- h) 3.8 – Finalizar papéis e responsabilidades das várias entidades de infraestrutura:** finalizar os papéis e responsabilidades para o comitê consultivo gerencial, o grupo de processo de engenharia de software e quaisquer outros grupos de coordenação e gerência de SPI. Definir papéis típicos e responsabilidades para os *grupos de trabalho técnico (TWG – Technical Working Groups)* em termos de responsabilidades, autoridade, requisitos de relatório etc.
- i) 3.9 – Priorizar atividades e desenvolver uma agenda de melhoria:** definir os critérios para a seleção de projetos de SPI que devem ser documentados no plano estratégico.
- j) 3.10 – Conciliar os esforços de melhoria existentes/planejados com os achados de baseline e recomendações:** incorporar resultados de *baseline* ao plano estratégico de SPI. Conciliar resultados de *baseline* com todas as outras atividades de melhoria de software planejadas e/ou existentes.
- k) 3.11 – Transformar os objetivos gerais de melhoria de processo de software em objetivos específicos mensuráveis:** todos os objetivos devem ser definidos a partir de uma ou mais medições que possam ser obtidas.
- l) 3.12 – Criar/atualizar o plano estratégico de SPI:** consolidar e finalizar o plano estratégico de SPI.
- m) 3.13 – Construir um consenso, revisar e aprovar o plano estratégico de SPI e garantir recursos para agir:** aprovar o plano de ação estratégica de SPI. Construir consenso e comprometimento em relação ao plano, garantindo recursos para agir.
- n) 3.14 – Formar os grupos de trabalho técnico (TWG):** montar uma equipe de pessoas com diversos *backgrounds* que tenham participação na área de melhoria.

As atividades 3.2 a 3.7 podem ser executadas em paralelo. As demais são sequenciais.

12.5.4 AÇÃO

A fase de *ação* é aquela em que as melhorias e os novos processos serão efetivamente colocados em prática na organização. Essas novidades deverão ser inicialmente testadas de forma piloto pelos grupos de trabalho técnico e, depois de aprovadas, ser incorporadas ao patrimônio de processos da organização.

A fase de ação vai ser o momento em que a organização vai mudar a forma como faz as coisas. Nesse momento, as equipes técnicas estarão desenvolvendo melhorias para processos específicos, para o que existem duas abordagens básicas:

- a) Abordagem centrada em problema:** a equipe vai focar em um problema específico e desenvolver uma solução usando projetos-piloto para validar e refinar a solução.
- b) Abordagem centrada em processo:** a equipe vai focar em um processo particular e desenvolver refinamentos incrementais para ele usando projetos-piloto para testar os refinamentos.

Os objetivos dessa fase são os seguintes:

- a) Desenvolver ou refinar os processos de desenvolvimento de software de acordo com as prioridades do plano de ação.**

- b) Fazer que a hierarquia organizacional seja funcional com relação às melhorias que vai usar.
- c) Integrar as melhorias de processos com os planos de desenvolvimento de projetos novos ou existentes.
- d) Monitorar e suportar a hierarquia organizacional à medida que usar os processos novos ou modificados.

Cada grupo de trabalho técnico deverá fazer o seguinte:

- a) Planejar o projeto de melhoria: entender o processo, incluindo as necessidades dos clientes, e desenvolver refinamentos para ele (abordagem orientada a processo) ou investigar o problema e desenvolver uma solução (abordagem orientada a problemas).
- b) Criar uma solução-piloto, validá-la e refiná-la.
- c) Desenvolver um *template* de plano e uma estratégia de implantação para aplicar a solução.
- d) Avaliar a solução durante seu uso.
- e) Iterar novamente o ciclo para melhorias posteriores.

Os critérios de entrada dessa fase são:

- a) Autorização para as equipes de trabalho técnico e *template* de plano de ação tática do comitê direutivo gerencial e do grupo de processo de engenharia de software.
- b) Problemas de maturidade de processo da fase de estabelecimento.
- c) Recomendações relacionadas e objetivos fáceis (projetos com consertos rápidos e retorno rápido) a partir das *baselines*.
- d) Descritores de processo de alto nível da *baseline* de processos.
- e) Objetivos mensuráveis e identificados durante a fase de estabelecimento.
- f) Métricas de processos-chave a partir das métricas de *baseline*.

Os critérios de saída são os seguintes:

- a) A estratégia de implantação e o plano foram completamente executados ou estão em fase de execução.
- b) Soluções foram apropriadamente empacotadas e enviadas para a equipe de processo de engenharia de software.
- c) Suporte de longo prazo foi obtido.
- d) A melhoria de processo começou a ser institucionalizada na hierarquia organizacional.

As atividades da fase de ação são:

- a) 4.1 – *Completar o plano tático para os grupos de trabalho técnico*: completar as seções do plano de ação tática não especificadas pelo comitê direutivo gerencial e preencher outras áreas do plano. Rever e estreitar o escopo do projeto, se necessário, para algo que possa ser feito numa quantidade de tempo relativamente curta (seis a nove meses).
- b) 4.2 – *Desenvolver soluções*: investigar as soluções alternativas para processar problemas descobertos durante as atividades de *baseline*. Selecionar a solução que melhor atender às necessidades de negócio e cultura da organização.
- c) 4.3 – *Implementar piloto para soluções potenciais*: verificar a solução em um projeto real da organização. Capturar lições aprendidas e resultados do piloto para refinar a solução e a instalação da solução.
- d) 4.4 – *Selecionar fornecedores de solução*: investigar vários fornecedores de solução e seus históricos para encontrar aqueles que melhor satisfazem às necessidades da organização, tanto de curto quanto de longo prazo.
- e) 4.5 – *Determinar necessidades de suporte de longo prazo*: identificar necessidades de suporte e fontes potenciais para suporte. Planejar mecanismos de suporte interno de longo prazo. Assegurar recursos para suporte de longo prazo.
- f) 4.6 – *Desenvolver estratégias de implantação do template de plano*: criar um *template* para o plano de implantação da solução que possa ser personalizado pelos projetos durante a implantação de suas soluções.
- g) 4.7 – *Empacotar a melhoria e enviá-la ao grupo de processo de engenharia de software*: completar e limpar todos os produtos e artefatos intermediários. Empacotar produtos e artefatos para arquivamento com o grupo de processo de engenharia de software.

- h) 4.8 – Desfazer o grupo de trabalho técnico:** capitalizar as lições aprendidas a partir do esforço específico do grupo de trabalho. Celebrar os resultados dessa equipe.
- i) 4.9 – Implantar solução:** instalar a solução na estrutura da organização.
- j) 4.10 – Fazer a transição para o suporte de longo prazo:** suportar a hierarquia organizacional no uso normal do processo.

As atividades 4.4, 4.5 e 4.6 podem ser executadas em paralelo. As demais são sequenciais.

12.5.5 ALAVANCAGEM

Depois de terminar um ciclo de melhoria de processos, a organização deve revisar o que aconteceu ao longo dele e preparar-se para o próximo. Esse é o propósito da fase de *alavancagem*.

Conforme já foi dito, em vez de reiniciar o processo pela fase de iniciação, os ciclos iterativos do processo vão fazer isso na fase de planejamento. Isso porque a fase de alavancagem já terá feito os ajustes necessários para o novo ciclo.

Os objetivos da fase de alavancagem são:

- a) Incorporar lições aprendidas nas fases anteriores na abordagem de SPI.**
- b) Ganhar visibilidade para o valor da iniciativa de SPI.**
- c) Reafirmar a continuidade do patrocínio de SPI.**
- d) Estabelecer/ajustar objetivos de alto nível para o próximo ciclo.**
- e) Determinar *baselines* adicionais ou novas que possam ser necessárias.**
- f) Criar um plano para guiar a organização através do próximo ciclo.**

Os critérios de entrada para a fase de alavancagem são os seguintes:

- a) O ciclo através das fases anteriores do modelo IDEAL foi completado.**
- b) Relatórios de lições aprendidas em cada uma das fases anteriores estão disponíveis.**
- c) Artefatos produzidos durante a implementação de SPI estão disponíveis.**

Os critérios de saída são os seguintes:

- a) Lições aprendidas são analisadas e melhorias são incorporadas nos processos de SPI.**
- b) Patrocínio e comprometimento foram reafirmados com a gerência sênior.**
- c) Objetivos de alto nível foram estabelecidos para o próximo ciclo.**

As atividades dessa fase são:

- a) 5.1 – Reunir lições aprendidas:** certificar-se de que toda a informação sobre lições aprendidas nas atividades prévias realizadas durante o ciclo estão disponíveis para revisão. Refrescar a memória em relação às atividades das fases previamente completadas do modelo IDEAL.
- b) 5.2 – Analisar lições aprendidas:** analisar práticas e processos de melhoria passados para fazer o próximo ciclo do modelo ideal funcionar melhor. Considerar a exclusão e a troca de processos que não funcionaram bem, assim como adicionar novos processos que farão o trabalho de SPI funcionar melhor.
- c) 5.3 – Revisar a abordagem organizacional:** desenvolver processos de SPI mais efetivos e eficientes. Reduzir a resistência a SPI. Garantir patrocínio efetivo para SPI.
- d) 5.4 – Revisar patrocínios e comprometimento:** certificar-se de que a gerência está comprometida com o esforço de SPI e continuará a fornecer patrocínio e comprometimento para que o programa tenha sucesso. Certificar-se de que os recursos estejam disponíveis para continuar o programa de SPI.
- e) 5.5 – Estabelecer objetivos de alto nível:** refinar os objetivos de longo prazo, as medições e o processo de medição para determinar objetivamente a satisfação dos objetivos. Ligar o programa de SPI à visão e às necessidades de negócio da organização.
- f) 5.6 – Desenvolver propostas de SPI em andamento ou novas:** fornecer orientação para o programa de SPI até que as *baselines* necessárias tenham sido completadas e um novo plano de ação tenha sido criado.
- g) 5.7 – Continuar com SPI:** fazer a transição da fase de alavancagem de volta para a fase de diagnóstico.

Todas as atividades listadas aqui são estritamente sequenciais.

12.5.6 GERENCIAMENTO DO PROGRAMA DE SPI

Uma iniciativa de melhoria de processos de software frequentemente representará um esforço considerável para uma organização. A coordenação dessas atividades é, portanto, fundamental para o sucesso da iniciativa.

As atividades relacionadas ao processo de gerência de SPI são as seguintes:

- a) 6.1 – *Preparar o palco para SPI*: estabelecer prioridades para o programa de SPI. Aprovar planos de ação estratégica para a SPI. Alocar recursos. Monitorar o progresso da melhoria em relação ao plano. Desenvolver um sistema de recompensas. Fornecer patrocínio visível e continuado.
- b) 6.2 – *Organizar o programa de SPI*: estabelecer infraestrutura para gerenciar o programa de SPI. Criar consciência organizacional para o programa de SPI.
- c) 6.3 – *Planejar programa de SPI*: definir os objetivos do programa de SPI. Fornecer foco e direção para as atividades de SPI. Determinar os recursos necessários para o programa de SPI. Mostrar comprometimento com o programa de SPI.
- d) 6.4 – *Alocar recursos humanos para o programa de SPI*: atribuir pessoal em nível de gerência para o comitê direutivo gerencial. Recrutar pessoal qualificado para a equipe de processo de engenharia de software. Recrutar e/ou atribuir representação apropriada para os grupos de trabalho técnicos.
- e) 6.5 – *Monitorar o programa de SPI*: certificar-se de que as atividades de melhoria são consistentes com os objetivos corporativos, os planos para o programa de SPI estão sendo seguidos e o progresso na direção dos objetivos de melhoria está sendo feito.
- f) 6.6 – *Dirigir o programa de SPI*: certificar-se de que a direção do programa de SPI é consistente com a visão e a missão da organização.

Essas atividades são basicamente sequenciais, sendo que apenas as duas últimas (6.5 e 6.6) devem ser executadas em paralelo. Convém lembrar, porém, que, quando se inicia um novo ciclo de melhoria, pode ser necessário realizar – ou pelo menos rever – as atividades de gerência desde o início (6.1).

12.6 Fatores Humanos

Apesar da existência de boas referências para melhoria de processos e equipes bem intencionadas, a literatura reporta que cerca de 70% das iniciativas falham ou sequer se iniciam (Iversen, Mathiassen & Nielsen, 2004; Santana, 2007).

Ocorre que, por mais detalhados que sejam os modelos de melhoria de processos, tanto o trabalho de melhoria quanto o trabalho com os processos melhorados acabam sendo feitos por pessoas. Devem-se, portanto, considerar os fatores humanos envolvidos com as atividades de mudança dentro da empresa (Ferreira & Wazlawick, 2011).

O processo de mudança é complexo e demanda grande esforço para que se obtenha sucesso. Conner e Patterson (1982) caracterizam o processo de adoção de mudanças em oito estágios, organizados em três fases (Figura 12.3):

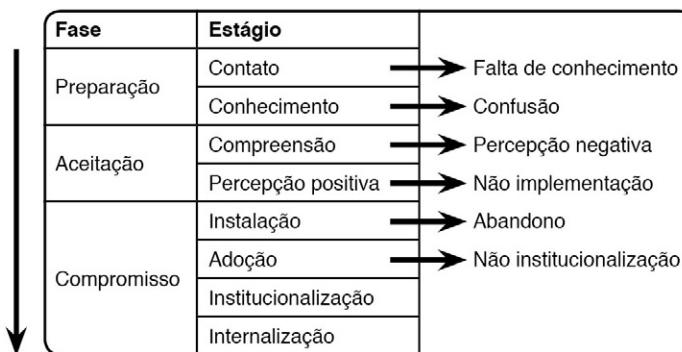


Figura 12.3 Fases, estágios e desistências no processo de mudança.

- a) *Preparação*: nessa fase ocorrem os estágios de *contato e conhecimento*. Se após o contato a educação necessária para chegar ao conhecimento não for estabelecida, a iniciativa de mudança poderá falhar por *falta de conhecimento*.
- b) *Aceitação*: nessa fase ocorrem os estágios de *compreensão e percepção positiva*. Se após o conhecimento não houver compreensão, a iniciativa poderá falhar devido à *confusão*. Se durante a compreensão houver percepção negativa sobre a possibilidade de mudança, esta também poderá não ocorrer. Porém, mesmo se houver compreensão e percepção positiva, a iniciativa ainda poderá não ser implementada em função de decisão baseada em custo/benefício ou outros fatores.
- c) *Compromisso*: nessa fase ocorrem os estágios de *instalação, adoção, institucionalização e internalização*. Mesmo após a instalação e o uso inicial, os novos processos ainda poderão ser abandonados antes de serem efetivamente adotados pela organização. E, mesmo após a adoção oficial e o uso extensivo dos novos processos, eles ainda poderão ser abandonados se não forem efetivamente institucionalizados. A internalização é o estágio final no qual o que foi institucionalizado passa a fazer parte da cultura organizacional e nem é mais percebido como algo à parte.

Kotter (2006) sugere uma sequência de oito passos que a organização deveria seguir para ter sucesso na gestão de mudanças organizacionais (*CM – Change Management*), como no caso de SPI. Como se pode ver abaixo, essa sequência envolve fortemente os fatores humanos da organização:

- a) *Estabelecer um senso de urgência*: é necessário mostrar para toda a organização que a mudança é efetivamente necessária para o sucesso do negócio e a realização pessoal dos envolvidos. Normalmente, nessa fase é necessário muito esforço para fazer que as pessoas saiam de sua zona de conforto (“Sempre fizemos assim. Por que vamos mudar?”).
- b) *Criar uma coalizão administrativa*: é necessário juntar uma equipe com capacidade de liderança para mudar. A liderança nem sempre é avaliada em função da hierarquia organizacional, pois líderes podem ser encontrados em qualquer posição, mesmo não sendo chefes.
- c) *Desenvolver uma visão estratégica*: a “visão” implica mostrar para as pessoas envolvidas como a empresa vai ficar depois das mudanças estabelecidas. Uma pessoa que se casa, muda de emprego ou se aposenta normalmente tem uma *visão* de como será sua vida depois da mudança decidida. No caso dos envolvidos na mudança de processos, essa visão também é fundamental para que eles se engajem no processo sem a sensação de estarem andando às cegas.
- d) *Comunicar a visão da mudança*: para ser efetiva, a comunicação da visão da mudança deve ser fortemente baseada em exemplos, e não só em palavras.
- e) *Dar poder aos empregados para ações amplas*: as ações e atitudes a favor da mudança devem ser incentivadas, ao passo que as barreiras removidas e atitudes contrárias à mudança devem ser analisadas e resolvidas.
- f) *Obter vitórias de curto prazo*: se os resultados do processo de melhoria demoram muito a aparecer, a tendência é que todos os envolvidos se desmotivem. Então, é preferível ter um plano de mudança que preveja pequenas melhorias de curto prazo, que possam ser celebradas com a equipe com frequência, para manter o ânimo e a motivação para continuar.
- g) *Consolidar as melhorias e produção de mais mudanças*: as pequenas vitórias de curto prazo, porém, não devem ser motivo para “sentar nos louros” e parar o processo. Pelo contrário. Devem ser os motivadores para as verdadeiras e grandes mudanças que, muitas vezes, são necessárias. A credibilidade que a equipe vai ganhar com as pequenas mudanças será seu combustível para trabalhar nas grandes mudanças.
- h) *Estabelecer os novos processos na cultura da empresa*: as mudanças obtidas, se positivas, devem ser institucionalizadas e internalizadas, sendo adotadas naturalmente por todos os envolvidos e pelos novos contratados.

Ferreira (2011) apresenta um comparativo entre a gestão da mudança (CM) de Kotter e as fases do modelo IDEAL a que os passos de CM se aplicam (Tabela 12.5).

Ferreira conclui seu trabalho apresentando o resultado de uma pesquisa feita com 24 pessoas envolvidas em papéis de liderança em SPI, com experiência acumulada em 38 projetos. A partir dessa entrevista foram identificadas importantes ferramentas motivacionais para bem conduzir o processo de gerenciamento da mudança nas empresas de software (Tabela 12.6).

TABELA 12.5 Comparação entre os modelos IDEAL e CM¹⁵

| Modelo IDEAL | | Gestão de mudança | |
|--------------------|--|---|---|
| Fase | Pontos principais | Pontos principais | Fase |
| 1. Iniciação | Criar estímulo para a melhoria. Definir o contexto e estabelecer patrocínio para o programa. Estabelecer a infraestrutura para a melhoria. | Apresentar um esboço sobre a situação dos concorrentes. Mostrar possíveis crises sem a mudança e oportunidades advindas da mudança. Convencer pelo menos 75% dos gestores da necessidade de mudança para a organização. | 1. Mobilização dos colaboradores através do estabelecimento de um senso de urgência |
| 2. Diagnóstico | Avaliar e caracterizar o estado atual da empresa. Desenvolver as recomendações de melhoria. Definir e preencher documentos que serão a base para o plano de ação de SPI. | Delinear claramente o estado futuro da organização com as mudanças estabelecidas. Desenvolver estratégias para atingir a visão. | 2. Desenvolvimento de uma visão e de uma estratégia |
| 3. Estabelecimento | Definir as questões da SPI, estratégias, metas mensuráveis, métricas e recursos. Estabelecer o processo utilizado na implantação, as equipes técnicas e o plano de ação. | Compromisso e poder devem estar presentes na coalizão administrativa. Eles devem trabalhar fora da hierarquia normal. | 3. Criação de uma coalizão administrativa |
| 4. Ação | Planejar, executar e acompanhar o plano de ação. Definir uma solução baseada no plano de ação. Testar, aplicar o piloto, simular a realidade da empresa da melhor maneira possível. Refinar os testes resultantes da implantação piloto, e então implantar a mudança. | Incentivar atividades, ideias e ações consistentes com a mudança. Remover obstáculos (estruturas, processos, pessoas) para o processo de mudança. | 5. Capacitação dos colaboradores para ações amplas |
| 5. Alavancagem | Documentar e analisar as lições aprendidas para serem transmitidas ao novo ciclo de interação. Revisar abordagem organizacional. | Definir e destacar as melhorias resultantes das mudanças. Reconhecer e recompensar os funcionários que colaboram com o programa. | 6. Priorização de conquistas em curto prazo |
| | | Alterar as políticas e estruturas que prejudicam a visão. Promover e capacitar os colaboradores que implementaram a visão. Usar agentes de mudança e projetos para revigorar o processo de mudança. | 7. Consolidação de ganhos e produção de mais mudanças |
| | | Conectar os novos comportamentos ao sucesso da organização. Desenvolver liderança e um plano de sucessão compatíveis com a nova abordagem. | 8. Institucionalização das mudanças na cultura da empresa |

¹⁵Ferreira (2011).

TABELA 12.6 Ferramentas motivacionais para aplicar nas diferentes fases do modelo IDEAL¹⁶

| Fase do modelo IDEAL | Ferramentas motivacionais |
|----------------------|--|
| Iniciação | <i>Workshop de sensibilização:</i> para apresentar todos os dados estatísticos sobre a situação da empresa e das empresas concorrentes, possíveis crises e melhorias introduzidas com uma iniciativa de SPI. |
| Diagnóstico | <i>Reuniões de definição:</i> reuniões para identificar claramente todos os benefícios da iniciativa de SPI e caracterizar a situação da empresa no futuro, com SPI estabelecida. |
| Estabelecimento | <i>Reuniões informais:</i> reuniões regulares fora do trabalho, cuidadosamente planejadas com o intuito de descontrair e estimular a união da equipe, também chamadas de <i>happy-hours</i> . <i>Liderança com fortes características sociais e psicológicas:</i> características reconhecidas que a equipe de condução deve possuir para guiar as pessoas e integrá-las no objetivo da implantação da SPI. |
| Ação | <i>Workshops esclarecedores:</i> esses seminários devem ter o objetivo de realmente esclarecer os benefícios da implantação da SPI e assegurar que os praticantes compreendam as razões da iniciativa. <i>Fóruns de discussão:</i> nesses fóruns, as ideias opostas podem ser expressas, analisadas e discutidas abertamente, sem receio de opressão. <i>Melhor empregado:</i> os profissionais mais envolvidos com a iniciativa devem ser reconhecidos e recompensados como forma de motivação. |
| Alavancagem | <i>Quadro de conquistas:</i> uma placa, na qual todos os objetivos da SPI são apresentados, como artefatos, metas preestabelecidas, processos etc. Cada realização deve ser destacada, assim como as pessoas que ajudaram na sua obtenção. <i>Reuniões sem hierarquias:</i> reuniões para discutir a iniciativa de SPI atual e as próximas, em que todos os praticantes devem sentir-se no mesmo nível hierárquico. A administração sênior deve estar ouvindo os desenvolvedores como iguais. <i>Recompensas e benefícios para os contribuintes da iniciativa:</i> as pessoas que contribuíram com a iniciativa podem ser recompensadas com dias de folga, com o reconhecimento público ou com itens materiais, por exemplo. |

12.7 Linha de Processo de Software

Uma *Linha de Processo de Software* é uma adaptação da técnica de linha de produto de software (Seção 3.14) para a instanciação de famílias de processos que apresentam partes comuns e especificidades. Se isso não for considerado seriamente na hora de instanciar processos, poderá haver processos redundantes e/ou inconsistentes, o que representará um aumento de custo operacional.

As linhas de processo de software lidam com o problema da rápida mudança na dinâmica dos negócios, permitindo que as organizações adaptem seus processos de maneira mais organizada.

Uma linha de processo de software vai conter uma *infraestrutura* de ativos com *processos ricos em pontos de variação e modelos de decisão*.

Um processo rico em pontos de variação, por sua vez, vai conter *elementos de processo* (responsável, atividade, entradas, saídas, ferramentas, regras etc.) e *pontos de variação*.

Um elemento de processo, por sua vez, também pode ser considerado um *elemento de processo rico em pontos de variação*, caso ele próprio tenha pontos de variação. Por exemplo, um papel que, em diferentes contextos, pode ser descrito de maneira diferente ou complementar, ou ainda uma regra que, em diferentes contextos, tem diferentes formas de aplicação. No Brasil, as regras de cálculo de ICMS, por exemplo, variam de estado para estado, embora em todos os estados haja um núcleo de aplicação em comum.

Finalmente, um modelo de decisão contém *decisões*, isto é, pontos de variação que restringem a resolução de outros pontos de variação.

¹⁶Ferreira (2011).

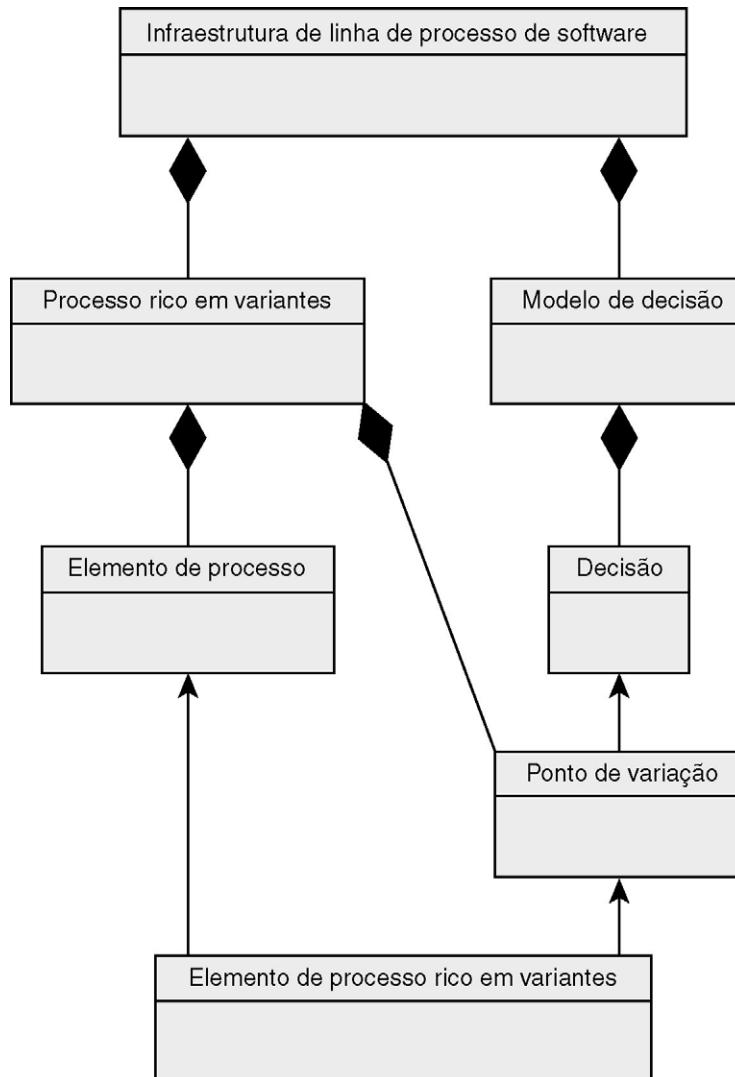


Figura 12.4 Modelo conceitual de uma linha de processo de software¹⁷.

O modelo conceitual de uma linha de processo de software (Armbrust, Katahira, Miyamoto, Münch, Nakao & Ocampo, 2009) é, então, apresentado na Figura 12.4.

Rombach (2005) sugere que linhas de processo de software se aplicam a empresas que já possuem uma área de engenharia de software bastante amadurecida, nas quais o reúso deixa de ser uma prática aplicada apenas em artefatos de projeto e passa a ser aplicado também aos próprios processos. Ele também propõe que as técnicas de linha de produto e linha de processo de software devem ser alinhadas, funcionando como um único corpo.

Essa área ainda é considerada incipiente, porém, e existem poucos exemplos de aplicações industriais de linhas de processo de software. Referências suplementares e exemplos podem ser encontrados no trabalho de Kulesza (2011).

¹⁷Ocampo e Armbrust (2009). Disponível em: <www.ove-armbrust.de/downloads/Armbrust-processlines+traceability.pdf>. Acesso em: 21 jan. 2013.

Capítulo

13

Teste

Este capítulo discute os *fundamentos* (Seção 13.1) da área de teste de software. Na sequência são apresentados os *níveis de teste de funcionalidade* (Seção 13.2) e os *testes suplementares* (Seção 13.3) mais frequentemente usados. Depois, são apresentadas técnicas específicas de teste, iniciando pelo *teste estrutural* (Seção 13.4), seguido da técnica de *teste funcional* (Seção 13.5). É apresentado o modelo de processo da técnica de *desenvolvimento dirigido por testes* (*TDD*) (Seção 13.6). Também são discutidas métricas (Seção 13.7) a serem usadas nas atividades de teste. A atividade de depuração (Seção 13.8) e a prova de correção de programas (Seção 13.9) são brevemente comentadas ao final do capítulo.

Por melhores que sejam as técnicas de modelagem e especificação de software, por mais disciplinada e experiente que seja a equipe de desenvolvimento, sempre haverá um fator que faz que o teste de software seja necessário: o *erro humano*. É um mito pensar que bons desenvolvedores, bem concentrados e com boas ferramentas serão capazes de desenvolver software sem erros (Beizer, 1990).

A Lei de Murphy (Bloch, 1977), em vários de seus enunciados, parece falar diretamente para a indústria de software:

- a) Se alguma coisa pode sair errado, sairá (no pior momento possível).
- b) Se tudo parece estar indo bem é porque você não olhou direito.
- c) A natureza sempre está a favor da falha oculta.

Durante muitos anos, a tarefa de teste de software foi considerada um castigo para os programadores. O teste era considerado uma tarefa ingrata, porque se esperava justamente que os desenvolvedores construíssem software de boa qualidade. A necessidade de testes destacava essa incapacidade que era indesejada.

A área de teste podia ser caracterizada, então, por situações caricatas como as seguintes:

- a) “Depois eu testo.”
- b) “Na minha máquina funcionou...”
- c) “Vamos deixar os testes para a próxima fase.”
- d) “Temos que entregar o produto na semana que vem” etc.

Entretanto, as coisas mudaram. Conforme visto em alguns modelos de ciclo de vida, a disciplina de teste passou a ser considerada extremamente importante. Hoje ela é parte do processo de desenvolvimento de software. Os métodos

ágeis também incorporaram o teste de software como uma atividade crítica, assumindo inclusive que os casos de teste deveriam passar a ser escritos antes das unidades de software que pretendem testar.

Além disso, grandes empresas desenvolvedoras de software passaram a contratar o teste de software de forma independente (fábricas de teste). Ou seja, os responsáveis pelo teste não são mais apenas os desenvolvedores, mas equipes especialmente preparadas para executar essa tarefa.

13.1 Fundamentos

A tarefa de testar software, porém, não é simples. Em algumas situações, pode ser mais difícil elaborar bons casos de teste do que produzir o próprio software. Assim, muita sistematização e controle são necessários para que a atividade de teste de software deixe de ser uma tarefa totalmente *ad hoc* e ingênua para se tornar uma atividade de engenharia com resultados efetivos e previsíveis.

Esta seção apresenta os fundamentos da área de teste de software, iniciando com uma fundamentação que caracteriza os termos mais comuns para evitar possíveis confusões, seguida de uma apresentação dos diferentes níveis e objetivos dos testes, bem como das principais técnicas de teste que podem ser imediatamente aplicadas, seja manualmente, seja pelo uso de ferramentas automatizadas.

13.1.1 ERRO, DEFEITO E FALHA

Inicialmente, convém definir alguns termos que, caso contrário, poderiam ser considerados sinônimos, mas que na literatura de teste têm significados bastante precisos:

- a) Um *erro* (*error*) é uma diferença detectada entre o resultado de uma computação e o resultado correto ou esperado.
- b) Um *defeito* (*fault*) é uma linha de código, bloco ou conjunto de dados incorretos que provocam um erro observado.
- c) Uma *falha* (*failure*) é um não funcionamento do software, possivelmente provocada por um defeito, mas com outras causas possíveis.
- d) Um *engano* (*mistake*), ou *erro humano*, é a ação que produz ou produziu um defeito no software.

Cabe observar aqui que o termo *fault* (defeito) algumas vezes é traduzido como *falha*, mas a falha em si (*failure*) é a observação de que o software não funciona adequadamente. Existem falhas que são provocadas por defeitos no software, mas outras que são provocadas por dados incorretos ou problemas tecnológicos (como falha de leitura, segurança ou comunicação).

Assim, nem todas as falhas são provocadas por defeitos. A área de teste de software ocupa-se principalmente das falhas provocadas por defeitos para que os defeitos sejam corrigidos e, assim, essas falhas nunca mais ocorram.

Já as falhas provocadas por causas externas ao software costumam ser assunto da área de *tolerância a falhas* (Linden, 1976). Os requisitos suplementares de tolerância a falhas de um sistema é que vão estabelecer, por exemplo, como o software se comporta no caso de uma falha de comunicação. Note que o requisito vai especificar o comportamento do software em uma situação de falha provocada externamente. Assim, caso o software se comporte de acordo com sua especificação e a falha ocorra, pode-se dizer que ela não é provocada por um defeito no software e que, pelo menos em relação a essa situação, ele *parece estar* livre de defeito.

No parágrafo anterior evitou-se afirmar categoricamente que o software está livre de defeito, porque, de fato, tal afirmação dificilmente poderá ser feita para sistemas que não sejam triviais. A maioria dos sistemas de médio e grande porte pode conter defeitos ocultos, porque a quantidade de testes necessária para garantir que estejam livres de defeitos pode ser praticamente infinita.

Assim, a área de teste de software ocupa-se em definir conjuntos finitos e exequíveis de teste que, mesmo não garantindo que o software esteja livre de defeitos, consigam localizar os mais prováveis, permitindo assim sua eliminação.

Uma máxima da área de teste de software afirma que o teste não consegue provar que o software está livre de defeitos. Ele apenas consegue provar que o software *possui* algum defeito.

13.1.2 VERIFICAÇÃO, VALIDAÇÃO E TESTE

Outra distinção que convém ser feita é entre verificação, validação e teste:

- a) *Verificação*: consiste em analisar o software para ver se ele está sendo construído de acordo com o que foi especificado.
- b) *Validação*: consiste em analisar o software construído para ver se ele atende às verdadeiras necessidades dos interessados.
- c) *Teste*: é uma atividade que permite realizar a verificação e a validação do software.

Assim, a pergunta-chave para a validação é “Estamos fazendo a coisa certa?”, enquanto a pergunta-chave para a verificação é “Estamos fazendo a coisa do jeito certo?”. No caso da validação, trata-se de saber se os requisitos incorporados ao software refletem efetivamente as necessidades dos interessados (cliente, usuário etc.). No caso da verificação, trata-se de saber se o produto criado atende aos requisitos da forma mais adequada possível, ou seja, se está livre de defeitos e possui outras características de qualidade já definidas (Seção 11.1).

Os níveis de teste vão estabelecer diferentes objetivos para as atividades de teste. A maioria dos objetivos de teste está relacionada à verificação. Em geral, apenas o teste de aceitação é efetuado visando à validação do software.

13.1.3 TESTE E DEPURAÇÃO

Enquanto a atividade de teste consiste em executar sistematicamente o software para encontrar erros desconhecidos, a *depuração* é a atividade que consiste em buscar a causa do erro, ou seja, o defeito oculto que a está causando.

O fato de se saber que o software não funciona não significa que necessariamente se saiba qual é (ou quais são) a(s) linha(s) de código que provoca(m) esse erro. A depuração pode ser uma atividade dispendiosa. Por isso, os processos mais modernos recomendam que a integração dos sistemas seja feita de forma incremental e que sejam integradas pequenas partes de código de cada vez, pois, se um sistema funcionava antes da integração e passou a falhar depois dela, o defeito provavelmente está nos componentes que acabaram de ser integrados, e não nos componentes que já funcionavam.

13.1.4 STUBS E DRIVERS

Frequentemente, partes do software precisam ser testadas de modo isolado, mas em geral elas se comunicam com outras partes.

Quando um componente *A* que vai ser testado chama operações de outro componente *B* que ainda não foi implementado, pode-se criar uma implementação simplificada de *B*, chamada *stub*, que será utilizada no lugar de *B*. Por exemplo, essa implementação simplificada pode ser uma função que, em vez de realizar um cálculo, retorna um valor predeterminado, mas que será útil para testar o componente *A*.

Suponha que *A* é uma classe que precisa usar um gerador de números primos *B*. A implementação completa de *B* capaz de gerar o *n*-ésimo número primo através da função `primo(n) : integer` pode ainda não ter sido feita. Mas, possivelmente, para testar *A* não será necessário gerar mais do que alguns poucos números primos, por exemplo, os cinco primeiros. Então *B* poderia ser substituído por uma função *stub* que gerasse apenas os 5 primeiros números primos, implementada da seguinte forma:

```
Função primo(n):integer
```

```
Caso n
```

```
1: retorna 2;  
2: retorna 3;  
3: retorna 5;  
4: retorna 7;  
5: retorna 11
```

```
Fim
```

Dessa forma, a classe *A* pode ser testada sem que *B* tenha sido efetivamente implementada.

Por outro lado, muitas vezes é o módulo *B* que já está implementado, mas o módulo *A* que chama as funções de *B* ainda não foi implementado. Nesse caso, deverá ser implementada uma simulação do módulo *A*, denominada *driver*. Essa simulação deverá chamar as funções do módulo *B* e, de preferência, executar todos os casos de teste necessários para testar *B*, de acordo com a técnica de teste adotada.

A diferença entre o *stub* e o *driver* reside na direção da relação de dependência entre os componentes. Se o componente testado depende (ou seja, chama operações) do componente simulado, então o componente simulado é um *stub*. Inversamente, se o componente simulado é que chama as operações do componente testado, então o componente simulado é um *driver*.

Assim, um *stub* e um *driver* são implementações simplificadas ou simuladas de componentes de sistema, construídas especificamente para atividades de teste. São, normalmente, um código do tipo *throw-away*, embora muito provavelmente os *drivers* devam ser guardados como patrimônio de teste para a execução de futuros testes de regressão (Seção 13.2.6). Isso porque os *drivers* terão sido construídos para testar sistematicamente as combinações de entradas mais importantes para as classes ou módulos aos quais eles estão associados, conforme será visto mais adiante neste capítulo.

■ 13.2 Níveis de Teste de Funcionalidade

Os objetivos dos testes podem variar bastante e abrangem desde verificar se as funções mais básicas do software estão bem implementadas até validar os requisitos junto ao cliente. A classificação de testes mais empregada considera que existem testes de funcionalidade (unidade, integração, sistema, ciclo de negócio, aceitação, operação etc.), que serão detalhados nesta seção, e testes suplementares (*performance*, segurança, tolerância a falhas etc.).

Os testes de funcionalidade têm como objetivo basicamente verificar e validar se as funções implementadas estão corretas nos seus diversos níveis.

13.2.1 TESTE DE UNIDADE

Os testes de unidade são os mais básicos e costumam consistir em verificar se um componente individual do software (*unidade*) foi implementado corretamente. Esse componente pode ser um método ou procedimento, uma classe completa ou, ainda, um pacote de funções ou classes de tamanho pequeno a moderado. Em geral, essa unidade está isolada do sistema do qual faz parte.

Os testes de unidade costumam ser realizados pelo próprio programador, e não pela equipe de teste. A técnica de desenvolvimento orientado a testes (Beck, 2003) recomenda inclusive que, antes de o programador desenvolver uma unidade de software, ele deve desenvolver o seu *driver* de teste, que, conforme foi visto, é um programa que obtém ou gera um conjunto de dados que serão usados para testar sistematicamente o componente que ainda vai ser desenvolvido.

O teste de unidade pode se valer bem da técnica de teste estrutural (Seção 13.4), que vai garantir pelo menos que todos os comandos e decisões do componente implementado tenham sido exercitados para verificar se eles têm defeitos. Essa técnica ainda poderá ser complementada pelo teste funcional (Seção 13.5), que vai verificar o componente relativo à sua especificação.

Um exemplo de teste de unidade consiste em verificar se um método foi corretamente implementado em uma classe. Suponha a classe *Livro* e o método *setISBN* (*umISBN : string*). A especificação do método estabelece que ele deve trocar o valor do atributo *isbn* do livro para o parâmetro passado, mas, antes disso, deve verificar se o valor do ISBN passado não corresponde a um ISBN já cadastrado (porque uma regra de negócio estabelece que dois livros não podem ter o mesmo ISBN). Assim, independentemente de ter sido implementado ou não o método *setISBN*, um caso de teste de unidade poderia ser escrito para ele da seguinte forma:

- a) Inserir um ISBN que ainda não consta na base e verificar se o atributo do livro foi adequadamente modificado.
- b) Inserir um ISBN que já existe na base e verificar se a exceção foi sinalizada como esperado.

O *driver* para testar a operação **setIsbn** poderia, então, ser escrito assim:

```
PROGRAMA DriverParaSetIsbn

(* 1 - entrada correta *)

REPITA

    isbn := geraIsbnAutomaticamente()

ATÉ NÃO existeLivroNaBaseComIsbn(isbn)

umLivro := obtemUmLivroDaBase()

umLivro.setIsbn(isbn)

SE umLivro.getIsbn() = isbn ENTÃO

    Escreva('teste 1 correto')

SENAO

    Escreva('falha encontrada no teste 1')

FIM SE

(* 2 - entrada incorreta *)

umLivro := obtemUmLivroDaBase()

REPITA

    outroLivro := obtemUmLivroDaBase()

ATÉ umLivro <> outroLivro

TENTE

    umLivro.setIsbn(outroLivro.getIsbn())

    Escreva('falha encontrada no teste 2 - não ocorreu exceção')

FIM TESTE

CAPTURE EXCEÇÃO

    SE EXCEÇÃO 'isbnInvalido' ENTÃO escreva ('teste 2 correto')

FIM CAPTURE
```

Escrever esse tipo de código para cada um dos milhares de *set* e *get* de um programa de porte médio pode ser altamente tedioso e consumir tempo. Contudo, é necessário que tal código exista, porque os testes de unidade possivelmente terão que ser repetidos inúmeras vezes ao longo do processo de desenvolvimento do software e talvez muitas outras vezes durante sua operação, para realizar testes de regressão (Seção 13.2.6).

De outro lado, esses testes de unidade podem ser extremamente reduzidos caso um gerador de código automatizado seja usado para gerar as operações padrão *get* e *set* sobre atributos ou associações. Nesse caso, assume-se que esse código estará correto, pois o gerador não comete enganos. Assim, apenas outros métodos (métodos delegados) teriam que ser testados na fase de teste de unidade. Felizmente, em sistemas de informação, *set* e *get* acabam sendo a maioria dos métodos (em quantidade, não em esforço). Assim, a economia de tempo que se pode conseguir com geradores automáticos de código, mesmo que implementados apenas para essas funções mais simples, é bastante significativa.

Uma ferramenta para a automatização de testes de unidade desenvolvida para Java, mas também adaptada para outras linguagens, é o *JUnit*¹. Trata-se de um *framework* gratuitamente distribuído. O *framework* permite inserir no programa comandos específicos de verificação que acusarão os erros, caso venham a ser encontrados.

¹Disponível em: <junit.wikidot.com/>. Acesso em: 21 jan. 2013.

Outra ferramenta é o *OCL Query-Based Debugger* (Hobatr & Malloy, 2001), que é uma ferramenta para depurar programas em C++ usando consultas formuladas em OCL (Object Constraint Language²).

Uma lista de *frameworks* de teste para mais de 70 linguagens de programação pode ser encontrada na Wikipédia³.

13.2.2 TESTE DE INTEGRAÇÃO

Testes de integração são feitos quando unidades (classes, por exemplo) estão prontas, são testadas isoladamente e precisam ser integradas em um *build* para gerar uma nova versão de um sistema.

Entre as estratégias de integração, em geral são citadas as seguintes:

- a) *Integração big-bang*: consiste em construir as diferentes classes ou componentes separadamente e depois integrar tudo junto no final. É uma técnica não incremental, utilizada no ciclo de vida Cascata com Sub-projetos. Tem como vantagens o alto grau de paralelismo que se pode obter durante o desenvolvimento e o fato de não precisar de *drivers* e *stubs* durante a integração, mas como desvantagem o fato de não ser incremental (portanto, inadequada para o Processo Unificado e métodos ágeis). Além disso, a integração de muitos componentes ao mesmo tempo pode dificultar bastante a localização dos defeitos, pois eles poderão estar em qualquer um dos componentes.
- b) *Integração bottom-up*: consiste em integrar inicialmente os módulos de mais baixo nível, ou seja, aqueles que não dependem de nenhum outro, e depois ir integrando os módulos de nível imediatamente mais alto. Assim, um módulo só é integrado quando todos os módulos dos quais ele depende já foram integrados e testados. Dessa forma, não é necessário escrever *stubs*, mas em compensação as funcionalidades de mais alto nível do sistema serão testadas tarde, quando os módulos de nível superior forem finalmente integrados.
- c) *Integração top-down*: consiste em integrar inicialmente os módulos de nível mais alto, deixando os mais básicos para o fim. A vantagem está em verificar inicialmente os comportamentos mais importantes do sistema em que repousam as maiores decisões. No entanto, como desvantagem há o fato de que muitos *stubs* são necessários – o teste, para ser efetivo, precisa de bons *stubs*, caso contrário, na integração dos módulos de nível mais baixo, poderão ocorrer problemas inesperados.
- d) *Integração sanduíche*: consiste em integrar os módulos de nível mais alto da forma *top-down* e os de nível mais baixo da forma *bottom-up*. Essa técnica reduz um pouco os problemas das duas estratégias anteriores, mas seu planejamento é mais complexo.

A principal desvantagem da maioria das técnicas (exceto a *big-bang*) é o fato de que as classes ou componentes individuais que vão ser testados necessitam comunicar-se com outros componentes ou classes que ainda não foram testados ou sequer escritos. Nesse caso, as interfaces que ainda não existem são supridas pelos *stubs*, que são implementações incompletas e simplistas e que se tornam descartáveis depois que o código real é produzido.

O problema com *stubs*, portanto, é que se perde tempo desenvolvendo software que não vai ser efetivamente entregue. Além disso, nem sempre é possível saber se a simulação produzida pelo *stub* será suficientemente adequada para os testes.

No caso do desenvolvimento iterativo, especialmente no caso dos métodos ágeis, nem sempre se sabe em que ordem os componentes serão integrados, porque vários desenvolvedores estarão trabalhando em paralelo em componentes de diferentes níveis da hierarquia de dependência. Assim, a cada integração deverá ser avaliado se o componente já possui no sistema os componentes dos quais ele depende e os componentes que dependem dele. Na falta destes, usam-se *drivers* ou *stubs*, como nos testes de unidade.

É necessário ainda estar atento às versões dos componentes do sistema, de forma que a integração sempre deixe claro quais versões de quais componentes foram efetivamente testadas juntas (Capítulo 10).

No caso de desenvolvimento utilizando o Processo Unificado e as técnicas de projeto baseadas em MVC (Wazlawick, 2011), o teste de integração deverá testar, sempre que houver integração de algum componente (novo

²Disponível em: <www.omg.org/spec/OCL/2.0/>. Acesso em: 21 jan. 2013.

³Disponível em: <en.wikipedia.org/wiki/List_of_unit_testing_frameworks> . Acesso em: 21 jan. 2013.

ou atualizado), as operações e consultas de sistema (isto é, implementadas na controladora) que utilizam funções desse componente (classe, pacote ou método). Mais adiante, a Seção 13.5 vai apresentar técnicas para de testes de operações e consultas de sistema em relação às suas especificações funcionais (contratos).

13.2.3 TESTE DE SISTEMA

O *teste de sistema* visa verificar se a versão corrente do sistema permite executar processos ou casos de uso completos do ponto de vista do usuário que executa uma série de operações de sistema em uma interface (não necessariamente gráfica) e é capaz de obter os resultados esperados.

Assim, o teste de sistema pode ser encarado como o teste de execução dos fluxos de um caso de uso expandido. Se cada uma das operações de sistema (passos do caso de uso) já estiver testada e integrada corretamente, então deve-se verificar se o fluxo principal do caso de uso pode ser executado corretamente, obtendo os resultados desejados, bem como os fluxos alternativos (Wazlawick, 2011).

É possível programar esses testes de forma automática, utilizando um módulo de programa que faz as chamadas diretamente na controladora de sistema e, dessa forma, testa todas as condições de sucesso e fracasso dos passos do caso de uso, ou ainda utilizar a interface do sistema para executar essas operações manualmente.

Considere-se, a título de exemplo, um teste de sistema a ser conduzido para avaliar o seguinte caso de uso (Wazlawick, 2011):

Caso de uso: comprar livros

1. [IN] O comprador informa sua identificação.
2. [OUT] O sistema informa os livros disponíveis para venda (título, capa e preço) e o conteúdo atual do carrinho de compras (título, capa, preço e quantidade).
3. [IN] O comprador seleciona os livros que deseja comprar.
4. O comprador decide se finaliza a compra ou se guarda o carrinho:
 - 4.1. Variante: Finalizar a compra.
 - 4.2. Variante: Guardar carrinho.

Variante 4.1: Finalizar a compra

- 4.1.1. [OUT] O sistema informa o valor total dos livros e apresenta as opções de endereço cadastradas.
- 4.1.2. [IN] O comprador seleciona um endereço para entrega.
- 4.1.3. [OUT] O sistema informa o valor do frete e total geral, bem como a lista de cartões de crédito já cadastrados para pagamento.
- 4.1.4. [IN] O comprador seleciona um cartão de crédito.
- 4.1.5. [OUT] O sistema envia os dados do cartão e valor da venda para a operadora.
- 4.1.6. [IN] A operadora informa o código de autorização.
- 4.1.7. [OUT] O sistema informa o prazo de entrega.

Variante 4.2: Guardar carrinho

- 4.2.1. [OUT] O sistema informa o prazo (dias) em que o carrinho será mantido.

Exceção 1a: Comprador não cadastrado

- 1a.1. [IN] O comprador informa seu CPF, nome, endereço e telefone.
Retorna ao passo 1.

Exceção 4.1.2a: Endereço consta como inválido

- 4.1.2a.1. [IN] O comprador atualiza o endereço.
Avança para o passo 4.1.2.

Exceção 4.1.6a: A operadora não autoriza a venda

- 4.1.6a.1. [OUT] O sistema apresenta outras opções de cartão ao comprador.
- 4.1.6a.2. [IN] O comprador seleciona outro cartão.
Retorna ao passo 4.1.5.

Esse caso de uso apresenta duas situações em que o processo termina sem percalços e que são indicadas pelas duas variantes: *guardar o carrinho de compras* e *finalizar a compra*. O processo completo, que vai do passo 1 até o final de cada uma dessas variantes (passos 4.1.7 e 4.2.1), deve ser, portanto, executado como caso de teste de sucesso.

Além das duas situações normais, os fluxos de exceção devem ser testados. A exceção 1a pode ocorrer tanto com a variante 4.1 quanto com a variante 4.2, mas normalmente não precisaria ser testada com ambas as variantes. O testador poderia escolher uma delas para testar o processo completo com a exceção 1a. No exemplo dado, poder-se-ia escolher a variante 4.2, por ser mais curta. Se a exceção ocorrer com essa variante, também vai ocorrer com a outra, pois a exceção é identificada durante a parte comum às duas variantes.

Já as outras exceções só ocorrem dentro da variante 4.1 e, portanto, serão testadas juntamente com ela.

A técnica de elaboração dos casos de teste para teste de sistema de um caso de uso consiste, então, em selecionar todos os caminhos possíveis e passar pelo menos uma vez em cada um dos fluxos alternativos (variantes e exceções). Essa técnica é bastante semelhante à técnica de caminhos independentes do teste estrutural (Seção 13.4), mas é mais simples de aplicar, porque a estrutura lógica de um caso de uso normalmente é bem mais simples do que a de um programa.

O plano de teste de sistema para esse caso de uso poderia, então, ser definido como na Tabela 13.1.

Normalmente, o teste de sistema é realizado com o uso da técnica funcional, ou seja, não se examina a estrutura interna do código, apenas a forma como o sistema se comporta em relação a sua especificação.

Considera-se que o teste de sistema somente é executado em uma versão (*build*) do sistema em que todas as unidades e os componentes recém-integrados já foram testados. Caso muitos erros sejam encontrados durante o teste de sistema, o usual é abortar o processo e refazer, ou mesmo replanejar, os testes de unidade e integração, para que uma versão suficientemente estável do sistema seja produzida e possa ser testada.

13.2.4 TESTE DE ACEITAÇÃO

O *teste de aceitação* costuma ser realizado utilizando-se a interface final do sistema. Ele pode ser planejado e executado exatamente como o teste de sistema, mas a diferença é que é realizado pelo usuário final ou cliente, e não pela equipe de desenvolvimento.

Em outras palavras, enquanto o teste de sistema faz a *verificação* do sistema, o teste de aceitação faz sua *validação*. O teste de aceitação tem como objetivo principal, portanto, a validação do software quanto aos requisitos, e não a verificação de defeitos. Ao final do teste de aceitação, o cliente poderá aprovar a versão do sistema testada ou solicitar modificações.

TABELA 13.1 Exemplo de plano de teste de sistema para um caso de uso

| Objetivo | Caminho | Como testar | Resultados |
|----------------------------------|---|--|---|
| Fluxo principal com variante 4.1 | 1, 2, 3, 4, 4.1, 4.1.1, 4.1.2, 4.1.3, 4.1.4, 4.1.5, 4.1.6, 4.1.7 | Um cliente cadastrado informa livros válidos, indica um endereço e cartão válidos, e a operadora (possivelmente um <i>stub</i>) autoriza a compra. | Compra efetuada |
| Fluxo principal com variante 4.2 | 1, 2, 3, 4, 4.2, 4.2.1 | Um cliente cadastrado informa livros válidos e guarda o carrinho. | Carrinho guardado |
| Exceção 1a | 1, 1a, 1a.1, 1, 2, 3, 4, 4.2, 4.2.1 | Um cliente não cadastrado informa livros válidos e guarda o carrinho. | O cliente é cadastrado, e o carrinho, guardado. |
| Exceção 4.1.2a | 1, 2, 3, 4, 4.1, 4.1.1, 4.1.2, 4.1.2a, 4.1.2a.1, 4.1.2, 4.1.3, 4.1.4, 4.1.5, 4.1.6, 4.1.7 | Um cliente cadastrado informa livros válidos, indica um endereço inválido e depois um endereço válido; indica um cartão válido, e a operadora (possivelmente um <i>stub</i>) autoriza a compra. | O endereço inválido é atualizado, e a compra, efetuada. |
| Exceção 4.1.6a | 1, 2, 3, 4, 4.1, 4.1.1, 4.1.2, 4.1.3, 4.1.4, 4.1.5, 4.1.6, 4.1.6a, 4.1.6a.1, 4.1.6a.2, 4.1.5, 4.1.6, 4.1.7 | Um cliente cadastrado informa livros válidos, indica um endereço e cartão válidos, e a operadora (possivelmente um <i>stub</i>) não autoriza a compra. O cliente informa outro cartão válido e a operadora autoriza a compra. | Compra efetuada com o segundo cartão informado. |

O teste de aceitação pode ter mais duas variantes:

- a) *Teste alfa*: esse teste é efetuado pelo cliente ou seu representante de forma livre, sem o planejamento e a formalidade do teste de sistema. O usuário vai utilizar o sistema e suas funções livremente e, por isso, esse teste também é chamado de *teste de aceitação informal*, em oposição ao *teste de aceitação formal*, que deveria seguir o mesmo planejamento utilizado pelo teste de sistema ([Tabela 13.1](#)).
- b) *Teste beta*: esse teste é ainda menos controlado pela equipe de desenvolvimento. Nele, versões operacionais do software são disponibilizadas para vários usuários, que, sem acompanhamento direto nem controle da empresa desenvolvedora, vão explorar o sistema e suas funcionalidades. Normalmente, versões beta de sistemas expiram após um período predeterminado, quando então os usuários são convidados a fazer uma avaliação do sistema.

Os sistemas feitos sob medida (*tailored*) devem ser testados pelo teste de aceitação formal. Já os sistemas de prateleira (*off-the-shelf*) são normalmente testados por testes alfa e beta.

13.2.5 TESTE DE CICLO DE NEGÓCIO

O *teste de ciclo de negócio* é uma abordagem possível tanto no teste de sistema quanto no teste de aceitação formal, e consiste em testar uma sequência de casos de uso que correspondem ao possível ciclo de negócio da empresa.

Assim, em vez de testar os casos de uso isoladamente, o analista ou cliente vai testá-los no contexto de um ciclo de negócio. Pode-se, por exemplo, testar a sequência de compra de um item do fornecedor, seu registro de chegada em estoque, sua venda, entrega e respectivo pagamento. São vários casos de uso relacionados no tempo, pois representam o ciclo de vida em um item em relação à empresa.

Os testes de ciclo de negócio podem ser planejados a partir de especificações de modelo de negócio (se existirem). Essas especificações costumam ser feitas em diagramas de atividade (Wazlawick, 2011) ou BPMN (*Business Process Modeling Notation*)⁴.

13.2.6 TESTE DE REGRESSÃO

O *teste de regressão* é executado sempre que um sistema em operação sofre manutenção. O problema é que a correção de um defeito no software, ou a modificação de alguma de suas funções, pode ter gerado novos defeitos. Nesse caso, devem ser executados novamente todos os testes de unidade das unidades alteradas, bem como os testes de integração e sistema sobre as partes afetadas.

No caso de manutenção adaptativa (Seção 14.2.2), pode ser necessário executar testes de aceitação, com participação do cliente. Mas, no caso da manutenção corretiva (Seção 14.2.1), tais testes não são necessários, visto que a funcionalidade do software, já aprovada pelo cliente, não muda.

O teste de regressão tem esse nome porque, se ao se aplicarem testes a uma nova versão na qual versões anteriores passaram e ela não passar, então considera-se que o sistema *regrediu*.

Como é um teste muito trabalhoso e aplicado diversas vezes ao longo do tempo de vida do software, o ideal é que o teste de regressão seja sempre automatizado (Dustin, 2002).

Algumas ferramentas que podem ser usadas para testes de regressão são o *Rational Funcional Tester*⁵, o *Mercury Quick Test Professional*⁶ e o *JUnit*⁷.

13.3 Testes Suplementares

Ao contrário dos testes de funcionalidade, que verificam e validam as funções do sistema, ou seja, os processos que podem ser efetivamente realizados, os *testes suplementares* verificam características normalmente associadas aos requisitos suplementares, como *performance*, interface, tolerância a falhas etc.

⁴Disponível em: <www.bpmn.org/>. Acesso em: 21 jan. 2013.

⁵Disponível em: <www-01.ibm.com/software/awdtools/tester/functional/>. Acesso em: 21 jan. 2013.

⁶Disponível em: <https://h10078.www1.hp.com/cda/hpm/diSPLay/main/hpms_content.jsp?zn=bto&cp=1-11-15-24^1322_4000_100>. Acesso em: 21 jan. 2013.

⁷Disponível em: <junit.wikidot.com/>. Acesso em: 21 jan. 2013.

13.3.1 TESTE DE INTERFACE COM USUÁRIO

O teste de interface com usuário tem como objetivo verificar se a interface permite realizar, com eficácia e eficiência, as atividades previstas nos casos de uso. Mesmo que as funções estejam corretamente implementadas, o que já teria sido visto no teste de sistema, isso não quer dizer que a interface também esteja. Então, em geral, é necessário testar a interface de forma objetiva e específica.

O teste de interface com usuário pode ainda verificar a conformidade das interfaces com normas ergonômicas que se apliquem, como a NBR ISO 9241-11:2011, por exemplo⁸.

No Processo Unificado, a base para a realização de testes de interface consistirá também nos casos de uso, de forma análoga ao que acontece no teste de sistema. Porém, agora, em vez de simplesmente verificar se o sistema permite a execução das funcionalidades e a obtenção dos resultados corretos, o teste deverá verificar se a interface está organizada da melhor forma possível para atender ao usuário com eficiência.

Outra vantagem do emprego de casos de uso como base para o teste de interface com usuário reside no fato de que eles apresentam a estrutura de fluxo principal de uma interação do usuário com o sistema. Assim, a interface projetada deverá procurar otimizar a sequência de ações do usuário nesse fluxo principal. Ao mesmo tempo, as ações necessárias para realizar os fluxos alternativos deverão estar disponíveis no momento em que forem necessárias, ou seja, quando o usuário estiver executando os passos do caso de uso que poderiam provocar esta ou aquela exceção.

Em outras palavras, a sequência de operações do caso de uso deve aparecer claramente na interface em uma ordem lógica, evitando-se, por exemplo, que o usuário precise focar sua atenção em diferentes áreas da interface para seguir uma simples sequência lógica. Algumas interfaces exigem que o usuário faça muitos movimentos de braço para levar o *mouse* para cima e para baixo, para a direita e para a esquerda, enquanto executa uma atividade sequencial, o que deveria ser evitado.

Da mesma forma, se ocorrerem exceções que precisarem ser tratadas pelo usuário, a mensagem deverá aparecer em local visível e próximo do foco de atenção desse usuário (são comuns sistemas que colocam mensagens de erro que não são vistas por ele), assim como os controles de interface necessários para iniciar as ações corretivas também deverão estar próximos à mensagem e ao foco de atenção do usuário sempre que possível (e não três ou quatro páginas adiante, que precisem ser roladas).

13.3.2 TESTE DE PERFORMANCE (CARGA, ESTRESSE E RESISTÊNCIA)

Com o sistema tendo ou não requisitos de desempenho, o teste de performance pode ser importante, especialmente nas operações que serão realizadas com grande frequência ou de forma iterativa. O teste consiste em executar a operação e mensurar seu tempo, avaliando se está dentro dos padrões definidos.

O teste de performance também pode ser usado para avaliar a confiabilidade e a estabilidade de um sistema. Existem, assim, diversos tipos de teste de performance:

- a) *Teste de carga*: é a forma mais simples de teste de performance. Normalmente, o teste de carga é feito para determinada quantidade de dados ou transações que deveriam ser típicos para um sistema e avalia o comportamento do sistema em termos de tempo para esses dados ou transações. Dessa forma, pode-se verificar se o sistema atende aos requisitos de performance estabelecidos e também se existem gargalos de performance para serem tratados.
- b) *Teste de estresse*: é um caso extremo de teste de carga. Procura-se levar o sistema ao limite máximo de funcionamento esperado para verificar como ele se comporta. Esse tipo de teste é feito para verificar se o sistema é suficientemente robusto em situações anormais de carga de trabalho. Ele também ajuda a verificar quais seriam os problemas encontrados caso a carga do sistema ficasse acima do limite máximo estabelecido.
- c) *Teste de resistência*: é feito para verificar se o sistema consegue manter suas características de performance durante um longo período de tempo com uma carga nominal de trabalho. Os testes de resistência devem verificar, basicamente, o uso da memória ao longo do tempo para garantir que não existam perdas acumulativas

⁸Disponível em: <www.abntcatalogo.com.br/norma.aspx?ID=86090>. Acesso em: 21 jan. 2013.

de memória em função de lixo não recolhido, e também se não existe degradação de performance após um substancial período de tempo em que o sistema opera com carga nominal ou acima dela.

Pode-se verificar também, no caso de sistemas concorrentes, o que acontece quando um número de usuários próximo ou acima do máximo gerenciável tenta utilizar o sistema. Algumas vezes, efeitos colaterais indesejados podem surgir nessas situações, por isso esses testes são considerados altamente desejáveis.

13.3.3 TESTE DE SEGURANÇA

Os *testes de segurança* costumam ser considerados parte da área de *segurança computacional*.

Basicamente, os seis tipos de segurança devem ser testados quando os requisitos de sistema assim o exigirem:

- a) *Integridade*: é uma forma de garantir ao receptor que a informação que ele recebeu é correta e completa.
- b) *Autenticação*: é a garantia de que um usuário realmente é quem ele diz ser e que os documentos, programas e sites realmente são os esperados.
- c) *Autorização*: é o processo realizado para verificar se alguma pessoa ou sistema pode ou não acessar determinada informação ou sistema.
- d) *Confidencialidade*: é a garantia de que pessoas que não têm direito à informação não poderão obtê-la.
- e) *Disponibilidade*: é a garantia de que pessoas que têm direito à informação conseguirão obtê-la quando necessário.
- f) *Não repúdio*: é uma forma de garantir que o emissor ou o receptor de uma mensagem não possam alegar, posteriormente, não ter enviado ou não ter recebido a mensagem.

13.3.4 TESTE DE RECUPERAÇÃO DE FALHA

Quando um sistema tem requisitos suplementares referentes à tolerância ou à recuperação de falhas, eles devem ser testados separadamente. Basicamente, busca-se verificar se o sistema de fato atende aos requisitos especificados relacionados a essa questão.

Normalmente, o *teste de recuperação de falha* trata de situações referentes a:

- a) queda de energia no cliente ou no servidor;
- b) discos corrompidos;
- c) problemas de comunicação;
- d) quaisquer outras condições que possam provocar a terminação anormal do programa ou a interrupção temporária de seu funcionamento.

13.3.5 TESTE DE INSTALAÇÃO

No *teste de instalação*, basicamente, busca-se verificar se o software não entra em conflito com outros sistemas eventualmente instalados em uma máquina, bem como se todas as informações e produtos para instalação estão disponíveis para os usuários instaladores.

O teste de instalação também é associado com o teste de compatibilidade, através do qual se busca verificar se o sistema é compatível com diferentes sistemas operacionais, fabricantes de máquinas, *browsers* etc.

13.4 Teste Estrutural

Em relação às técnicas de teste, existem duas grandes famílias:

- a) *Testes estruturais ou caixa-branca*: testes executados com conhecimento do código implementado, ou seja, que testam a estrutura do programa em si.
- b) *Testes funcionais ou caixa-preta* (Seção 13.5): testes executados sobre as entradas e saídas do programa sem que se tenha necessariamente conhecimento do seu código-fonte.

Essas duas famílias de teste incluem ainda várias técnicas, algumas das quais serão definidas a seguir. Procurou-se propositalmente não apresentar um tratamento muito formal e extenso a tais técnicas, embora tal tratamento exista (Delamaro, Maldonado & Jino, 2007). Pode-se, a partir desse capítulo, compreender o espírito das técnicas, sua aplicabilidade e limitações, ou seja, no nível em que um engenheiro de software deve conhecer tais técnicas.

Os primeiros testes aos quais um sistema é normalmente submetido são os testes de unidade, que têm como objetivo verificar se as funções mais simples do sistema estão corretamente implementadas. Em geral, esse tipo de teste pode ser conduzido como um teste do tipo caixa-branca, ou teste estrutural, pois o que se deseja é analisar exaustivamente a estrutura interna do código implementado.

Um dos problemas com os testes de programas é que é impossível definir um procedimento algorítmico que certifique que um programa qualquer está livre de defeitos. Esse problema é computacionalmente *indecidível* (Lucchesi, Simon, Simon & Kowaltowski, 1979). Assim, de certa forma, os testes precisam ser feitos por amostragem. Mas não se trata de amostragem aleatória. As técnicas de teste indicam como os programas devem ser testados para que se possa, com um número razoável de tentativas, avaliar se um programa contém erros.

Assim, o teste estrutural não vai testar *todas* as possibilidades de funcionamento de um programa, mas as mais representativas. Em especial, o que se busca com o teste estrutural é exercitar todos os comandos do programa e todas as condições pelo menos uma vez para cada um de seus valores possíveis (verdadeiro ou falso).

Uma primeira técnica de teste estrutural pode ser definida assim: cada estrutura de seleção (*if* ou *case*) ou repetição (*while* ou *repeat*) deve ser testada em pelo menos duas situações: quando a condição é verdadeira e quando a condição é falsa. No caso do *for* devem-se testar os casos-limite, isto é, quando a variável que limita o número de repetições assume um valor mínimo e quando ela assume um valor máximo. Se esse valor máximo for indefinido, então pode-se testar com um número arbitrariamente grande.

Assim, o teste estrutural é capaz de detectar uma quantidade substancial de possíveis erros pela garantia de ter executado pelo menos uma vez todos os comandos e condições do programa.

13.4.1 COMPLEXIDADE CICLOMÁTICA

Uma medida de complexidade de programa bastante utilizada é a *complexidade ciclomática* (McCabe, 1972), a qual pode ser definida, de forma simplificada, assim: se n é o número de estruturas de seleção e repetição no programa, então a complexidade ciclomática do programa é $n+1$. Como estruturas contam-se:

- a) IF-THEN: 1 ponto.
- b) IF-THEN-ELSE: 1 ponto.
- c) CASE: 1 ponto para cada opção, exceto OTHERWISE.
- d) FOR: 1 ponto.
- e) REPEAT: 1 ponto.
- f) OR ou AND na condição de qualquer das estruturas acima: acrescenta-se 1 ponto para cada OR ou AND (ou qualquer outro operador lógico binário, se a linguagem suportar, como XOR ou IMPLIES).
- g) NOT: não conta.
- h) Chamada de subrotina (inclusive recursiva): não conta.
- i) Estruturas de seleção e repetição em subrotinas ou programas chamados: não conta.

Assume-se, em geral, que programas com complexidade ciclomática menor ou igual a 10 são simples e fáceis de testar; com complexidade ciclomática de 11 a 20, são de médio risco em relação ao teste; de 21 a 50, são de alto risco; e que programas com complexidade acima de 50 não são testáveis⁹. Porém, deve-se tomar cuidado com a interpretação do termo “programa”. Um programa, nesse sentido, é um bloco de código único. Se ele fizer chamadas a outros blocos de código, as estruturas desses outros blocos não serão contabilizadas na complexidade ciclomática do programa em questão.

A Figura 13.1 apresenta um programa Pascal com duas estruturas de seleção e uma estrutura de repetição, portanto, com complexidade ciclomática 4 (o *else* não é contado como uma nova estrutura, pois faz parte do *if*).

⁹SEI – Software Engineering Institute. Disponível em: <www.sei.cmu.edu>. Acesso em: 21 jan. 2013.

```

01. if num = 0 then
02.     fib := 0
03. else
04.     if num = 1 then
05.         fib := 1
06.     else
07.         begin
08.             ultimoFib := 1;
09.             cont := 1;
10.             repeat
11.                 penultimoFib := ultimoFib;
12.                 ultimoFib := fib;
13.                 fib := penultimoFib + ultimoFib;
14.                 cont := cont + 1;
15.             until cont = num;
16.         end
17.     ;
18. ;
    
```

Figura 13.1 Exemplo de programa com complexidade ciclomática 4.

Na sessão seguinte é definido o grafo de fluxo de um programa, a partir do qual pode-se também calcular a complexidade ciclomática. Em caso de dúvida, na contagem de estruturas de seleção e repetição, pode-se calcular essa complexidade contando-se o número de regiões do grafo de fluxo ou, ainda, o número de nós e arestas e calculando: $2 + \text{nós} - \text{arestas}$. Nos três casos, o valor deve ser o mesmo.

13.4.2 GRAFO DE FLUXO

O *grafo de fluxo* de um programa é obtido colocando-se todos os comandos em nós e os fluxos de controle em arestas. Comandos em sequência podem ser colocados em um único nó, e estruturas de seleção e repetição devem ser representadas através de nós distintos com arestas que indiquem a decisão e a repetição, quando for o caso.

As regras da Figura 13.2 podem ser seguidas para a criação do grafo de fluxo.

Expressões booleanas compostas com OR podem ser tratadas como instruções à parte, como na Figura 13.3. No passo 1, se cond1 for falsa, então o fluxo vai para 1a, onde cond2 será testada. Se no passo 1 cond1 for verdadeira, então o fluxo vai diretamente para o passo 2. Já no nodo 1a, se cond2 for verdadeira, o fluxo vai para o passo 2, senão vai para o passo 3-4.

Simetricamente, uma condição com AND deve ser tratada como na Figura 13.4. No passo 1, se cond1 for verdadeira, então o fluxo vai para 1a, onde cond2 será testada. Se no passo 1 cond1 for falsa, então o fluxo vai diretamente para o passo 3-4. Já no nodo 1a, se cond2 for verdadeira, o fluxo vai para o passo 2, senão vai para o passo 3-4.

Nota-se que, no caso de estruturas de seleção ou repetição com OR ou AND, a complexidade ciclomática resultante é maior do que em estruturas que não tenham esses operadores binários. Assim, cada OR ou AND deve ser efetivamente contabilizado na contagem de predicados para cálculo da complexidade ciclomática.

Assim, aplicando as regras da Figura 13.2 ao programa da Figura 13.3, será obtido um grafo de fluxo semelhante ao da Figura 13.5. A figura também mostra as quatro regiões do grafo de fluxo: R1, R2, R3 e R4, cuja quantidade corresponde ao valor da complexidade ciclomática.

Outra forma de calcular essa complexidade, caso se tenha dúvida com as duas anteriores, é contar o número de arestas do grafo, subtrair o número de nodos e somar 2. No caso da Figura 13.6, o grafo possui 10 arestas e 8 nodos. Portanto, sua complexidade ciclomática é $10 - 8 + 2 = 4$.

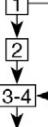
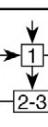
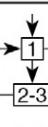
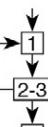
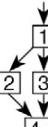
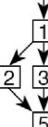
| Regra | Código | Grafo |
|--|--|---|
| Comandos em sequência (dois ou mais) | 01. <...>; 02. <...>; 03. <...>; |  |
| If-Then | 01. IF condição THEN 02. <...>; 03. ENDIF; 04. <...>; |  |
| If-Then-Else | 01. IF condição THEN 02. <...>; 03. ELSE 04. <...>; 05. ENDIF; 06. <...>; |  |
| For | 01. FOR condição DO 02. <...>; 03. ENDFOR; 04. <...>; |  |
| While | 01. WHILE condição DO 02. <...>; 03. ENDWHILE; 04. <...>; |  |
| Repeat | 01. REPEAT 02. <...>; 03. UNTIL condição; 04. <...>; |  |
| Case sem otherwise (para qualquer quantidade de casos) | 01. CASE X OF 02. a: <...>; 03. b: <...>; 04. END; 05. <...>; |  |
| Case com otherwise (para qualquer quantidade de casos) | 01. CASE X OF 02. a: <...>; 03. b: <...>; 04. OTHERWISE <...>; 05. <...>; |  |

Figura 13.2 Regras para criação de grafos de fluxo.

13.4.3 CAMINHOS INDEPENDENTES

O valor da complexidade ciclomática indica número *máximo* de execuções *necessárias* para exercitar todos os comandos do programa. No caso do exemplo da seção anterior seriam 4. Porém, a execução de todos os comandos (nós) não vai necessariamente testar os valores verdadeiro e falso de todas as condições possíveis. Por exemplo, é possível passar por todos os nós sem nunca passar pela aresta que vai dos nós 11-15 ao nó 10. Isso acontecerá se todas as vezes que o REPEAT da linha 10 for executado ele sair para a linha 16 logo após a primeira repetição.

```

01. IF cond1 OR cond2 THEN
02.   <...>;
03. ELSE
04.   <...>;
05. ENDIF;
06. <...>;
    
```

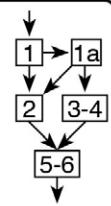


Figura 13.3 Regra para criação de grafo de fluxo para estruturas com condição OR.

```

01. IF cond1 AND cond2 THEN
02.   c1;
03. ELSE
04.   c2;
05. ENDIF;
06. c3;
    
```

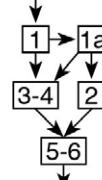


Figura 13.4 Regra para criação de grafo de fluxo para estruturas com condição AND.

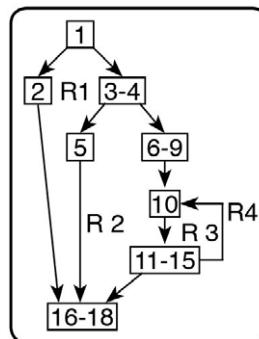


Figura 13.5 Grafo de fluxo do programa da Figura 13.1.

Assim, uma abordagem mais adequada para o teste seria exercitar não apenas todos os nodos, mas todas as *arestas* do grafo de fluxo. Isso é feito pela determinação dos *caminhos independentes* do grafo, que são possíveis navegações do início ao fim do grafo (do nodo 1 ao nodo 16-18).

O conjunto dos caminhos independentes pode ser definido da seguinte forma:

- Inicialize o conjunto dos caminhos independentes com um caminho qualquer do início ao fim do grafo (pode ser o caminho mais curto).
- Enquanto for possível, adicione ao conjunto dos caminhos independentes outros caminhos que passem pelo menos por uma aresta na qual nenhum dos caminhos anteriores passou antes.

Dessa forma, o conjunto dos caminhos independentes do grafo da Figura 13.6 pode ser definido como:

- O caminho $c_1 = \langle 1, 2, 16-18 \rangle$.
- O caminho $c_2 = \langle 1, 3-4, 5, 16-18 \rangle$.
- O caminho $c_3 = \langle 1, 3-4, 6-9, 10, 11-15, 16-18 \rangle$.
- O caminho $c_4 = \langle 1, 3-4, 6-9, 10, 11-15, 10, 11-15, 16-18 \rangle$.

A complexidade ciclomática define o número *máximo* de testes necessários para passar por todas as arestas. Com 4 testes, como mostrado, é possível exercitar todos os comandos e todas as possíveis condições das estruturas de seleção e repetição. Apesar disso, nada impede que mais testes sejam feitos, se assim for desejado.

```

01. var anos : array [1..6] of integer;
02.   coluna : integer;
03. begin
04.   coluna := 1;
05.   while (anos[coluna] <> 1996) and (coluna < 6) do
06.     coluna := coluna + 1;
07. end

```

Figura 13.6 Um programa com uma decisão baseada em múltiplas condições.

Além disso, esse número define a quantidade *máxima* de testes necessários. Isso significa que, possivelmente, um conjunto menor de testes poderia passar por todas as arestas. Por exemplo, no caso descrito, o caminho c_4 exercita os mesmos comandos e condições do caminho c_3 , pois ao final da repetição a condição do *until* será verdadeira (e isso é testado no caminho c_3). Mas convém manter o caminho c_3 , visto que ele trata de uma condição-limite da estrutura de repetição quando ela é executada apenas uma única vez.

13.4.4 CASOS DE TESTE

Resta ainda definir os *casos de teste* (Myers, Sandler, Badgett & Thomas, 2004), ou seja, quais dados de entrada levam o programa a executar os caminhos independentes e qual é a saída esperada do programa para cada um desses casos.

Considerando novamente o programa da Figura 13.1, a única entrada desse programa é a variável num, da qual se deseja obter o número de Fibonacci correspondente. Pode haver mais de uma possibilidade, mas de forma geral deve acontecer o seguinte:

- a) Para exercitar o caminho c_1 : num=0. Dará verdadeiro na condição da linha 1.
- b) Para exercitar o caminho c_2 : num=1. Dará falso na linha 1 e verdadeiro na linha 4.
- c) Para exercitar o caminho c_3 : num=2. Dará falso nas linhas 1 e 4 e verdadeiro na primeira vez que passar pela linha 15.
- d) Para exercitar o caminho c_4 : num > 2. Dará falso nas linhas 1, 4 e, pelo menos, na primeira vez que passar pela linha 15. Depois, em algum momento, dará verdadeiro na linha 15.

Um caso de teste, então, poderia ser definido como na Tabela 13.2.

Poderiam ser definidos outros testes para valores de “num” superiores a 3, mas estariam exercitando os mesmos nodos e arestas do caminho c_4 , ou seja, a mesma lógica.

13.4.5 MÚLTIPHAS CONDIÇÕES

O critério de cobertura de todas as arestas, conforme visto anteriormente, precisa cobrir também os desvios provocados pelo uso de operadores lógicos binários em estruturas de seleção e repetição, como o programa da Figura 13.6.

O programa em questão deve verificar se o valor 1996 pertence a um vetor de 6 posições. Se pertencer, o programa deve retornar à coluna em que o valor ocorre; caso contrário, deve retornar o valor 7. Esse programa, propositalmente, possui um erro de lógica muito comum embutido.

TABELA 13.2 Casos de teste

| Caminho | Entrada | Saída esperada |
|---------|---------|----------------|
| c_1 | num=0 | fib=0 |
| c_2 | num=1 | fib=1 |
| c_3 | num=2 | fib=1 |
| c_4 | num=3 | fib=2 |

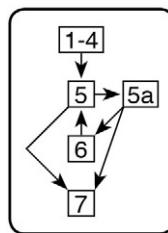


Figura 13.7 Grafo de fluxo do programa da Figura 13.6.

O grafo de fluxo do programa é mostrado na Figura 13.7.

Nesse caso, portanto, o grafo de fluxo tem três regiões e, a princípio, três testes serão o máximo necessário para verificar todas as condições e comandos.

O conjunto de caminhos independentes desse grafo poderia ser definido assim:

- a) $c_1: 1\text{-}4, 5, 7.$
- b) $c_2: 1\text{-}4, 5, 5a, 7.$
- c) $c_3: 1\text{-}4, 5, 5a, 6, 5, 7.$

Para executar o primeiro caminho independente é necessário que a condição (`anos [coluna] <> 1996`) seja falsa na primeira vez. Como a coluna será necessariamente 1 nesse ponto, então será necessário ter `anos [1] = 1996`. Assim o `while` não será executado nenhuma vez e o primeiro caminho independente estará concluído.

Para executar o segundo caminho independente, é necessário que a condição (`anos [coluna] <> 1996`) seja verdadeira, e a condição (`coluna < 6`), falsa na primeira vez que o `while` for executado. Esse caminho é impossível pela lógica do programa, pois, na primeira vez que o `while` for executado, `coluna` terá o valor 1 e a segunda condição não poderá ser falsa. A escolha do caminho independente, porém, foi feita de forma arbitrária, isto é, sem considerar as restrições lógicas do programa. Em um caso como esse, o testador deverá tentar divisar uma maneira de executar o caminho mesmo que outras arestas precisem ser adicionadas. No caso em tela, o objetivo é testar o `while` quando a primeira condição for verdadeira e a segunda condição for falsa. Isso pode ser obtido considerando-se a lógica do programa e fazendo-se que o `loop` seja executado até que `coluna` seja igual a 6. Para que a primeira condição seja verdadeira até esse ponto, é necessário que todos os valores do vetor `anos` sejam diferentes de 1996. Assim, uma entrada que poderia exercitar esse caminho seria `anos = <0, 0, 0, 0, 0, 0>`. O caminho, por sua vez, vai incluir uma sequência de repetições do padrão 5, 5a, 6. Assim, c_2 será 1-4, 5, 5a, 6, 5, 5a, 6, ..., 5, 5a, 7.

Embora o segundo caminho independente faça várias repetições do `loop`, em nenhuma delas ele repete a sequência 5, 7 esperada pelo terceiro caminho independente. Assim, esse terceiro caminho precisa efetivamente ser executado de forma independente. Para exercitá-lo é necessário que as duas condições sejam verdadeiras na primeira passada pelo `while` e a primeira condição seja falsa na segunda passada pelo `while`. Para obter esse efeito é necessário um vetor que tenha um valor diferente de 1996 na primeira posição e igual a 1996 na segunda posição.

Assim, o conjunto de casos de teste para esse programa poderia ser definido conforme a Tabela 13.3.

Pode-se observar que a execução do programa com as entradas definidas na tabela acima vai exercitar todas as combinações possíveis dos valores-verdade das condições pelo menos uma vez.

A execução dos testes com os valores de entrada definidos vai mostrar que o programa apresenta um defeito, porque o teste com o caminho c_2 vai resultar `coluna = 6`, e não `coluna = 7`, como esperado.

TABELA 13.3 Casos de teste para o programa da Figura 13.6

| Caminho independente | Entrada | Saída esperada |
|----------------------|--|-------------------------|
| c_1 | <code>anos = <1996,0,0,0,0,0></code> | <code>coluna = 1</code> |
| c_2 | <code>anos = <0,0,0,0,0,0></code> | <code>coluna = 7</code> |
| c_3 | <code>anos = <0,1996,0,0,0,0></code> | <code>coluna = 2</code> |

```

01. var anos : array [1..6] of integer;
02.   coluna : integer;
03. begin
04.   coluna := 1;
05.   while (anos[coluna] <> 1996) and (coluna < 6) do
06.     coluna := coluna + 1
07.   ;
08.   if anos[coluna]<>1996) then
09.     coluna := 7;
10. end

```

Figura 13.8 Um programa com caminhos impossíveis de se testar.

Assim, observa-se claramente que o teste aponta apenas que o programa contém um defeito, mas não diz onde ele está nem como consertá-lo. Se o programa parasse com um erro de divisão por zero, por exemplo, seria mais fácil saber onde esse defeito está, mas o programa executa até o final. Apenas o resultado é que não é o esperado.

Contudo, uma pista é dada pelo teste estrutural, porque, entre vários caminhos independentes possíveis, apenas o caminho c_2 não produziu o resultado esperado. Então, o processo de depuração deverá analisar os comandos que compõem esse caminho, bem como as condições com os respectivos valores-verdade testados.

13.4.6 CAMINHOS IMPOSSÍVEIS

Uma correção do programa da Figura 13.6 pode levar a outra situação à qual o testador deve estar atento. Algumas vezes, certos caminhos do grafo de fluxo são simplesmente impossíveis de se testar, porque a lógica do programa não permite passar por eles (Figura 13.8).

Esse programa tem duas estruturas de controle em sequência: o WHILE e o IF. Mas o IF está estruturado de forma a depender do WHILE, ou seja, só poderá ser verdadeiro se o WHILE terminar em determinada condição. Assim, o comando da linha 09 só poderá ser executado quando o WHILE sair na sexta repetição sem encontrar o valor 1996. O comando 09 é, portanto, incompatível com os caminhos c_1 e c_3 . Assim, embora o grafo de fluxo desse programa seja como mostrado na Figura 13.8, nem todos os caminhos são possíveis.

Um caminho como 1-4, 5, 7-8, 9, 10, por exemplo, embora permitido pelo grafo, seria impossível de executar em razão da lógica do programa (Figura 13.9).

Embora esse grafo tenha complexidade ciclomática 4, apenas três caminhos independentes serão necessários para testá-lo (lembrando que a complexidade ciclomática é um limite máximo):

- a) c_1 : 1-4, 5, 7-8, 10.
- b) c_2 : 1-4, 5, 5a, 6, 5, 5a, 6, ..., 5, 5a, 7-8, 9, 10.
- c) c_3 : 1-4, 5, 5a, 6, 5, 7-8, 10.

Esse tipo de problema deixa claro por que programas com complexidade ciclomática acima de 10 começam a apresentar maior risco em relação ao teste. É cada vez mais difícil construir bons casos de teste.

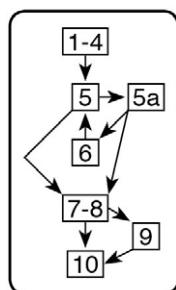


Figura 13.9 Grafo de fluxo do programa da Figura 13.8.

Outra situação em que surgem caminhos impossíveis é quando se tem um FOR com número fixo de repetições. Usualmente, um FOR no qual um dos limites (inferior ou superior) é dado por uma variável ou expressão contendo variável, gera dois caminhos possíveis: aquele em que a repetição ocorre pelo menos uma vez e aquele em que a repetição não ocorre nenhuma vez. Por exemplo:

```
for i:=1 to fim do  
  <comando>;
```

Neste caso, o primeiro caminho pode ser testado atribuindo-se qualquer valor inteiro maior ou igual a 1 à variável `fim`, e o segundo caminho pode ser testado atribuindo-se qualquer valor inteiro menor do que 1 à variável `fim`.

Mas na seguinte situação, existe apenas um caminho possível:

```
for i:=1 to 10 do  
  <comando>
```

Assim, o caminho no qual `<comando>` não é executado nenhuma vez é impossível de ser executado, porque a repetição vai ocorrer sempre 10 vezes, independentemente de quaisquer valores atribuídos às variáveis neste programa.

Esse tipo de situação também pode ocorrer com estruturas WHILE ou REPEAT que simulem uma repetição com número fixo de vezes, por exemplo:

```
x:=1;  
while x<10 do  
  begin  
    <comando>;  
    x:=x+1;  
  end;
```

Também os comandos IF e CASE podem sofrer de caminhos impossíveis, nem sempre fáceis de serem identificados. Um exemplo de IF com caminho impossível (fácil de verificar) é:

```
a:=0;  
if a>0 then  
  <comando>;
```

Qualquer programador com um mínimo de experiência poderia perceber que `<comando>` nunca será executado neste programa.

Mas há situações em que é muito mais difícil (quase impossível) saber se um comando será ou não executado, por exemplo:

```
read(a,b,c,x);  
if power(a,x)=power(b,x)+power(c,x) and x>=3 then  
  <comando>;
```

Esse programa, que testa o último teorema de Fermat (não existe nenhum conjunto de inteiros positivos a, b, c e x tal que $a^x = b^x + c^x$) trata de uma conjectura que levou 359 anos para ser respondida (Singh, 1998).

13.4.7 LIMITAÇÕES

O teste estrutural costuma ser aplicado para verificar defeitos no código, mas esse tipo de teste não é capaz de identificar, por exemplo, se o programa foi bem especificado. Ou seja, o teste vai verificar se o programa se comporta de acordo com a especificação dada, mas não necessariamente se ele se comporta de acordo com a especificação

esperada. Então, essa técnica não pode ser convenientemente usada para validar sistemas, apenas para verificar defeitos.

Outra limitação desse tipo de teste é que ele não necessariamente cobre potenciais problemas com estruturas de dados, como *arrays* e *listas*. Além disso, ele não trata necessariamente de situações típicas de programas orientados a objetos, especialmente quando se usam herança e polimorfismo. Para essas situações existem técnicas de teste específicas (Delamaro, Maldonado & Jino, 2007).

Outra limitação conhecida do teste estrutural é o fato de que funcionalidades ausentes não são testadas, pois a técnica preconiza apenas o teste daquilo que existe no programa. Ou seja, se algum comando *deveria* ter sido incluído no código, mas não foi, o teste estrutural não será capaz de identificar isso. Por exemplo, o programa de cálculo de número de Fibonacci da Figura 13.1 não testa entradas formadas por números negativos. Uma entrada negativa nesse programa fará que ele entre em *loop* infinito. Mas isso não é percebido no teste estrutural, porque apenas os caminhos efetivamente definidos no programa são testados, e não os caminhos ausentes. Além disso, algumas vezes, o programa pode estar produzindo o resultado correto para uma entrada por mera coincidência, não significando que esteja correto.

Uma última limitação também relaciona-se ao fato de que o teste estrutural só pode ser produzido *depois* que o código está escrito, o que não habilita seu uso com a técnica de desenvolvimento dirigido pelo teste (TDD), preconizada pelos métodos ágeis, que sugere que os casos de teste sejam definidos antes de se escrever o código.

Apesar dessas limitações, a técnica é relevante e seu uso é importante para a detecção de vários tipos de defeitos no software, especialmente nas suas unidades mais básicas, pois ele é usado para garantir que todos os comandos e condições lógicas sejam executados pelo menos uma vez.

13.5 Teste Funcional

O teste estrutural é adequado quando se pretende verificar a estrutura de um programa. Mas, em várias situações, a necessidade consiste em verificar a funcionalidade de um programa independentemente de sua estrutura interna. Um programa pode ter uma especificação ou comportamento esperado, em geral elaborado em um contrato de operação de sistema, e o que se deseja saber é se ele efetivamente cumpre esse contrato ou especificação. O teste estrutural é adequado para uma verificação em primeira mão de unidades (métodos, procedimentos ou algoritmos), ou seja, dos elementos mais básicos do software.

Em orientação a objetos, em especial, o teste estrutural pode ser aplicado às operações básicas das classes, que criam e destroem instâncias, adicionam ou removem associações, ou alteram o valor de atributos. Tais testes também são adequados para as consultas básicas, que retornam o valor de atributos normais ou derivados, bem como o de associações normais ou derivadas (Wazlawick, 2011).

Em um nível mais alto de operações, porém, tal verificação estrutural pode ser difícil de realizar, pois métodos podem chamar métodos de outras classes, delegando responsabilidades de um objeto a outro. Assim, no nível de integração de funções mais básicas do software será mais adequado realizar *testes funcionais*, que avaliam o comportamento de uma operação mais abrangente do que as operações elementares, porque essa técnica pode avaliar comportamentos que estão distribuídos em várias classes (coisa que a técnica estrutural não faz diretamente).

O padrão MVC (*Model View Control*), bastante empregado em sistemas de informação, sugere que um sistema deve ser dividido em pelo menos três camadas, sendo que a superior (*View*) corresponde à interface com o usuário, e a intermediária (*Control*) é a que efetivamente se responsabiliza pela execução de processamento lógico sobre a informação. Com o uso dessa técnica, todas as transformações da informação vão ocorrer encapsuladas pela controladora (uma classe com a função específica de encapsular esses comportamentos) e serão acessíveis pela interface através de chamadas a métodos dessa classe. Quando os métodos fazem alteração de dados, são chamados de *operações de sistema*; quando fazem consulta a dados, são chamados de *consultas de sistema*.

Técnicas de análise de sistemas (Wazlawick, 2011; Larman, 2001) recomendam que essas operações e consultas sejam especificadas por contratos bem definidos com parâmetros tipados, pré-condições, pós-condições e exceções. O teste funcional, então, consistirá em verificar se em situação de normalidade (pré-condições atendidas) as pós-condições desejadas são realmente obtidas, e se em situações de anormalidade as exceções são efetivamente levantadas.

13.5.1 PARTICIONAMENTO DE EQUIVALÊNCIA

Um dos princípios do teste funcional é a identificação de situações equivalentes. Por exemplo, se um programa aceita um conjunto de dados (normalidade) e rejeita outro conjunto (exceção), pode-se dizer que existem duas *classes de equivalência* para os dados de entrada do programa: os dados aceitos e os dados rejeitados. Pode ser impossível testar todos os elementos de cada conjunto¹⁰, até porque esses conjuntos podem ser infinitos. Então, o *particionamento de equivalência* vai determinar que pelo menos um elemento de cada conjunto seja testado.

Classicamente, a técnica de particionamento de equivalência considera a divisão das entradas possíveis da seguinte forma (Myers, Sandler, Badgett & Thomas, 2004):

- a) Se as entradas válidas são especificadas como um *intervalo de valores* (por exemplo, de 10 a 20), então são definidos um conjunto válido (de 10 a 20) e dois inválidos (menor do que 10 e maior do que 20).
- b) Se as entradas válidas são especificadas como uma *quantidade de valores* (por exemplo, uma lista com 5 elementos), então são definidos um conjunto válido (lista com 5 elementos) e dois inválidos (lista com menos de 5 elementos e lista com mais de 5 elementos).
- c) Se as entradas válidas são especificadas como um *conjunto de valores aceitáveis* que podem ser tratados de forma diferente (por exemplo, os *strings* “masculino” e “feminino”), então são definidos um conjunto válido para cada uma das formas de tratamento e um conjunto inválido para outros valores quaisquer.
- d) Se as entradas válidas são especificadas como uma condição do tipo “deve ser de tal forma” (por exemplo, uma restrição sobre os dados de entrada como “a data final deve ser posterior à data inicial”), então devem ser definidos um conjunto válido (quando a condição é verdadeira) e um inválido (quando a condição é falsa).

Os conjuntos de valores válidos devem ser definidos não só em termos de restrições sobre as entradas, mas também em função dos resultados a serem produzidos. Se a operação a ser testada puder ter dois comportamentos possíveis em função do valor de um dos parâmetros, vão existir dois conjuntos distintos de valores válidos para aquele parâmetro, um para cada comportamento possível.

Por exemplo, considere um contrato de operação de sistema “adicionaLivro(idLivro, idCompra, quant)” que toma da interface três valores: um identificador de livro (idLivro), um identificador de uma compra (idCompra) e uma quantidade (quant). O objetivo da operação consiste em adicionar à compra indicada um item que associe o livro indicado e uma quantidade. A Figura 13.10 apresenta o modelo conceitual de referência para esse exemplo. Apenas os atributos relevantes para o exemplo foram usados, embora as classes originais pudessem ter vários outros.

Em função das entradas, os seguintes conjuntos de valores válidos e inválidos podem ser definidos:

- a) Para idLivro:
 - Válido: se existe um livro com esse id.
 - Inválido: se tal livro não existe.
- b) Para idCompra:
 - Válido: se existe uma compra com esse id e ela está aberta.
 - Inválido: caso a compra não exista.
 - Inválido: se a compra já está fechada.
- c) Para quantidade:
 - Válido: se for um valor maior ou igual a 1.
 - Inválido: para zero.

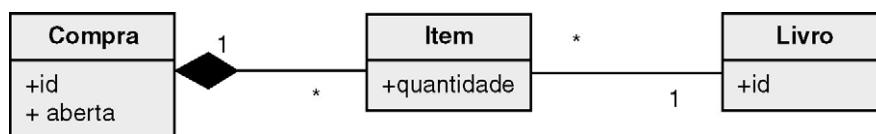


Figura 13.10 Modelo conceitual de referência para exemplo.

¹⁰A literatura geralmente fala em *classes* de elementos aceitos e rejeitados, que é um conceito matemático. Aqui será usada a palavra *conjunto* para que não haja confusão com o conceito de *classe* de orientação a objetos.

Em relação aos resultados da operação, pode-se ainda considerar que, se o livro já consta da compra, em vez de criar um novo item deve-se somar a quantidade solicitada com a quantidade já existente. Ou seja, existem dois comportamentos possíveis para um parâmetro de entrada, dependendo do valor deste e do estado interno dos objetos. Nesse caso, pode-se subdividir o conjunto de valores válidos relacionados com `idLivro` em dois conjuntos válidos: um quando o `idLivro` passado não consta na compra e outro quando o `idLivro` passado já consta da compra. Isso resulta na seguinte subdivisão:

a) Para `idLivro`:

- Válido: se existe um livro com esse id e ele não consta da compra.
- Válido: se existe um livro com esse id e ele já consta da compra.
- Inválido: se tal livro não existe.

Assim, os casos de teste poderiam ser definidos em termos desses conjuntos válidos e inválidos conforme a Tabela 13.4.

Nem todas as combinações de conjuntos válidos e inválidos para todos os parâmetros são testadas. Normalmente, testam-se todas as combinações de conjuntos válidos, como nas duas primeiras linhas da Tabela 13.4, e acrescenta-se uma possível classe inválida de cada vez, que corresponde às demais linhas da tabela.

Assim, em geral, não se testam condições de erro que ocorrem combinadas, ou seja, duas ou mais condições de erro de cada vez. Isso ocorre porque a combinação de todos os conjuntos válidos e inválidos cresce exponencialmente em relação à quantidade deles. Ou seja, cada novo conjunto válido ou inválido identificado tecnicamente multiplica o número de testes necessários. No exemplo acima, a condição *a* revisada tem 3 conjuntos, a condição *b* também tem 3 conjuntos, e a condição *c* tem dois conjuntos, resultando, portanto, em $3 \times 3 \times 2 = 18$ combinações possíveis, caso todas fossem ser testadas. Se houvesse apenas mais um conjunto na condição *c*, haveria $3 \times 3 \times 3 = 27$ combinações possíveis.

Então, embora não se testem todas as combinações, é necessário testar pelo menos:

- todas as combinações de conjuntos válidos;
- todas as combinações de conjuntos válidos com cada um dos conjuntos inválidos.

Assim, o testador é obrigado a incluir cada conjunto inválido em pelo menos um teste. Mas, se por algum motivo, ele quiser combinar conjuntos inválidos também, isso não é proibido, apenas poderá ser mais trabalhoso e possivelmente pouco efetivo, já que usualmente as exceções são detectadas uma de cada vez.

13.5.2 ANÁLISE DE VALOR LIMITE

Um dos ditados em teste de software diz que os *bugs* (insetos, em português) costumam se esconder nas *frestas*. Em função dessa realidade, a técnica de partição de equivalência normalmente é usada em conjunto com o critério de análise de valor-limite.

TABELA 13.4 Casos de teste funcional para uma operação de sistema

| Tipo | <code>idLivro</code> | <code>idCompra</code> | <code>quant</code> | Resultado esperado |
|---------|--|---------------------------------------|---------------------------|---|
| Sucesso | Um id já cadastrado que não consta da compra | Um id já cadastrado de compra aberta | Qualquer valor acima de 0 | Um item é criado e associado à compra e ao livro, com atributo quantidade = quant |
| Sucesso | Um id já cadastrado que consta da compra | Um id já cadastrado de compra aberta | Qualquer valor acima de 0 | O item do livro que já consta da compra tem seu valor incrementado com quant |
| Exceção | Um id não cadastrado | Um id já cadastrado de compra aberta | Qualquer valor acima de 0 | Exceção: livro não cadastrado |
| Exceção | Um id já cadastrado | Um id não cadastrado | Qualquer valor acima de 0 | Exceção: compra não cadastrada |
| Exceção | Um id já cadastrado | Um id já cadastrado de compra fechada | Qualquer valor acima de 0 | Exceção: compra fechada |
| Exceção | Um id já cadastrado | Um id já cadastrado de compra aberta | 0 | Exceção: quantidade não pode ser zero |

A análise de valor-limite consiste em considerar não apenas um valor qualquer para teste dentro de uma classe de equivalência, mas um ou mais valores fronteiriços com outras classes de equivalência quando isso puder ser determinado.

Em domínios ordenados (números inteiros, por exemplo), esse critério pode ser aplicado. Por exemplo, se um programa exige uma entrada que, para ser válida, deve estar no intervalo $[n..m]$, então existem três classes de equivalência:

- a) Inválida para qualquer $x < n$.
- b) Válida para qualquer $x \geq n$ e $x \leq m$.
- c) Inválida para qualquer $x > m$.

A análise de valor-limite sugere que possíveis erros de lógica do programa não vão ocorrer em pontos arbitrários dentro desses intervalos, mas nos pontos em que um intervalo se encontra com outro. Então:

- a) Para a primeira classe inválida, deve-se testar para o valor $n - 1$.
- b) Para a classe válida, devem-se testar os valores n e m .
- c) Para a segunda classe inválida, deve-se testar para o valor $m + 1$.

Assim, se existir um erro no programa para alguma dessas classes, é muito mais provável que ele seja capturado dessa forma do que se fosse selecionado um valor qualquer dentro de cada um desses três intervalos.

13.6 TDD – Desenvolvimento Orientado a Testes

TDD (*Test Driven Development*) ou *Desenvolvimento Orientado a Testes* (Beck, 2003) é uma técnica ou filosofia de programação que incorpora o teste ao processo de produção de código da seguinte forma:

- a) Primeiramente o desenvolvedor, que recebe a especificação de uma nova funcionalidade a ser implementada, deve programar um conjunto de testes automatizados para testar o código que ainda não existe.
- b) Esse conjunto de testes deve ser executado e falhar. Isso é feito para mostrar que os testes efetivamente têm algum poder de teste e não terão sucesso a não ser que o código específico seja desenvolvido. Se o código passar em algum desses testes, é possível que ou o teste não seja efetivo, ou que a característica a ser implementada já exista no sistema.
- c) Em seguida, o código deve ser desenvolvido da forma mais minimalista possível, isto é, apenas para passar nos testes.
- d) Depois que o código passar em todos os testes, deve ser refatorado para atender a padrões de qualidade interna e testado novamente até que passe em todos os testes novamente.

A Figura 13.11 apresenta o diagrama de atividades para esse tipo de processo.

Um dos pontos-chave nesse processo é que, depois que os testes foram estabelecidos, o programador não deve implementar nenhuma funcionalidade além daquelas para as quais os testes foram criados. Isso evita que os programadores percam tempo criando estruturas que efetivamente não eram necessárias desde o início. Isso satisfaz ao princípio KISS (*Keep it Simple, Stupid!*)¹¹, que indica que seria estúpide fazer algo mais do que o mais simples necessário em um projeto, e também o princípio YAGNI (*You Ain't Gonna Need It*)¹².

Para escrever um conjunto de testes efetivo, é necessário que o programador primeiro comprehenda perfeitamente os requisitos e a especificação do componente que vai desenvolver. Nesse sentido, os contratos de operação de sistema e o teste funcional correspondente podem ajudar muito.

A técnica é bastante exigente em relação ao processo a ser seguido. Por exemplo, um programador que descubra a necessidade de adicionar um ELSE a um IF em um código já aprovado nos testes deverá, antes de escrever o código, escrever um teste para o ELSE, certificar-se de que o teste falha para o programa atual e somente depois adicionar o ELSE ao código. Pular etapas não é encorajado pela técnica.

¹¹“Mantenha isso simples, estúpido!”. Disponível em: <en.wikipedia.org/wiki/KISS_principle>. Acesso em: 21 jan. 2013.

¹²“Você não vai precisar disso”. Disponível em: <en.wikipedia.org/wiki/You_ain%27t_gonna_need_it>. Acesso em: 21 jan. 2013.

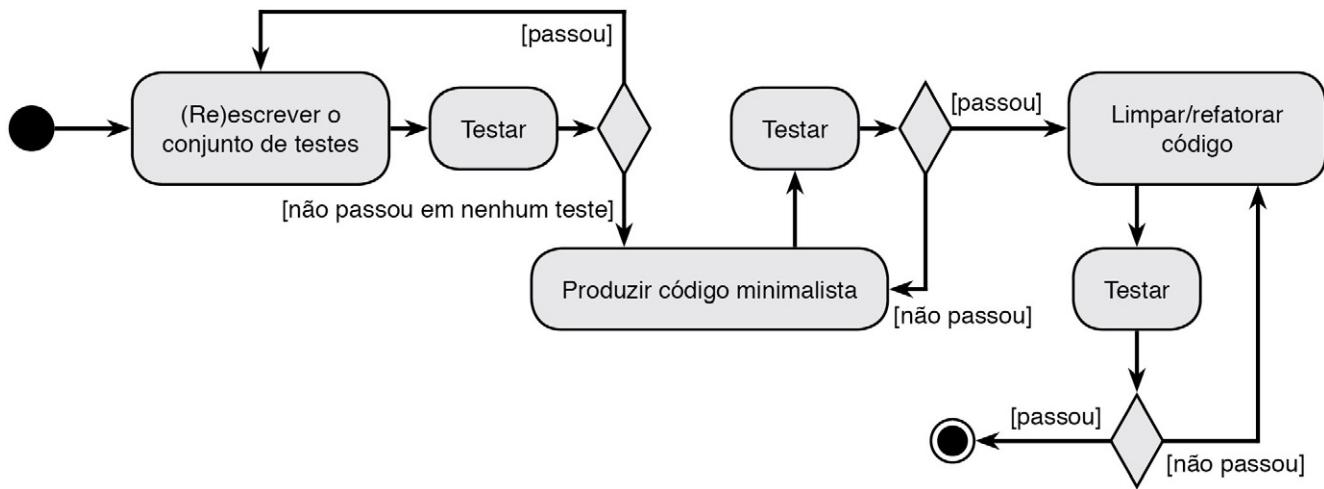


Figura 13.11 Processo de desenvolvimento orientado a testes.

Um desenvolvimento extra de TDD pode ser atingido quando se consegue automatizar os testes de aceitação do usuário. Essa abordagem, conhecida com ATDD (*Acceptance Test-Driven Development*; Koskela, 2007), disponibiliza aos desenvolvedores a garantia de que os requisitos estão sendo atendidos, mesmo que o usuário não esteja presente todo o tempo.

13.7 Medição em Teste

Uma informação importante a ser relatada por equipes de teste é o *status* da atividade de teste (Crowther, 2012), o que deve ser feito de forma precisa e compreensiva. Então, nesse caso, aplica-se também o conceito de métrica para avaliação objetiva de *status*. Pode-se falar em dois grandes conjuntos de métricas de teste:

- a) *Métricas do processo de teste*: as medidas relacionadas reportam a quantidade de testes requeridos, planejados, executados etc., mas não o estado do produto em si.
- b) *Métricas de teste do produto*: as medidas relacionadas reportam o estado do produto em relação à atividade de teste, por exemplo, quantos defeitos foram encontrados, quantos estão em revisão, quantos foram resolvidos etc.

As seguintes medições relacionadas às métricas de processo de teste poderiam ser apresentadas durante a preparação e a execução dos testes:

- a) Número de testes previstos (sua necessidade já foi identificada).
- b) Número de testes planejados (casos de teste definidos).
- c) Número de testes executados, incluindo: a) testes que falharam e b) testes que foram aprovados.

Essas métricas podem ser comparadas ou relativizadas com outras métricas usualmente tomadas sobre o processo, como o tempo investido nas atividades, o número de pontos de função etc.

Alguns exemplos de métricas de teste relativas ao produto são:

- a) Número de defeitos descobertos.
- b) Número de defeitos corrigidos.
- c) Distribuição dos defeitos por grau de gravidade.
- d) Distribuição dos defeitos por módulo.

As medições relacionadas a essas métricas podem ser apresentadas em gráficos que permitem visualizar o desempenho das equipes de desenvolvimento e testes, em que se espera que o número de defeitos descobertos seja

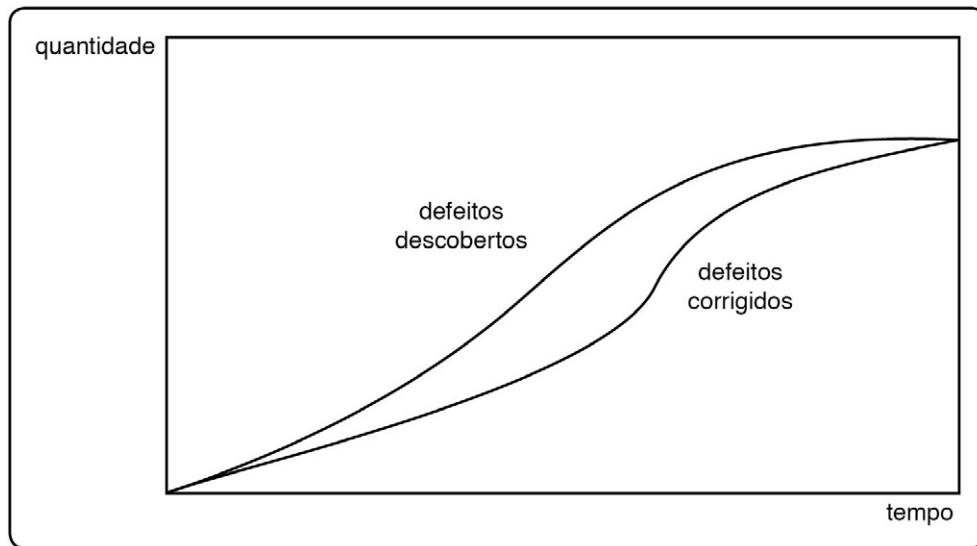


Figura 13.12 Evolução esperada das medidas de defeitos de produto ao longo de um projeto.

relativamente constante no tempo e que as atividades de correção atinjam seus objetivos também de forma constante, até que o produto esteja suficientemente livre de defeitos para ser implantado (Figura 13.12).

Outras medidas relacionadas a teste são o tempo gasto com as atividades de teste (isto é, o custo dos testes) e a cobertura dos testes, isto é, quantas funcionalidades do sistema tiveram seus testes efetivamente planejados, executados e aprovados.

O site *Software Testing Stuff*¹³ apresenta várias métricas compostas, como:

- Custo para encontrar um defeito = esforço total com atividades de teste/número de defeitos encontrados.
- Adequação dos casos de teste = número de casos de teste real/número de casos de teste estimado.
- Efetividade dos casos de teste = número de defeitos encontrados através de testes/número total de defeitos encontrados.

13.8 Depuração

Depuração é o processo de remover defeitos de um programa. Normalmente se inicia com atividades de teste, que identificam o problema, a partir de inspeções de código, ou ainda a partir de relatos de usuários. No último caso, a equipe deve, no início do processo de depuração, tentar reproduzir o defeito relatado pelo usuário, o que nem sempre é possível ou fácil.

A depuração é uma atividade artesanal em que a quantidade e a variedade de casos dificultam a elaboração de um processo padrão. Porém, a adoção de boas técnicas de engenharia de software, como integrações frequentes, controle de versões, desenvolvimento orientado a testes etc., podem facilitar bastante o processo de descoberta e correção de defeitos.

A tarefa de depuração pode ser tão simples quanto corrigir uma *string* com erro ortográfico ou tão complexa a ponto de exigir um trabalho de pesquisa, coleta de dados e formulação de hipóteses digna dos melhores detetives.

De forma geral, ferramentas, conhecidas como *debuggers*, ajudam no trabalho de depuração, pois permitem executar programas passo a passo, observando o valor das variáveis, parar, retornar etc. A Wikipédia¹⁴ apresenta uma extensa lista de tais ferramentas para várias linguagens.

¹³Disponível em: <www.softwaretestingstuff.com/2007/10/software-testing-metrics.html>. Acesso em: 21 jan. 2013.

¹⁴Disponível em: <en.wikipedia.org/wiki/Comparison_of_debuggers> . Acesso em: 21 jan. 2013.

13.9 Prova de Correção de Programas

Como foi visto neste capítulo, o teste de software ajuda a encontrar defeitos no software, mas não *garante* que ele esteja livre de defeitos. Uma abordagem que forneça esse tipo de garantia em relação a essa especificação é possível, mas tem alto custo. Por isso, em geral, só é aplicada em sistemas e componentes críticos nos quais o erro não é aceitável.

Uma das técnicas atualmente usadas para a prova de correção de programas é o *Cálculo de Hoare* (Hoare, 1969)¹⁵, que consiste em uma técnica de definição de semântica axiomática para programas. A principal ideia do cálculo é colocar um comando ou fragmento de programa entre pré-condições e pós-condições, da seguinte forma:

$$\{P\} C \{Q\}$$

em que P são as pré-condições, Q são as pós-condições e C é o comando a ser testado.

Nesse caso, afirma-se que o comando C está correto quando os valores de entrada passam por P , resultando em verdadeiro, e, depois de C ser executado, os valores de saída passam por Q , também resultando em verdadeiro.

O cálculo de Hoare utiliza um conjunto de regras baseadas em técnicas de análise de algoritmos:

- a) *Comando skip* (\cdot): um fragmento de programa que não executa nada tem pré e pós-condições idênticas. Portanto, $\{P\} ; \{P\}$ é um axioma do cálculo.
- b) *Comando de atribuição*: se uma variável x recebe uma atribuição com uma expressão f , então qualquer propriedade que fosse verdadeira para f antes da atribuição será verdadeira para x após a atribuição (mas não mais necessariamente verdadeira para f , já que a expressão f pode conter ocorrências de x). Assim, $\{P(f)\} x := f \{P(x)\}$ será um esquema de axioma, que define um conjunto infinito de axiomas do cálculo para todas as variáveis e expressões possíveis.
- c) *Comando if-then-else*: no caso de comandos *if-then-else*, há dois caminhos, e a regra de inferência indica que cada um deles deve manter as pós-condições esperadas. Ou seja, considerando-se a expressão $\{P\} \text{ if } C \text{ then } X \text{ else } Y \text{ endif } \{Q\}$, essa sequência de pré-condição, comando e pós-condição somente será verdadeira se antes for verdadeiro $\{P \text{ AND } C\} X \{Q\}$ e se também for verdadeiro $\{Q \text{ AND NOT } C\} Y \{Q\}$.
- d) *Comando if-then*: a regra de *if-then* é idêntica à de *if-then-else*. Apenas se substitui o comando que estaria no *else* por um comando *skip*. Então, $\{P\} \text{ if } C \text{ then } X \text{ endif } \{Q\}$ será verdadeiro se antes $\{P \text{ AND } C\} X \{Q\}$ for verdadeiro e $\{Q \text{ AND NOT } C\} ; \{Q\}$ for verdadeiro.
- e) *Comando while*: um comando como *WHILE C DO X ENDWHILE* será executado até que a condição C seja falsa. Então, se $\{P \text{ AND } C\} X \{P\}$ for verdadeira, $\{P\} \text{ WHILE } C \text{ DO } X \text{ ENDWHILE } \{P \text{ AND NOT } C\}$ também deverá ser verdadeiro, onde P é uma invariante do *loop* (Meyer, 1997), ou seja, uma expressão que deve ser verdadeira ao longo de toda a execução do laço.
- f) *Enfraquecimento da pós-condição*: se $\{P\} X \{Q\}$ for verdadeiro e Q implica R , então $\{Q\} X \{R\}$ também é verdadeiro.
- g) *Enfraquecimento da pré-condição*: se P implica Q e $\{Q\} X \{R\}$ for verdadeiro, então $\{P\} X \{R\}$ também será verdadeiro.

Imperial (2003)¹⁶ apresenta exemplos de aplicação dessas regras para a prova formal de propriedades de programas. Apenas para exemplificar, Leite (2000) apresenta um exemplo de aplicação da regra do comando *while*. Deseja-se provar que, se x for inicialmente maior ou igual a zero, o programa abaixo fará que ele seja igual a zero ao final:

```
while x>0 do
    x := x-1
end while
```

¹⁵Disponível em: <sunnyday.mit.edu/16.355/Hoare-CACM-69.pdf>. Acesso em: 21 jan. 2013.

¹⁶Disponível em: <www2.dbd.puc-rio.br/pergamum/tesesabertas/0124811_03_cap_03.pdf>. Acesso em: 21 jan. 2013.

Nesse caso, a pré-condição P é $\{x \geq 0\}$, a condição C é $\{x > 0\}$, e assim $\{P \text{ AND } C\}$ fica: $\{x \geq 0 \text{ AND } x > 0\}$. O comando X é $x := x - 1$, e a pós-condição $\{P \text{ AND NOT } C\}$ será $\{x \geq 0 \text{ AND NOT } x > 0\}$, ou seja, $\{x = 0\}$. Aplicando-se a regra do *while*, tem-se que:

$$\{x \geq 0 \text{ AND } x > 0\} x := x - 1 \{x \geq 0\} \text{ implica } \{x \geq 0\} \text{ while } x > 0 \text{ do } x := x - 1 \text{ endwhile } \{x \geq 0 \text{ AND NOT } x > 0\}$$

Assim, como o lado esquerdo da implicação é verdadeiro, pois se x é maior que zero, subtrair 1 de x fará que continue sendo maior ou igual a zero, a seguinte conclusão pode ser obtida:

$$\{x \geq 0\} \text{ while } x > 0 \text{ do } x := x - 1 \text{ endwhile } \{x = 0\}$$

Dessa forma, comprova-se, pela aplicação do axioma do *while*, que o programa satisfaz a sua especificação.

Como no caso do método *cleanroom* (Seção 11.4.3), existe certa dificuldade para encontrar desenvolvedores capacitados a aplicar especificação formal a software, e o alto custo da realização dessas provas faz que acabem sendo utilizadas mais fortemente apenas em aspectos de segurança crítica de sistemas.

Manutenção e Evolução de Software

Este capítulo apresenta os conceitos de *manutenção e evolução de software*, iniciando pelas *Leis de Lehman* (Seção 14.1), que procuram explicar a necessidade e as características do processo de manutenção e evolução. Em seguida são caracterizadas as quatro *formas de manutenção em função dos objetivos* (Seção 14.2). O *processo de manutenção* também é explicado (Seção 14.3), bem como um conjunto de *ferramentas* úteis para manutenção de software (Seção 14.4). Finalmente são apresentados os diferentes tipos de *atividades* de manutenção e suas *métricas* (Seção 14.5), além de *modelos para estimativa de esforço* em atividades de manutenção e evolução (Seção 14.6). O capítulo termina com a apresentação dos conceitos relacionados a *reengenharia* e *engenharia reversa* (Seção 14.7), que são técnicas frequentemente usadas em processos de manutenção de software.

Manutenção de software é como se denomina, em geral, o processo de adaptação e otimização de um software já desenvolvido, bem como a correção de defeitos que ele possa ter. A manutenção é necessária para que um produto de software preserve sua qualidade ao longo do tempo, pois se isso não for feito haverá uma deterioração do valor percebido desse software e, portanto, de sua qualidade.

Classicamente, a atividade de manutenção é considerada o conjunto de modificações que o software sofre depois de ter sido terminado, ou seja, após o final da fase de transição no UP. Porém, o EUP (Seção 5.6) e o OUM (Seção 5.7) adicionam ao Processo Unificado uma fase chamada “Produção”, que ocorre justamente no momento em que o software está em operação e necessita de manutenção. Assim, as atividades de manutenção são consideradas tão relevantes e necessárias quanto as atividades de desenvolvimento para o sucesso do produto.

Modernamente, a expressão “manutenção de software” vem sendo substituída ou utilizada em conjunto com “evolução de software”. *Evolução* talvez seja um termo mais adequado, porque as atividades de modificação do software na fase de produção não visam mantê-lo como está, mas fazê-lo evoluir de forma a adaptar-se a novos requisitos ou ainda a corrigir possíveis defeitos.

Em uma interpretação do termo, pode-se considerar “manutenção” o conjunto de tarefas individuais de modificação de um software em uso. Nesse caso, a “evolução” seria o processo de mais longo prazo, ou seja, a evolução do software pode ser vista como uma sequência de manutenções ao longo do tempo.

De outro lado, há autores (Erdil, Finn, Keating, Meattle, Park & Yoon, 2003) que acreditam que apenas as correções de erros (manutenção corretiva) podem ser consideradas atividades tradicionais de manutenção, enquanto a otimização, a adaptação e a prevenção de erros são consideradas evolução. Essa interpretação será seguida neste livro.

14.1 Necessidade de Manutenção e Evolução de Software (Leis de Lehman)

Considera-se que, uma vez desenvolvido, um software terá um valor necessariamente decrescente com o passar do tempo. Isso ocorre porque:

- a) Falhas são descobertas.
- b) Requisitos mudam.
- c) Produtos menos complexos, mais eficientes ou tecnologicamente mais avançados são disponibilizados.

Dessa forma, torna-se imperativo que, simetricamente, para manter o valor percebido de um sistema:

- a) Falhas sejam corrigidas.
- b) Novos requisitos sejam acomodados.
- c) Sejam buscadas simplicidade, eficiência e atualização tecnológica.

A realização dessas atividades é parte do processo de manutenção e evolução de software.

As *Leis de Lehman* (Lehman M. M., 1980; Lehman & J. F. Ramil, 1997) procuram explicar a necessidade e a inevitabilidade da evolução de software. São atualmente oito leis, baseadas nas observações do autor sobre os processos de evolução de sistemas. Elas foram apresentadas em várias publicações entre 1974 e 1996, até chegarem à forma atual.

Inicialmente, Lehman identifica dois tipos de sistema:

- a) *Tipo-S*: sistemas especificados formalmente e entendidos como objetos matemáticos, cuja correção em relação a uma especificação pode ser provada por ferramentas formais.
- b) *Tipo-E*: sistemas desenvolvidos pelos processos usuais de análise, projeto e codificação, que têm uso corrente em um ambiente real, isto é, são tipicamente sistemas de informação e outros sistemas não gerados por métodos formais.

Entende-se, então, que os sistemas do tipo-S são diferentes em relação à possibilidade de terem defeitos, pois apenas defeitos relacionados à especificação ausente podem ocorrer. Um exemplo de sistema de tipo-S pode ser um pacote de funções matemáticas ou de cálculo de equações físicas, ou ainda de engenharia, que tenha sido desenvolvido através de métodos formais.

Já os sistemas de tipo-E são propensos a defeitos em seu processo de produção. Além disso, pelo fato de serem usados em conexão com o mundo real, esses sistemas podem precisar evoluir para atender a requisitos que mudam conforme o contexto de uso, por exemplo, no caso de leis que mudam, ou objetivos da empresa, ou ainda expectativas dos usuários.

As Leis de Evolução de Lehman, portanto, se aplicam apenas aos sistemas de tipo-E. Elas estabelecem que a evolução de software desse tipo é inevitável, e não apenas resultado de uma equipe de desenvolvimento incapaz. Além disso, as leis mostram que existem limites em relação ao que as equipes de manutenção podem fazer. A autorregulação impede que trabalho demais ou insuficiente seja executado, sob pena de descontinuação do sistema.

14.1.1 LEI DA MUDANÇA CONTÍNUA

A primeira lei, de 1974, afirma que um sistema que é efetivamente usado deve ser continuamente melhorado, caso contrário torna-se cada vez menos útil, pois seu contexto de uso evolui. Se o programa não evoluir, terá cada vez menos valor até que se chegue à conclusão de que vale a pena substituí-lo por outro.

Essa lei expressa o fato conhecido de que programas suficientemente grandes nunca são terminados. Eles simplesmente continuam a evoluir.

Ela também expressa um princípio segundo o qual o software envelhece como se fosse um organismo vivo. Esse envelhecimento seria resultado das inconsistências entre o software e o ambiente no qual ele está inserido. No caso, o ambiente muda, mas o software não se adapta sozinho. Assim, um processo de evolução é necessário para evitar a obsolescência do software, prolongando sua vida útil. Caso tal evolução não seja feita, o software pode chegar a ser aposentado.

14.1.2 LEI DA COMPLEXIDADE CRESCENTE

A segunda lei, também de 1974, expressa que, à medida que um programa evolui, sua complexidade inerente aumenta, porque as correções feitas podem deteriorar sua organização interna. Isso só não acontece quando medidas específicas de cuidado são tomadas durante as atividades de evolução, como a refatoração do sistema quando necessário.

Segundo essa lei, à medida que mudanças são introduzidas no software, as interações entre elementos – nem sempre previstas ou planejadas na estrutura do software – fazem que a entropia interna aumente, ou seja, ocorre um crescimento cada vez mais desestruturado.

A cada nova mudança, a estrutura interna do software se torna menos organizada, aumentando gradativamente o custo de manutenções posteriores. De fato, chega-se a um ponto em que a refatoração do sistema torna-se obrigatória.

14.1.3 LEI FUNDAMENTAL DA EVOLUÇÃO DE PROGRAMAS: AUTORREGULAÇÃO

A terceira lei, também de 1974, estabelece que a evolução de programas está sujeita a uma dinâmica de autorregulação que faz que as medidas globais de esforço e outros atributos de processo sejam estatisticamente previsíveis (distribuição normal).

Essa lei reconhece que o desenvolvimento e a manutenção de um sistema ocorrem dentro de uma organização com objetivos que vão muito além do sistema. Então, os processos dessa organização acabam regulando a aplicação de esforço a cada um de seus sistemas.

Os pontos de controle e realimentação são exemplos de formas de regulação da organização. Quaisquer processos que fujam muito ao padrão da organização são logo refatorados para se adequar, de forma que o esforço gasto nas diferentes atividades permaneça distribuído de forma normal.

14.1.4 LEI DA CONSERVAÇÃO DA ESTABILIDADE ORGANIZACIONAL: TAXA DE TRABALHO INVARIANTE

A quarta lei, de 1980, expressa que a taxa média efetiva de trabalho global em um sistema em evolução é invariante no tempo, isto é, ela não aumenta nem diminui.

Essa lei parece ser a menos intuitiva de todas, pois se acredita que a carga de trabalho aplicada a um projeto depende apenas de decisões da gerência e que vá diminuindo com o tempo. Na prática, porém, as demandas de usuários também influenciam essas decisões, que se mantêm praticamente constantes no tempo.

14.1.5 LEI DA CONSERVAÇÃO DA FAMILIARIDADE: COMPLEXIDADE PERCEBIDA

A quinta lei, também de 1980, expressa que, durante a vida ativa de um programa, o conteúdo das sucessivas versões do programa (mudanças, adições e remoções) é estatisticamente invariante. Isso ocorre porque, para que um sistema evolua de forma saudável, todos os agentes relacionados a ele devem manter a familiaridade com suas características e funções. Se o sistema crescer demais, essa familiaridade será perdida, e leva tempo para recuperá-la.

Essa lei expressa também o fato de que a taxa de crescimento de um sistema é limitada pela capacidade dos indivíduos envolvidos em absorver as novidades coletiva e individualmente. Os dados observados sugerem que, se certo valor-limite de novidades for excedido, mudanças comportamentais ocorrerão de forma a baixar novamente a taxa de crescimento.

14.1.6 LEI DO CRESCIMENTO CONTÍNUO

A sexta lei, de 1980, estabelece que o conteúdo funcional de um sistema deve crescer continuamente para manter a satisfação do usuário.

Essa lei estabelece ainda que a mudança no software será sempre necessária, seja pela correção de erros (manutenção corretiva), aperfeiçoamento de funções existentes (manutenção perfectiva), seja pela adaptação a novos contextos (manutenção adaptativa).

14.1.7 LEI DA QUALIDADE DECRESCENTE

A sétima lei, de 1996, expressa que a qualidade de um sistema vai parecer diminuir com o tempo, a não ser que medidas rigorosas sejam tomadas para mantê-lo e adaptá-lo.

Essa lei estabelece que, mesmo que um software funcione perfeitamente por muitos anos, isso não significa que continuará sendo satisfatório. Conforme o tempo passa, os usuários ficam mais exigentes em relação ao software e, consequentemente, mais insatisfeitos com ele.

14.1.8 LEI DO SISTEMA REALIMENTADO

A oitava lei, de 1996, estabelece que a evolução de sistemas é um processo multinível, *multiloop* e multiagente de realimentação, devendo ser encarado dessa forma para que se obtenham melhorias significativas em uma base razoável.

Essa lei lembra que a evolução de software é um sistema retroalimentado e complexo que recebe *feedback* constante dos vários interessados. Em longo prazo, a taxa de evolução de um sistema acaba sendo determinada pelos retornos positivos e negativos de seus usuários, bem como pela quantidade de verba disponível, pelo número de usuários solicitando novas funções, interesses administrativos etc.

14.2 Classificação das Atividades de Manutenção

Segundo a Norma ISO/IEC 14764:2006¹, a manutenção ou a evolução de software deve ser classificada em quatro tipos:

- a) Corretiva: toda atividade de manutenção que visa corrigir erros ou defeitos do software.
- b) Adaptativa: toda atividade que visa adaptar as características do software a requisitos que mudaram, sejam novas funções, sejam questões tecnológicas.
- c) Perfectiva: toda atividade que visa melhorar o desempenho ou outras qualidades do software sem alterar necessariamente sua funcionalidade.
- d) Preventiva: toda atividade que visa melhorar as qualidades do software de forma que erros potenciais sejam descobertos e mais facilmente resolvidos.

14.2.1 MANUTENÇÃO CORRETIVA

A manutenção corretiva visa corrigir possíveis defeitos (que provocam erros) do software. Pode ser subdividida em dois subtipos:

- a) Manutenção para correção de erros conhecidos.
- b) Manutenção para detecção e correção de novos erros.

Os erros conhecidos de um software costumam ser registrados em um documento de considerações operacionais ou em notas de versão (Seção 10.3), de forma que os usuários do software possam contornar as falhas e evitar maiores transtornos.

Conforme visto no Capítulo 13, nem sempre o fato de se saber que um software tem um erro implica saber onde se localiza o defeito que provoca esse erro. Assim, a atividade de teste pode até ter descoberto que, em determinada situação, o software não se comporta de acordo com o esperado, mas a atividade de depuração pode não ter ainda encontrado o ponto no código-fonte causador desse erro. Nesse caso, enquanto a atividade de depuração ou mesmo de refatoração do software prossegue, a versão atual do software pode ter seus erros conhecidos registrados.

Novos erros podem, porém, ser detectados pelos usuários ao longo do uso do software. Quando relatados, tais erros devem ser incluídos no relatório de erros da versão do software e, dependendo de como a disciplina é

¹A norma pode ser adquirida no site: <www.iso.org/iso/catalogue_detail.htm?csnumber=39064>. Além disso, há uma preview oficial disponível em: <webstore.iec.ch/preview/info_14764%7Bed2.0%7Den.pdf>. Acessos em: 21 jan. 2013.

organizada, imediatamente encaminhados ao setor de manutenção para que o defeito seja identificado e corrigido. O processo deve ser rastreado de forma que se saiba sempre se o erro foi encaminhado, se já foi resolvido e se foi incorporado a uma nova versão do software. Além disso, deve-se saber se o usuário que relatou o erro recebeu a nova versão do software em um prazo razoável.

14.2.2 MANUTENÇÃO ADAPTATIVA

Conforme visto nas Leis de Lehman (Seção 14.1), a manutenção adaptativa é inevitável quando se trata de sistemas de software. Isso porque:

- a) Requisitos de cliente e usuário mudam com o passar do tempo.
- b) Novos requisitos surgem.
- c) Leis e normas mudam.
- d) Tecnologias novas entram em uso etc.

O sistema desenvolvido poderá estar ou não preparado para acomodar tais modificações de contexto. Na verdade, qualquer requisito inicialmente identificado é passível de mudança durante a operação do sistema. Cabe ao analista, na fase de levantamento de requisitos, identificar quais desses requisitos serão considerados permanentes e quais serão considerados transitórios (Wazlawick, 2011).

Com os requisitos permanentes acontece o seguinte:

- a) É mais barato e rápido incorporá-los ao software durante o desenvolvimento.
- b) É mais caro e demorado mudá-los depois que o software está em operação.

Com os requisitos transitórios acontece o inverso:

- a) É mais caro e demorado incorporá-los ao software durante o desenvolvimento.
- b) É mais barato e rápido mudá-los depois que o software está em operação.

Durante a análise de requisitos, cabe ao analista identificar quais requisitos serão considerados transitórios para serem adaptados por um simples processo de configuração. Os demais requisitos somente serão alterados a partir de um processo de manutenção, que pode envolver inclusive a refatoração do software.

Normalmente os requisitos não funcionais devem ser classificados como transitórios ou permanentes. Considere, por exemplo, que um sistema de venda de livros (Livraria Virtual) será produzido e um dos requisitos não funcionais (suplementar) é de que a moeda usada é o Real. Se o analista considerar esse requisito como transitório, deverá implementar no sistema a possibilidade de trabalhar com outras moedas, mesmo que em um primeiro momento apenas o Real seja cadastrado. Se novas moedas passarem a ser utilizadas no futuro, bastará cadastrá-las.

Mas esse processo tem um custo de desenvolvimento mais alto do que se o analista considerar o requisito como permanente. O fato de o requisito ser considerado *permanente* significa que o sistema não será preparado para a sua mudança. Assim, apenas uma moeda existirá para o sistema (o Real), e não será possível cadastrar novas moedas. Se no futuro surgir a necessidade de cadastrar uma nova moeda, será necessário aplicar manutenção adaptativa ao sistema, refatorando algumas de suas partes para passar a trabalhar com essa nova moeda. Assim, a economia que se obteve ao longo do projeto será gasta durante a operação do sistema.

Dessa forma, pode-se considerar que o analista deve ponderar, antes de decidir se um requisito é transitório ou permanente, qual é a real probabilidade de que ele mude durante a operação do sistema, qual é o impacto de se prevenir para essa mudança ao longo do projeto e qual é o impacto de mudar o sistema durante sua operação. Ponderando esses três valores, o analista deverá tomar uma decisão. De um lado da balança estará o custo de implementar a flexibilidade do requisito no projeto e do outro lado da balança o custo de acomodar a mudança do requisito depois de o sistema estar em operação combinado com a probabilidade de o requisito realmente mudar.

Em geral, na falta de bons motivos para o contrário, a maioria dos requisitos é deixada como permanente.

14.2.3 MANUTENÇÃO PERFECTIVA

A manutenção perfectiva consiste em mudanças que afetam mais as características de desempenho do que as características de funcionalidade do software. Em geral, tais melhorias são buscadas em função da pressão de

mercado, visto que se preferem produtos mais eficientes ou com melhor usabilidade e com mesma funcionalidade a produtos menos eficientes, especialmente em áreas nas quais o processamento é crítico, como jogos e sistemas de controle em tempo real.

A melhoria de características vai estar quase sempre ligada às qualidades externas do software (Capítulo 11), mas especialmente às qualidades ligadas a funcionalidade, confiabilidade, usabilidade e eficiência.

14.2.4 MANUTENÇÃO PREVENTIVA

A manutenção preventiva pode ser realizada através de atividades de reengenharia, nas quais o software é modificado para resolver problemas potenciais.

Um sistema que suporta até 50 acessos simultâneos e que já conta com picos de 20 a 30 acessos, por exemplo, pode sofrer um processo de manutenção preventiva através de reengenharia ou refatoração de sua arquitetura, de forma que passe a suportar 500 acessos, afastando a possibilidade de colapso por um período de tempo razoável.

Outro uso da manutenção preventiva consiste em aplicar técnicas de engenharia reversa como refatoração ou redocumentação para melhorar a manutenção do software.

14.3 Processo de Manutenção

A manutenção pode ser vista como um processo a ser executado. A não ser no caso da manutenção preventiva, esse processo costuma ser iniciado a partir de uma requisição do cliente ou usuário do software.

No caso da manutenção corretiva, pode ser interessante que o usuário apresente uma descrição clara do erro, que pode ser produzida a partir da captura de telas.

No caso da manutenção perfectiva, o usuário deve descrever da melhor forma possível os novos requisitos que deseja. Para efetuar essa atividade, ele pode precisar do apoio de um analista capacitado em levantamento de requisitos.

Sugere-se que as tarefas de manutenção identificadas sejam priorizadas e colocadas em uma fila de prioridades para serem resolvidas de acordo com a urgência. Nessa fila, as solicitações de manutenção perfectiva costumam ocupar os últimos lugares. No caso da manutenção corretiva e adaptativa, o impacto do erro ou inadequação sobre os clientes deve ser o primeiro critério de desempate, devendo ser tratados prioritariamente os problemas de maior impacto.

Recomenda-se também que, uma vez selecionada uma tarefa de manutenção, as seguintes ações sejam efetuadas:

- a) Análise de esforço para a tarefa de manutenção (Seção 14.6).
- b) Análise de risco para a tarefa de manutenção (verificando possíveis riscos, sua probabilidade e impacto, elaborando e executando possíveis planos de mitigação – Capítulo 8).
- c) Planejamento da tarefa de manutenção (estabelecendo prazos, responsáveis, recursos e entregas – Capítulo 6).
- d) Execução da tarefa de manutenção.

Ao final da tarefa de manutenção devem ser executados os testes de regressão (Seção 13.2.6).

A Norma IEEE 1219-98² define que o processo de manutenção de software deve ser composto por oito atividades:

- a) *Classificação e identificação da requisição de mudança*: essa atividade vai avaliar, entre outras coisas, se a manutenção necessária será corretiva, adaptativa ou perfectiva e, em função de sua urgência, receberá um lugar na fila de prioridades das atividades de manutenção. Opcionalmente, também a solicitação de modificação poderá ser rejeitada se for impossível ou indesejável implementá-la.
- b) *Análise*: abrange as atividades tradicionais de análise, com a identificação ou modificação de requisitos, modelo conceitual, casos de uso e outros artefatos, conforme a necessidade.
- c) *Design*: abrange as atividades usuais de *design*, com a definição da tecnologia e das camadas de interface, persistência, comunicação etc.

²A partir de junho de 2010, essa norma foi revisada e substituída por outra, a P14764 (*draft*). Disponível em: <ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=11168>. Acesso em: 21 jan. 2013.

- d) *Implementação*: nessa atividade é gerado novo código, que atende à modificação solicitada, e são feitos os testes de unidade e integração.
- e) *Teste de sistema*: nessa atividade são feitos os testes finais das novas características do sistema do ponto de vista do usuário.
- f) *Teste de aceitação*: nessa atividade, o cliente é envolvido para aprovar ou não as modificações feitas.
- g) *Entrega*: nessa atividade o produto é entregue ao cliente para nova instalação.

O processo de manutenção é, portanto, bastante semelhante ao de desenvolvimento de software. Existem algumas diferenças específicas, especialmente no que se refere ao tratamento com o cliente e também ao fato de que o processo de manutenção costuma durar muitos anos, ao passo que em geral o processo de desenvolvimento dura meses ou poucos anos.

14.4 Ferramentas para Manutenção de Software

Um dos pontos-chave referentes ao processo de manutenção é que se tenha uma boa disciplina de gerenciamento de configuração e mudança (Capítulo 10), sem a qual se poderá perder o controle sobre atividades de manutenção realizadas e disponibilizadas aos clientes. Essa atividade deverá ser, então, suportada por ferramentas adequadas.

Existem várias ferramentas disponíveis que auxiliam o processo de manutenção ou evolução de software. Um exemplo é o Bugzilla³, sistema de rastreamento de defeitos que permite que indivíduos ou equipes mantenham controle efetivo sobre defeitos encontrados e tratados em seus sistemas.

Pelo fato de ser gratuita e ter várias funcionalidades interessantes, essa ferramenta tem se tornado uma das mais populares. A ferramenta é baseada em *Web* e *e-mail*, servindo para o controle e o acompanhamento de defeitos, bem como para pedidos de alterações e correções. É utilizada por grandes corporações e projetos, como a NASA, a NBC e a Wikipédia⁴.

14.5 Tipos de Atividades de Manutenção e suas Métricas

Vários autores, dentre eles Boehm (1981), apontam que as atividades de manutenção, longe de serem um mero detalhe, são aquelas em que as empresas colocam mais esforço.

A análise de centenas de projetos em longo prazo mostrou que foram colocados mais tempo e mais esforço nas atividades de manutenção do que nas atividades de desenvolvimento de software. Assim, é necessário que se tenha também modelos de estimativa de esforço e custo para atividades de manutenção.

Jones (2002) apresenta um estudo em que se observa que, em média, cada funcionário remove 8 defeitos de um sistema em operação por mês. Porém, as atividades de manutenção variam muito e podem ser classificadas em vários tipos (Jones T. C., 1998), os quais serão discutidos nas próximas subseções.

14.5.1 REPARAÇÃO DE DEFEITOS

A reparação de defeitos é, possivelmente, a atividade mais importante e urgente em manutenção de software, porque se destina a eliminar problemas que não deveriam existir. O custo dessas atividades normalmente é absorvido pela empresa desenvolvedora, a não ser que cláusulas contratuais específicas estabeleçam outro tipo de entendimento.

Dificilmente essas atividades podem ser estimadas com pontos de função, porque a maioria das atividades de reparação de defeitos implica escrever algumas poucas linhas de código, o que equivaleria a uma fração de um ponto de função. Mas, de outro lado, algumas dessas atividades podem ser altamente consumidoras de tempo.

Uma métrica mais comum para esse tipo de atividade, uma vez que os custos são arcados pela organização desenvolvedora, é o número de defeitos que a organização consegue reparar em um mês. Um valor aceitável, de acordo com normas norte-americanas, é de 8 defeitos reparados por mês. Porém, empresas com bons processos e práticas conseguem reparar até 20 defeitos por mês em seus sistemas.

³Disponível em: <www.bugzilla.org/about/>. Acesso em: 21 jan. 2013.

⁴Disponível em: <pt.wikipedia.org/wiki/Bugzilla>. Acesso em: 21 jan. 2013.

TABELA 14.1 Tempo de resposta ao erro em função de sua gravidade⁵

| Gravidade | Significado | Tempo nominal da descoberta aos reparos iniciais | Percentual em relação aos defeitos relatados |
|-----------|--|--|--|
| 1 | Aplicação não funciona | 1 dia | 1% |
| 2 | Funcionalidade principal não funciona | 2 dias | 12% |
| 3 | Funcionalidade secundária não funciona | 30 dias | 52% |
| 4 | Erro cosmético | 120 dias | 35% |

Jones (1998) ainda indica que, conforme a gravidade do erro, diferentes tempos de espera são toleráveis (Tabela 14.1).

Os fatores a seguir ainda podem influenciar a estimativa de esforço a ser aplicada às atividades de reparação de defeitos:

- a) *Defeitos suspensos (abeyant)*: em cerca de 10% das vezes, a falha relatada pelo cliente não é reproduzida no ambiente de manutenção. É a típica situação “Na minha máquina funciona”. Esse tipo de defeito deve-se a combinações de condições (versão do sistema operacional, outros produtos instalados na mesma máquina etc.) que, muitas vezes, são difíceis de detectar e reproduzir, portanto é o tipo mais caro de manutenção corretiva. Esses defeitos ficam suspensos até que se consiga repeti-los.
- b) *Defeitos inválidos*: cerca de 15% dos defeitos relatados por usuários não são propriamente defeitos no software, mas produto de erros gerados pelos próprios usuários ou por sistemas relacionados. Mesmo assim, esses problemas devem ser catalogados e processados, e sua análise demanda tempo e esforço da empresa que faz a manutenção do sistema.
- c) *Consertos ruins (bad fix injection)*: cerca de 7% das atividades de correção de erros acabam introduzindo novos erros no software. Essa porcentagem pode variar de 1 a 20%, dependendo do nível de qualidade do processo de manutenção e da seriedade com que os testes de regressão são feitos.
- d) *Defeitos duplicados*: em sistemas com muitos usuários é comum que um mesmo defeito seja relatado por mais de um usuário. Assim, embora o defeito só precise ser resolvido uma vez, o fato de ele ser relatado por vários usuários faz que seja necessário investir tempo nisso. Grandes empresas de software chegam a ter 10% de seus relatos de defeitos classificados como defeitos duplicados.

14.5.2 REMOÇÃO DE MÓDULOS SUJEITOS A ERROS

Uma pesquisa realizada pela IBM nos anos 1960 demonstrou que os defeitos não se distribuem aleatoriamente ao longo de uma aplicação. Muito pelo contrário: eles tendem a se concentrar em determinados módulos da aplicação. Foi observado que, em um grande sistema da empresa, com 425 módulos, 300 módulos nunca foram alvo de manutenção corretiva, enquanto outros 31 módulos concentraram cerca de 2.000 relatos de erros ao longo de um ano, correspondendo a mais de 60% do total de erros relatados para o produto inteiro (Mukhija, 2003).

Módulos sujeitos a defeitos podem nunca estabilizar, porque a taxa de consertos ruins pode passar de 100%, ou seja, a cada defeito consertado, novos defeitos podem acabar sendo introduzidos⁶.

14.5.3 SUPORTE A USUÁRIOS

O suporte a usuários fará a interface entre o cliente do software e a empresa que presta manutenção ao software. O suporte a usuários costuma receber as reclamações, fazer uma triagem delas, encaminhar uma solução previamente conhecida ao cliente ou o problema ao setor de manutenção.

⁵Jones (1998).

⁶A internet apresenta uma brincadeira relacionada a esse fato, que é incorporada à “técnica” XGH, ou *eXtreme Go Horse*. Uma das regras dessa técnica estabelece que, a cada defeito consertado com XGH 7, novos defeitos sejam criados, o que faz a quantidade de defeitos tender ao infinito. Disponível em: <gohorseprocess.wordpress.com>. Acesso em: 21 jan. 2013.

O tamanho da equipe de suporte dependerá de vários fatores, entre os quais os mais importantes são a quantidade esperada de defeitos e a quantidade de clientes.

Estima-se que, para um software típico (que não apresenta grandes problemas de qualidade logo de partida), um atendente consiga tratar as chamadas de cerca de 150 clientes por mês, caso o meio de contato seja o telefone. De outro lado, se o meio de contato for *e-mail* ou *chat*, esse número pode subir para 1.000 usuários por atendente por mês.

14.5.4 MIGRAÇÃO ENTRE PLATAFORMAS

A migração de um produto para outra plataforma, quando se trata de software personalizado, é feita por demanda do cliente. Quando se trata de um software de prateleira, ela é feita com a intenção de aumentar o mercado.

Normalmente, migrações são projetos por si só, embora possam ser consideradas atividades de evolução de software. Assume-se que sistemas desenvolvidos de acordo com boas práticas e com boa documentação possam ser migrados a uma taxa de 50 pontos de função por desenvolvedor-mês. Porém, se os sistemas forem mal documentados e tiverem organização obscura, essa taxa pode baixar para até 5 pontos de função por desenvolvedor-mês.

14.5.5 CONVERSÃO DE ARQUITETURA

Geralmente, uma conversão de arquitetura de sistema é feita por pressão tecnológica. É o caso, por exemplo, de mudar de arquivos simples para bancos de dados relacionais, ou de mudar uma interface orientada a linha de comando para uma interface gráfica.

No caso da migração entre plataformas, se o software for personalizado, a conversão possivelmente será uma demanda do cliente, enquanto no caso de software de prateleira será uma estratégia para buscar novos mercados.

A conversão de arquitetura também pode ser uma estratégia para melhorar a manutenção de um sistema, em que se pode transformar um sistema monolítico ou feito com blocos *ad hoc* em um sistema bem estruturado com componentes e classes coesas.

A produtividade de um projeto de conversão de arquitetura dependerá basicamente da qualidade das especificações do sistema. Quanto mais obscuras forem as especificações, mais difícil será a conversão.

Em geral, sistemas mal documentados ou obscuros precisam passar por processos de engenharia reversa antes de serem convertidos para uma nova arquitetura.

14.5.6 ADAPTAÇÕES OBRIGATÓRIAS

Talvez os piores tipos de manutenção de software sejam as adaptações obrigatórias, decorrentes de mudanças em normas e leis, formas de cálculo de impostos etc. O problema é que essas mudanças não podem ser previstas pela equipe de desenvolvimento ou manutenção, nem mesmo pelo cliente. Além disso, normalmente têm um prazo curto e estreito para serem aplicadas, e as penalidades por não adaptação costumam ser altas.

14.5.7 OTIMIZAÇÃO DE PERFORMANCE

Atividades de otimização de performance implicam analisar e resolver gargalos da aplicação, geralmente relacionados com o acesso a dados, processamento e número de usuários simultâneos.

Essas atividades variam muito em relação ao tipo e à carga de trabalho, por isso é muito difícil estabelecer um padrão para a estimativa de custos. Uma técnica que pode ser empregada em alguns casos é a otimização estilo *anytime*, usando *timeboxing* (Seção 4.2.2), ou seja, faz-se a melhor otimização possível dentro do tempo e com os recursos previamente destinados a essa atividade.

14.5.8 MELHORIAS

As melhorias são um tipo de manutenção adaptativa e perfectiva normalmente iniciado por solicitação dos clientes, que em geral acabam arcando com os custos relacionados.

Muitas vezes, elas implicam a introdução de novas funcionalidades, de forma que possam ser aplicadas técnicas usuais de estimativa de esforço por CII, pontos de função ou pontos de caso de uso.

Pode-se considerar que existem dois tipos de melhoria:

- a) *Pequenas melhorias*: consistem de aproximadamente 5 pontos de função, ou seja, da introdução de um novo relatório, consulta ou tela.
- b) *Grandes melhorias*: consistem de um número significativamente maior de pontos de função, em geral com mais de 20 pontos de função, e devem ser tratadas como pequenos projetos de desenvolvimento.

Um dos aspectos que possivelmente diferenciam a estimativa de esforço das melhorias em relação ao desenvolvimento do novo software é que no caso das melhorias deve-se levar em conta o estado atual do sistema. Se o sistema for bem organizado e documentado será mais fácil integrar as novas funcionalidades; caso contrário, haverá um esforço de integração maior, que deverá ser levado em conta como fator técnico no momento de aplicar a estimativa.

Estima-se que sistemas em operação, em média, aumentam seus pontos de função em cerca de 7% anualmente em função de melhorias (Jones T. C., 1998).

14.6 Modelos de Estimação de Esforço de Manutenção

Modelos de estimação de esforço para atividades de manutenção foram propostos na literatura, alguns dos quais serão apresentados nas subseções seguintes. Assim como os modelos de estimação de esforço de desenvolvimento, esses modelos paramétricos podem usar como base para a estimação tanto o número de pontos de função quanto o número de linhas de código estimadas.

14.6.1 MODELO ACT

O modelo *ACT* baseia-se em uma estimativa da porcentagem de linhas de código que vão sofrer manutenção. São consideradas linhas em manutenção tanto as linhas de código novas criadas quanto as linhas alteradas durante a manutenção. O valor da variável *ACT* é, portanto, o número de linhas que sofrem manutenção dividido pelo número total de linhas do código em um ano típico.

Boehm (1981) estabeleceu a seguinte equação para calcular o esforço estimado de manutenção durante um ano:

$$E = ACT * SDT$$

em que *E* é o esforço, medido em desenvolvedor-mês, a ser aplicado no período de um ano nas atividades de manutenção, *ACT* é a porcentagem esperada de linhas modificadas ou adicionadas durante um ano em relação ao tamanho do software, e *SDT* é o tempo de desenvolvimento do software (*Software Development Time*).

Um software que foi desenvolvido com um esforço de 80 desenvolvedores-mês, por exemplo, terá *SDT*=80. Se a taxa anual esperada de linhas em manutenção (*ACT*) for de 2%, então o esforço anual esperado de manutenção para esse software será dado por:

$$E = 0,02 * 80 = 1,6$$

Assim, para esse sistema hipotético, de acordo com essa fórmula, espera-se um esforço anual de 1,6 desenvolvedor-mês em atividades de manutenção.

Schaefer (1985) apresenta a seguinte variação para essa fórmula:

$$E = ACT * 2,4 * KSLOC^{1,05}$$

ou seja, Schaefer substitui o tempo de desenvolvimento do produto (caso ele não seja conhecido) por uma fórmula baseada no número total de milhares de linhas de código do produto (KSLOC). Assim, um software com 20 mil linhas de código e *ACT* de 2% teria o seguinte esforço anual de manutenção (em desenvolvedor-mês):

$$E = 0,02 * 2,4 * 20^{1,05} = 1,115$$

Esse modelo tem as mesmas desvantagens do modelo COCOMO 81, ou seja, não é realisticamente aplicável a sistemas novos quando não existem dados históricos para *ACT*, a quantidade de linhas de código modificadas não necessariamente indica esforço de manutenção e, sobretudo, a abordagem não usa nenhum atributo das atividades de manutenção como base para calcular o esforço. Porém, na falta de outras informações, é um método simples de aplicar.

14.6.2 MODELO DE MANUTENÇÃO DE CII

Como já foi visto na Seção 7.3, CII é um método de estimativa de esforço de desenvolvimento que toma como entrada o número de linhas de código a serem desenvolvidas ou pontos de função convertidos em KSLLOC.

A equação de estimativa de esforço para manutenção é semelhante à equação usada no modelo *post-architecture*, com a seguinte forma:

$$E = A * KSLLOC_m^s * \Pi_{i=1}^n = M_i$$

Em que:

- E é o esforço de manutenção em desenvolvedor-mês a ser calculado.
- A é uma constante calibrada pelo método, inicialmente valendo 2,94.
- $KSLLOC_m$ é o número de linhas de código que se espera adicionar ou alterar ajustado pelo fator de manutenção (ver a seguir). Não são contadas as linhas que eventualmente serão excluídas do código.
- S é o coeficiente de esforço determinado pelos fatores de escala e calculado como mostrado na Seção 7.3.
- M_i são os multiplicadores de esforço.

As seguintes mudanças em relação ao modelo *post-architecture* são implementadas para o cálculo do esforço de manutenção:

- O multiplicador de esforço SCED (Cronograma de Desenvolvimento Requerido) não é usado (ou assumido como nominal), porque se espera que os ciclos de manutenção tenham duração fixa predetermineda.
- O multiplicador de esforço RUSE (Desenvolvimento para Reúso) não é usado (ou assumido como nominal), porque se considera que o esforço requerido para manter a reusabilidade de um componente de software é balanceado pela redução do esforço de manutenção por causa do projeto, da documentação e dos testes cuidadosos do componente.
- O multiplicador de esforço RELY (Software com Confiabilidade Requerida) tem uma tabela de aplicação diferenciada. Assume-se que RELY na fase de manutenção vai depender do valor que RELY tinha na fase de desenvolvimento. Se o produto foi desenvolvido com baixa confiabilidade, haverá maior esforço para consertá-lo. Se o produto foi desenvolvido com alta confiabilidade, haverá menor esforço para consertá-lo (Tabela 14.2), exceto no caso de sistemas com risco à vida humana, nos quais a necessidade de confiabilidade, mesmo na fase de manutenção, faz crescer o esforço.

Assim, a tabela dada deve ser aplicada da seguinte forma: avalia-se a confiabilidade com a qual o sistema foi originalmente desenvolvido (primeira linha) e encontra-se o valor numérico a ser aplicado na fase de manutenção

TABELA 14.2 Forma de obtenção do equivalente numérico para RELY na fase de manutenção

| Descriptor | Pequena inconveniência | Perdas pequenas facilmente recuperáveis | Perdas moderadas facilmente recuperáveis | Alta perda financeira | Risco à vida humana | |
|-----------------------------|------------------------|---|--|-----------------------|---------------------|------------|
| Avaliação | Muito baixo | Baixo | Nominal | Alto | Muito alto | Extra-alto |
| Equivalente numérico | 1,23 | 1,10 | 1,00 | 0,99 | 1,07 | n/a |

na terceira linha. Note que os valores são inversamente proporcionais aos usados na fase de desenvolvimento (Tabela 7.12).

O número de KSLOC usado na fase de manutenção deve ser ajustado antes de ser aplicado na equação de cálculo de esforço pelo uso do *fator de ajuste de manutenção (MAF)*:

$$KSLOC_m = (KSLOC_{\text{adicionadas}} + KSLOC_{\text{modificadas}}) * MAF$$

O fator de ajuste de manutenção *MAF* é calculado a partir da equação a seguir:

$$MAF = 1 + \left(\frac{SU}{100} * UNFM \right)$$

Em que:

- *SU* é o fator de ajuste relacionado à *compreensão do software (software understanding)*, calculado de acordo com a Tabela 14.3.
- *UNFM* é o fator de *não familiaridade* com relação ao software, calculado de acordo com a Tabela 14.4.

Caso não se conheça, *a priori*, o número de linhas a serem adicionadas ou modificadas, esse valor pode ser obtido a partir de uma análise histórica dos processos de manutenção do projeto ou da empresa. A partir de uma estimativa da porcentagem de linhas adicionadas ou alteradas, pode-se prever a quantidade de manutenção que será necessária no futuro.

TABELA 14.3 Forma de cálculo do equivalente numérico para SU

| | Muito baixo | Baixo | Nominal | Alto | Muito alto |
|-----------------------------|---|---|---|--|--|
| Estrutura | Coesão muito baixa, acoplamento alto, código espaguete | Coesão moderadamente baixa, acoplamento alto | Razoavelmente bem estruturado, algumas áreas fracas | Alta coesão, baixo acoplamento | Modularidade forte, ocultamento de informação em estruturas de dados ou controle (objetos) |
| Clareza da aplicação | As visões do programa e sua aplicação no mundo real não batem | Alguma correlação entre o programa e a aplicação | Correlação moderada entre o programa e a aplicação | Boa correlação entre o programa e a aplicação | As visões do programa e da aplicação no mundo real claramente batem |
| Autodescrição | Código obscuro, documentação faltando, obscura ou obsoleta | Alguns comentários no código e cabeçalhos, alguma documentação útil | Nível moderado de documentação no código e cabeçalhos | Código e cabeçalhos bem documentados, algumas áreas fracas | Código autodescritivo, documentação atualizada e bem organizada baseada em design |
| Valor de SU | 50 | 40 | 30 | 20 | 10 |

TABELA 14.4 Forma de cálculo do equivalente numérico para UNFM

| Nível de não familiaridade | Valor de UNFM |
|----------------------------|---------------|
| Completamente familiar | 0,0 |
| Basicamente familiar | 0,2 |
| Um tanto familiar | 0,4 |
| Um tanto não familiar | 0,6 |
| Basicamente não familiar | 0,8 |
| Completamente não familiar | 1,0 |

14.6.3 MODELOS FP E SMPEEM

O modelo *FP* (Albrecht, 1979) para cálculo de esforço de manutenção é baseado unicamente em pontos de função, e não em linhas de código. Segundo esse modelo, é necessário calcular os pontos de função não ajustados de quatro tipos de funções:

- a) *ADD*: UFP de funções que vão ser adicionadas.
- b) *CHG*: UFP de funções que vão ser alteradas.
- c) *DEL*: UFP de funções que vão ser removidas.
- d) *CFP*: UFP de funções que serão adicionadas por conversão.

Além de classificar as entradas, saídas, consultas, arquivos internos e arquivos externos nesses quatro tipos antes de contabilizar seus pontos de função não ajustados, a técnica propõe que os fatores de ajuste técnico (*VAF*) (Seção 7.4.3) sejam calculados para dois momentos: antes da manutenção (VAF_A) e depois da manutenção (VAF_D). A equação seguinte é, então, aplicada:

$$E = (ADD + CHG + CFP) * VAF_D + DEL * VAF_A$$

Ahn, Suh, Kim e Kim (2003)⁷ apresentam uma evolução desse modelo, chamada *SMPEEM* (*Software Maintenance Project Effort Estimation Model*)⁸, na qual incluem mais 10 fatores de ajuste específicos para as atividades de manutenção:

- a) Conhecimento do domínio da aplicação.
- b) Familiaridade com a linguagem de programação.
- c) Experiência com o software básico (sistema operacional, gerenciador de banco de dados).
- d) Estruturação dos módulos de software.
- e) Independência entre os módulos de software.
- f) Legibilidade e modificabilidade da linguagem de programação.
- g) Reusabilidade de módulos de software legados.
- h) Atualização da documentação.
- i) Conformidade com padrões de engenharia de software.
- j) Testabilidade.

Os três primeiros fatores são referentes às habilidades de engenharia da equipe. Os quatro fatores seguintes são referentes a características técnicas, e os três últimos fatores estão relacionados ao ambiente de manutenção.

14.7 Engenharia Reversa e Reengenharia

Em algumas situações, o processo de manutenção ou evolução de um sistema exige uma atividade mais drástica do que simplesmente consertar partes do código. Sistemas antiquados, mal documentados e mal mantidos poderão requerer um processo completo de reengenharia para que possam voltar a evoluir de modo mais saudável.

A reengenharia de um sistema é, basicamente, o processo de descobrir como um sistema funciona para que se possa refatorá-lo ou mesmo criar um novo sistema tecnologicamente atualizado que cumpra suas tarefas.

Chikofsky e Cross II (1990) apresentam uma taxonomia de termos relacionados à reengenharia de software:

- a) *Engenharia direta (forward engineering)*: processo tradicional de produção de software que vai das abstrações de mais alto nível até o código executável.
- b) *Engenharia reversa (reverse engineering)*: processo de analisar um sistema ou seus modelos de forma a conseguir produzir especificações de nível mais alto. É um processo de exame e explicação.
- c) *Redocumentação (redocumentation)*: é uma subárea da engenharia reversa. Em geral, trata-se de obter formas alternativas de uma especificação no mesmo nível do artefato examinado.

⁷Disponível em: <https://files_ifi_uzh_ch_rerg_arvo_ftp_kvse_ahn03.pdf>. Acesso em: 21 jan. 2013.

⁸Modelo de Estimação de Esforço para Projeto de Manutenção de Software.

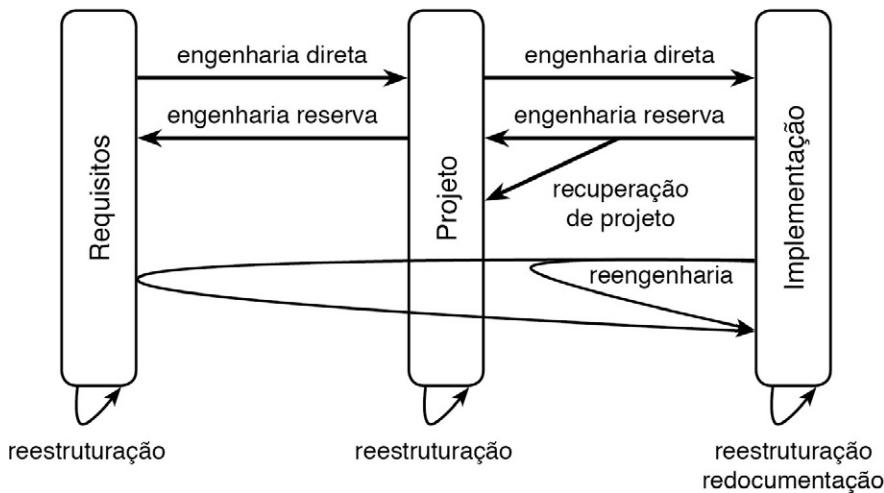


Figura 14.1 Relação esquemática entre os diferentes termos relacionados à engenharia reversa⁹.

- d) *Recuperação de projeto (design recovery)*: é outra subárea da engenharia reversa. Ao contrário da anterior, a recuperação de projeto vai realizar abstrações a partir dos elementos examinados, a fim de produzir artefatos em níveis mais altos do que os examinados.
- e) *Reestruturação (restructuring)*: é uma das formas de refatoração e consiste em transformar um artefato internamente, porém mantendo sua funcionalidade aparente. Normalmente, a reestruturação é realizada para simplificar a arquitetura de sistemas de forma a minimizar futuros problemas de manutenção.
- f) *Reengenharia (reengineering)*: exame e alteração de um sistema para reconstruí-lo de forma diferente. Geralmente, inclui alguma forma de engenharia reversa seguida de engenharia direta ou reestruturação.

A Figura 14.1 apresenta, esquematicamente, a relação entre os diferentes termos.

Segundo os autores, a expressão “engenharia reversa” teve origem na análise de hardware, em que a prática de decifrar projetos a partir de produtos prontos é usual. Porém, na área de hardware, o objetivo da engenharia reversa normalmente é obter informações para duplicar um produto. Já na área de software os objetivos podem ser outros:

- a) Lidar com a complexidade do código.
- b) Gerar visões alternativas, como diagramas.
- c) Recuperar informações perdidas sobre modificações não documentadas.
- d) Detectar efeitos colaterais que não foram previstos no processo de engenharia direta.
- e) Sintetizar conceitos e estrutura em abstrações de nível mais alto.
- f) Facilitar o reúso de ativos de software existentes (por exemplo, para aproveitá-los em linhas de produto de software).

Uma das diretrizes da engenharia reversa consiste em, por causa de seu alto custo, aplicá-la apenas onde for realmente necessária, não necessariamente ao sistema todo (Bennett, 2000).

Em função do seu objeto, a engenharia reversa pode ser classificada em dois tipos: *engenharia reversa de código* e *engenharia reversa de dados*. As subseções a seguir apresentam esses dois tipos.

14.7.1 ENGENHARIA REVERSA DE CÓDIGO

Segundo Muller et al. (2000), a engenharia reversa de software tem se concentrado no *código*, enquanto a engenharia direta atua mais amplamente nos diferentes *níveis de abstração* de uma especificação de sistema. Isso pode ser explicado pelo fato de que, em geral, a questão de engenharia reversa se coloca quando existe um sistema legado e é necessário, entre outras coisas, descobrir regras de negócio escondidas nele.

⁹Chikofsy e Cross II (1990).

O problema é que muitas vezes o código, por si só, não contém todas as informações de que se necessita. Muitas decisões tomadas ficam registradas apenas na memória dos desenvolvedores. Com o tempo, essas memórias podem ser perdidas ou as pessoas podem ficar inacessíveis. Além disso, segundo as Leis de Lehman (Seção 14.1), a complexidade do software tende a crescer com o tempo. Assim, fica cada vez mais difícil lidar com o código.

É possível caracterizar dois tipos de engenharia reversa de código:

- a) A partir de código-fonte.
- b) A partir de código-objeto.

O primeiro tipo é usado para melhorar a compreensão sobre sistemas legados ou mal documentados. Já o segundo tipo pode ser usado, entre outras coisas, para a clonagem de produtos. Nesse caso, pode-se usar a técnica de *clean room design*¹⁰ (Schwartz, 2001) para evitar a quebra de direitos autorais. Essa técnica consiste em usar a engenharia reversa apenas para entender o produto original e, então, desenvolver um produto novo que não use quaisquer partes do produto original. Deve-se tomar cuidado, porém, porque no caso de patentes essa técnica não pode ser aplicada, já que os projetos patenteados são protegidos por leis.

A engenharia reversa de código pode ser realizada a partir de uma ou mais entre as técnicas a seguir:

- a) *Análise de fluxo de dados*: consiste em verificar o comportamento do sistema como uma caixa-preta, ou seja, sem ter conhecimento de sua estrutura interna. A análise do comportamento do sistema pode permitir, então, que um novo sistema seja desenvolvido para ter o mesmo comportamento.
- b) *Dessassemblagem*: consiste em usar um desassemblador, que converte o código executável em mnemônicos de linguagem *Assembly*.
- c) *Descompilação*: consiste em usar um descompilador para obter uma aproximação do código original usado para produzir o executável. Os resultados podem variar bastante, pois há questões difíceis de tratar, como a escolha de nomes para variáveis e procedimentos.

A análise de fluxo de dados pode ser feita com ferramentas como *bus analyzers* ou *packet sniffers*. Um *bus analyzer* é uma combinação de software e hardware usada para monitorar e apresentar, em formato adequado, o fluxo de informações que passa por um barramento de dados escolhido. Um exemplo de ferramenta é JTAG (*Joint Test Action Group*), baseada no padrão IEEE 1149.1¹¹.

Já os *packet sniffers* usualmente são conectados a redes de computadores (com ou sem fio). Uma comparação entre vários *packet sniffers* pode ser encontrada na Wikipédia¹².

Um desassemblador bastante popular é o IDA¹³, disponível para vários sistemas operacionais e processadores. A empresa fabricante também disponibiliza um descompilador para programas compilados em C ou C++. Embora a versão corrente seja paga, versões anteriores do software são *freeware*. Outros exemplos de desassembladores são: PVDasm¹⁴, OllyDbg¹⁵ e Ildasm¹⁶.

Um tipo especial de desassemblador é o *debugger*, que permite que o código seja executado e alterações de partes do código sejam feitas interativamente.

Os descompiladores são bem mais complexos do que os desassembladores, pois precisam gerar comandos em nível bem mais alto do que estes últimos, que fazem apenas uma tradução direta dos códigos de máquina em mnemônicos. Os descompiladores costumam trabalhar em fases caracterizadas por diferentes componentes:

- a) *Carregador*: esse componente faz o carregamento do programa e identifica algumas informações básicas, como o tipo de processador para o qual o código foi gerado e o ponto de entrada. Pode chegar até a encontrar o equivalente ao módulo principal de um programa, a partir do qual as inicializações e chamadas são feitas.
- b) *Desassemblador*: esse componente procura transformar os códigos de máquina carregados por uma representação mnemônica independente de processador.

¹⁰Não confundir com o método *cleanroom* (Seção 11.4.3).

¹¹Disponível em: <www.jtag.com/>. Acesso em: 21 jan. 2013.

¹²Disponível em: <en.wikipedia.org/wiki/Comparison_of_packet_analyzers>. Acesso em: 21 jan. 2013.

¹³Disponível em: <www.hex-rays.com/idapro/>. Acesso em: 21 jan. 2013.

¹⁴Disponível em: <pvdasm.reverse-engineering.net/>. Acesso em: 21 jan. 2013.

¹⁵Disponível em: <www.ollydbg.de/>. Acesso em: 21 jan. 2013.

¹⁶Disponível em: <msdn.microsoft.com/en-us/library/f7dy01k1%28VS.80%29.aspx>. Acesso em: 21 jan. 2013.

- c) *Identificador de expressões idiomáticas*: alguns processadores usam instruções muito específicas para realizar operações que seriam bem mais simples em uma linguagem independente de tecnologia. Por exemplo, a instrução “`xor eax, eax`” é usada para atribuir zero ao registrador `eax`. Ela poderia ser descrita mais claramente como “`eax := 0`”. Expressões idiomáticas são catalogadas para cada processador.
- d) *Análise de programa*: o analisador de programa vai identificar sequências de operações e tentar agrupá-las em comandos. Por exemplo, uma expressão que, em linguagem de alto nível, seria escrita como “`x := y + 45 * (z - x) / 2`” seria compilada como uma sequência de operações elementares de adição, multiplicação, subtração e divisão. O analisador de programa deve ser capaz de identificar essa sequência e transformá-la na expressão que possivelmente era a original do programa-fonte.
- e) *Análise de fluxo de dados*: consiste em detectar as variáveis e seu escopo no programa. O problema é complexo, porque a mesma posição de memória pode ser ocupada por mais de uma variável em momentos diferentes, e uma variável pode ocupar mais de uma posição da memória. A abordagem geral para tratar esse problema é baseada em grafos e foi definida por Kildall (1973).
- f) *Análise de tipos*: observando as operações (de máquina) efetuadas sobre determinadas variáveis, pode-se inferir seu possível tipo. Por exemplo, operações AND nunca são executadas em variáveis de ponto flutuante ou ponteiros.
- g) *Estruturação*: consiste em transformar estruturas de máquina em estruturas de alto nível, como *if* e *while*.
- h) *Geração de código*: a fase final consiste em gerar o código na linguagem-alvo. Possivelmente, vários problemas ainda restarão e terão que ser resolvidos interativamente pelo usuário.

14.7.2 ENGENHARIA REVERSA DE DADOS

Engenharia reversa de dados pode ser considerada um caso especial da engenharia reversa cujo foco está na localização, na organização e na reinterpretAÇÃO do significado dos dados de um sistema. Davis e Aiken (2000) apresentam um apanhado histórico da evolução dessa área.

Uma das atividades relacionadas à engenharia reversa de dados é a *análise de dados*. Segundo Muller et al. (2000), essa atividade consiste em recuperar um modelo de dados atualizado (a partir de um sistema em operação), estruturalmente completo e semanticamente anotado. A atividade é particularmente difícil de ser automatizada. A recuperação dos modelos de bancos de dados (quando existem) é relativamente simples, mas em geral essas estruturas não contêm informações semânticas e estruturais completas sobre os dados, que acabam sendo diluídas em código executável e documentação.

Posfácio

O futuro da Engenharia de Software

A ciência da computação é uma área que evolui muito rapidamente, todos sabem. Assim, o leitor deste livro poderia se perguntar: por quanto tempo estes conhecimentos ainda serão úteis? De fato, a evolução das tecnologias e arquiteturas (hoje a computação já está até nas “nuvens”) tem criado grandes desafios e oportunidades nas atividades de desenvolvimento de software. Com o surgimento de geradores de código automatizados realmente efetivos e a consolidação de padrões de projeto e linguagens de programação de nível cada vez mais alto, as atividades de desenvolvimento, especialmente de programação e de teste de código, têm se transformado.

Apesar disso, projetos continuarão a ser feitos e precisarão ser gerenciados. Seus riscos continuarão a ser examinados e o esforço necessário para desenvolvê-los continuará a ser estimado, mesmo que esse esforço seja cada vez menor relativamente às funcionalidades implementadas. Além disso, a qualidade, tanto do produto final quanto dos processos de trabalho, continuará sempre a ser buscada. Assim, por mais avançadas que sejam as ferramentas de desenvolvimento e por mais automatizados que sejam os geradores de código, o erro humano sempre estará presente, e por muito tempo ainda o teste de software será uma necessidade, nem que seja apenas para verificar se os requisitos foram perfeitamente compreendidos. Então, embora novas tecnologias, normas e abordagens surjam, os conhecimentos aqui examinados continuarão a ser necessários por tempo indeterminado.

Novas tecnologias vão surgir, assim como maneiras mais eficazes e mais eficientes de desenvolver software. Mas o fator humano continuará sendo o motor sem o qual essas tecnologias e ferramentas nada fazem. Os humanos, que estão à frente desse processo, são os depositários e desenvolvedores desses conhecimentos. Um longo caminho já foi trilhado em termos de criação de conceitos e práticas úteis para desenvolver sistemas. Resta agora aplicar o que já foi descoberto e, a partir daí, evoluir ainda mais.

Referências

- AGILE. *Manifesto for Agile Software Development*, 2011. Disponível em <<http://agilemanifesto.org>>. Acesso em: jun. 2010.
- AHERN, D., ARMSTRONG, J., CLOUSE, A., FERGUSON, J., HAYES, W. & NIDIFFER, K. *CMMI SCAMPI Distilled: Appraisals for Process Improvement*. Addison-Wesley Professional, 2005.
- AHN, Y., SUH, J., KIM, S. & KIM, H. The Software Maintenance Project Effort Estimation Model Based on Function Points. *Journal of Software Maintenance and Evolution: Research and Practice*. 2003. p. 71-85.
- ALBRECHT, A. J. Measuring Application Development Productivity. *Proceedings of the Joint SHARE/GUIDE and IBM Application Development Symposium*. Chicago, 1979. p. 83-92.
- _____ & GAFFNEY JR., J. E. Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation. *IEEE Transactions on Software Engineering*, 1983. p. 639-48.
- ALVIM, P. *Tirando o Máximo do Java EE 5 Open-Source com jCompany® Developer Suite*. Belo Horizonte: Powerlogic Publishing, 2008.
- AMBLER, S. W. *A Manager's Introduction to The Rational Unified Process (RUP)*. Ambsoft, 2005.
- _____. *The Operations and Support Discipline: Scaling Agile Software Development*, 2010. Enterprise Unified Process (EUP). Disponível em <www.enterpriseunifiedprocess.com/essays/operationsAndSupport.html>. Acesso em: mar. 2010.
- _____ & CONSTANTINE, L. L. *The Unified Process Elaboration Phase: Best Practices in Implementing the UP*. Elsevier, 2000.
- _____ & JEFFRIES, R. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. John Wiley and Sons, 2002.
- _____, NALBONE, J. & VIZDOS, M. J. *The Enterprise Unified Process: Extending the Rational Unified Process*. Prentice-Hall, 2005.
- _____ & CONSTANTINE, L. L. *The Unified Process Construction Phase: Best Practices in Implementing the UP*. Elsevier, 2000a.
- ANDA, B. Estimating Software Development Effort Based on Use Cases – Experiences from Industry. *4th International Conference on the Unified Modelling Language*. Toronto: Springer-Verlag, 2001. p. 487-502.
- ANDERSON, D. J. *Agile Management for Software Engineering: Applying the Theory of Constraints for Business Results*. Prentice-Hall Professional, 2004.
- ARMBRUST, O., KATAHIRA, M., MIYAMOTO, Y., MÜNCH, J., NAKAO, H. & OCAMPO, A. Scoping Software Process Lines. *Software Process: Improvement and Practice*, 14(3), 2009, p. 181-97.
- BASILI, V. R., CALDIERA, G. & ROMBACH, H. D. (1994). *The Goal Question Metric Approach*, 1994.
- BAUER, F. L. *Software Engineering: Report on a conference sponsored by the NATO Scientific Committee*. Garmish, 1968. Disponível em <<http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF>>. Acesso em: jul. 2012.
- BECK, K. *Test-Driven Development by Example*. Addison Wesley, 2003.
- BEIZER, B. *Software Testing Techniques*. 2. ed. Van Nostrand Reinhold, 1990.
- BELCHIOR, A. D. *Um Modelo Fuzzy para Avaliação da Qualidade de Software*. Tese de Doutorado, UFRJ, COPPE, Rio de Janeiro, 1997.
- BENNETT, K. Software Maintenance: A tutorial. In: M. Dorfman & e R. Thayer. *Software Engineering*. IEEE Computer Society Press, 2000.
- BLOCH, A. *Murphy's Law and Other Reasons Why Things go Wrong*. Methuen, 1977.
- BOEHM, B. W. A Spiral Model for Software Development and Enhancement. *ACM SIGSOFT Software Engineering Notes*, 11(4), 1986, p. 14-24.
- _____. *Software Cost Estimation With COCOMO II*. Prentice-Hall, 2000.
- _____. *Software Engineering Economics*. Prentice Hall, 1981.
- BOOCH, G. *Object-Oriented Analysis and Design With Applications*. 2. ed. Santa Clara, CA: Addison-Wesley, 1994.
- BRAVO, M. V. *Abordagens para o Ensino de Práticas de Programação eXtrema*. Dissertação de Mestrado, USP, IME, São Paulo, 2010.
- BRAZ, M. & VERGILIO, S. Software Effort Estimation Based on Use Cases. *Proceedings of the 30th Annual International Computer Software and Applications Conference*, 2006. p. 221-28.
- BROOKS, F. *The Mythical Man-Month*. Addison-Wesley, 1975.
- BUDDE, R., KRAUTZ, K., KUHLENKAMP, K. & ZULLIGHOVEN, H. *Prototyping: An Approach to Evolutionary System Development*. Berlin: Springer-Verlag, 1992.
- CANTOR, M. *Rational Unified Process for Systems Engineering*. IBM, 2003.
- CARR, M. J., KONDA, S. L., MONARCH, I., ULRICH, F. C. & WALKER, C. F. *Taxonomy-Based Risk Identification*. SEI. Pittsburgh: Carnegie Mellon University, 1993.
- CERI, S., FRATERNALI, P., BONGIO, A., BRAMBILLA, M., COMAI, S. & MATERA, M. *Designing Data-Intensive Web Applications*. Morgan Kaufmann Publishers, 2003.
- CHIKOFSKY, E. J. & CROSS II, J. H. Reverse Engineering and Design Recovery: A taxonomy. *IEEE Software*, 7(1), 1990, p. 13-7.
- CLEGG, D. & BARKER, R. *Case Method Fast-Track: A RAD Approach*. Addison-Wesley, 2004.
- CLELAND-HUANG, J., ZEMONT, G. & LUKASIK, W. A Heterogeneous Solution for Improving the Return on Investment of Requirements Traceability. *12th IEEE International Conference on Requirements Engineering*, 2004, p. 230-9.
- COAD, P., DE LUCA, J. & LEFEBVRE, E. *Java Modeling In Color With UML: Enterprise Components and Process*, Prentice-Hall 1997.
- COCKBURN, A. *Crystal Clear: A Human-Powered Methodology for Small Teams*. Addison-Wesley Professional, 2004.
- CONNER, D. R. & PATTERSON, R. W. Building Commitment to Organizational Change. *Training and Development Journal*, 36(4), 1982, p. 18-26.
- CRINNION, J. *Evolutionary Systems Development, a Practical Guide to the Use of Prototyping Within a Structured Systems Methodology*. New York: Plenum Press, 1991.
- CROSBY, P. *Quality is Free*. McGraw-Hill, 1979.
- CROWTHER, M. *Software Test Metrics: Key Metrics and Measures for Use Within the Test Function*, 2012. Disponível em <www.cyreath.co.uk/papers/Cyreath_Software_Test_Metrics.pdf>. Acesso em: jan. 2012.
- DAVIS, K. H. & AIKEN, P. H. Data Reverse Engineering: A Historical Survey. *IEEE Seventh Working Conference on Reverse Engineering (WCW 2000)*. Brisbane: Australia, 2000.
- DE TOLEDO, R. *Por que usar "story points"?* Disponível em <<http://visaoagil.files.wordpress.com/2009/01/storypoints.pdf>>. Acesso em: jan. 2009.

- DEGRACE, P. & STAHL, L. H. *Wicked Problems, Righteous Solutions: A Catalog of Modern Engineering Paradigms*. Prentice-Hall, 1990.
- DELAMARO, M. E., MALDONADO, J. C. & JINO, M. *Introdução ao Teste de Software*. Rio de Janeiro: Elsevier, 2007.
- DIJKSTRA, E. W. The Humble Programmer. *Communications of the ACM*, 15(10), 1971, p. 859-66.
- DINSMORE, P. C., CABANIS-BREWIN, J., ABDOLLAHYAN, F., ANSELMO, J. L., COTA, M. F. & CAVALIERI, A. *AMA – Manual de Gerenciamento de Projetos*. Brasport, 2009.
- DROMEY, R. G. *Software Product Quality: Theory, Model and Practice*. Brisbane: Griffith University, Software Quality Institute, 1998.
- DUARTE, J. *Gestão de Riscos e Oportunidades em Comunicação*, 2008. Comunicação e Crise. Disponível em <www.comunicacaoecrire.com/palestras/gestaobrasil2008jorge.pdf>. Acesso em: jan. 2012.
- DUSTIN, E. Automate Regression Tests When Feasible. In: E. Dustin. *Effective Software Testing: 50 Specific Ways to Improve Your Testing*. Addison-Wesley Professional, 2002.
- ELLIS, R. *Project Management: Project Lifecycle Planning*, 2010.
- EMAM, K. E., MELO, W. & DROUIN, J.-N. *Spice: The Theory and Practice of Software Process Improvement and Capability Determination*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1997.
- ERDIL, K., FINN, E., KEATING, K., MEATTLE, J., PARK, S. & YOUNG, D. *Software Maintenance as Part of the Software Life Cycle*. Tufts University, Computer Science, 2003.
- FAGAN, M. E. Advances in Software Inspections. *IEEE Transactions on Software Engineering*, SE-12(7), 1986, p. 744-51.
- _____. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 15(3), 1976, p. 182-211.
- FELIX, L. *O que significam Story Points*, 2009. Código Ágil. Disponível em <http://lucianofelix.wordpress.com/2009/06/10/o-que-significam-story-points/>. Acesso em: jan. 2012.
- FERREIRA, M. G. *Melhoria de Processo de Software sob a Ótica da Gestão de Mudança Organizacional: A necessidade de gerenciar e de motivar as pessoas durante a implementação*. Dissertação de Mestrado, UFSC, PPGCC, Florianópolis, 2011.
- _____. & WAZLAWICK, R. S. Complementing the SEI-IDEAL Model with Deployers Real Experiences: The Need to Address Human Factors in SPI Initiatives. *XIV Iberoamerican Conference on Software Engineering*, 2011.
- FOWLER, P. & RIFKIN, S. *Software Engineering Process Group Guide*. SEI, 1990.
- GENUCHTEN, M., CORNELISSEN, W. & DIJK, C. V. Supporting Inspections with an Electronic Meeting System. *Journal of Management Information Systems*, 14(3), 1997, p. 165-79.
- GILB, T. *Principles of Software Engineering Management*. Reading: Addison-Wesley, 1987.
- GOMES, A. S., MEDEIROS, L. M., ALVES, C. F., CAPARICA, F., NIBON, R. & VASCONCELOS, A. M. Uso de Storyboards para a Documentação dos Requisitos no Desenvolvimento Distribuído de Software. *I Workshop de Desenvolvimento Distribuído de Software (WDDS) – Simpósio Brasileiro de Engenharia de Software (SBES)*. João Pessoa, PB: SBC, 2007.
- GRAY, L. *A Comparison of IEEE/EIA 12207, ISO/IEC 12207, J-STD-016, and MIL-STD-248 for Acquirers and Developers*. Abelia Corporation, 1999.
- HIGHSMITH, J. A. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. New York: Dorset House, 2000.
- HOARE, C. A. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10), 1969, p. 576-80.
- HOBATR, C. & MALLOY, B. A. The Design of an OCL Query-based Debugger for C++. *Proceedings of the 2001 ACM Symposium on Applied Computing*. Las Vegas, USA: ACM, 2001, p. 658-62.
- IBARRA, G. *Full Use Case Size (FUCS): Estimativa de software com base no tamanho de casos de uso*. UFSC, PPGCC. Florianópolis: UFSC, 2011.
- IEEE COMPUTER SOCIETY. *SWEBOK: Guide to the Software Engineering Body of Knowledge – 2004 Version*. A. Abran, J. W. Moore, P. Bourque & R. Dupuis (Eds.). IEEE Computer Society, 2004.
- IMPERIAL, J. C. *Técnicas para uso do Cálculo de Hoare em PCC*. Dissertação de Mestrado, PUC-RJ, Rio de Janeiro, 2003.
- IVERSEN, J. H., MATHIASSEN, L. & NIELSEN, P. Managing Risk in Software Process Improvement: An Action Research Approach. *MIS Quarterly*, 28(3), 2004, p. 395-433.
- JACOBSON, I. *The Object Advantage: Business Process Reengineering with Object Technology*. Addison-Wesley, 1994.
- _____. The Use-Case Construct in Object-Oriented Software Engineering. In: J. M. Carroll (Ed.). *Scenario-Based Design*. Wiley and Sons, 1995.
- _____. BOOCHE, G. & RUMBAUGH, J. *The Unified Software Development Process*. Addison-Wesley, 1999.
- JONES, C. *Applied Software Measurement, Assuring Productivity and Quality*. New York, NY, USA: McGraw-Hill, 1996.
- JONES, T. C. *Estimating Software Costs*. McGraw-Hill, 1998.
- _____. *Software Cost Estimation in 2002*. CrossTalk, 2002.
- KAMAL, M. W. & AHMED, M. A. A Proposed Framework for Use Case Based Effort Estimation using Fuzzy Logic: Building Upon the Outcomes of a Systematic Literature Review. *International Journal on New Computer Architectures and Their Applications*, 1(4), 2011, p. 976-99.
- KARNER, G. *Metrics for Objectivity*. Sweden: Thesis, University of Linköpm, 1993.
- KEEFER, G. *Extreme Programming Considered Harmful for Reliable Software Development 2.0*. Draft: Avoca, 2003.
- KEHOE, R. & JARVIS, A. *ISO 9000-3: A Tool for Software Product and Process Improvement*. New York, USA: Springer Verlag, 1996.
- KERZNER, H. *In Search of Excellence in Project Management: Successful Van Nostrand Reinhold*, 1998.
- KILDALL, G. A Unified Approach to Global Program Optimization. *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1973, p. 194-206.
- KITCHENHAM, B. & LAWRENCE, S. Software Quality: The Elusive Target. *IEEE Software*, 13(1), 1996, p. 12-21.
- KNIBERG, H. *Scrum and XP from the Trenches: How We Do Scrum*, InfoQ, 2007.
- KOSKELA, L. *Test Driven: TDD and Acceptance TDD for Java Developers*. Manning Publications, 2007.
- KOTTER, J. P. Leading Change: Why Transformation Efforts Fail? *Engineering Management Review*, 37(3), 2006, p. 42-8.
- KROLL, P. & KRUCHTEN, P. *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*. Addison Wesley, 2003.
- KRUCHTEN, P. *The Rational Unified Process – An Introduction*. 3. ed. Addison Wesley, 2003.
- KULESZA, U. *Linhás de Processo de Software: Conceitos, Técnicas e Ferramentas*, 2011. SBQS. Disponível em <www.slideshare.net/uirakulesza/linhas-de-processos-de-software-minicurso-sbqs-2011>. Acesso em: jan. 2012.
- LAPLANTE, P. *What Every Engineer Should Know about Software Engineering*. Boca Raton, CA: CRC, 2007.
- LARMAN, C. *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and the Unified Process*. Prentice Hall, 2001.
- LEHMAN, M. M. Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE*, 68(9), 1980, p. 1.060-76.

- _____ & J. F. RAMIL, P. D. Metrics and Laws of Software Evolution – The Nineties View. *Proc. 4th International Software Metrics Symposium*. Albuquerque, NM, USA, 1997, p. 20-32.
- LEITE, J. *Verificação e Testes de Programas*, 2000. Notas de aula de Engenharia de Software. Disponível em <www.dimap.ufrn.br/~jair/ES/c8.html>. Acesso em: jan. 2012.
- LENZ, G. & MOELLER, T. *.NET: A Complete Development Cycle*. Pearson Education Inc., 2004.
- LEVESON, N. G. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.
- LINDEN, T. A. Operating System Structures to Support Security and Reliable Software. *ACM Computing Surveys*, 8(4), 1976, p. 409-45.
- LINGER, R. C. & TRAMMELL, C. J. *Cleanroom Software Engineering – Reference Model – Version 1.0*. Technical Report, SEI, 1996.
- LONGSTREET, D. *Function Points Analysis Training Course*, 2012. Software Metrics. Disponível em <www.softwaremetrics.com/Function%20Point%20Training%20Booklet%20New.pdf>. Acesso em: jan. 2012.
- LUCCHESI, C. L., SIMON, I., SIMON, J. & KOWALTOWSKI, T. *Aspectos Teóricos da Computação*. Rio de Janeiro: Ao Livro Técnico e Científico, 1979.
- MAR, K. *Seven Common Sprint Burndown Graph Signatures*, 2006. Disponível em <<http://kanemar.com/2006/11/07/seven-common-sprint-burndown-graph-signatures/>>. Acesso em: ago. 2011.
- MARTIN, J. *Rapid Application Development*, New York: McMillan, 1990.
- MCCABE, T. A Complexity Measure. *IEEE Transactions on Software Engineering*, 1972, p. 308-20.
- MCCONNELL, S. *Rapid Development*. Redmond, Washington: Microsoft Press, 1996.
- MCFEELEY, B. *IDEAL: A User's Guide for Software Process Improvement*. CMU, SEI, 1996.
- MEYER, B. *Object-Oriented Software Construction*. 2. ed. Prentice Hall, 1997.
- MILLS, E. E. *Software Metrics – SEI Curriculum Module SEI-CM-12-1.1*. SEI, 1988.
- MILLS, H., DYER, M. & LINGER, R. Cleanroom Software Engineering. *IEEE Software*, 4(5), 1987, p. 19-25.
- MOHAGHEGI, P., ANDA, B. & CONRADI, R. Effort Estimation of Use Cases for Incremental Large Scale Software Development. *International Conference on Software Engineering*, 2005.
- MUKHIJA, A. *Seminar on Software Cost Estimation*, 2003. Disponível em <https://files.ifi.uzh.ch/erg/arvo/courses/seminar_ws02/reports/Seminar_9.pdf>. Acesso em: jan. 2012.
- MÜLLER, H. A., JAHNKE, J. H., SMITH, D. B., STOREY, M. A., TILLEY, S. R. & WONG, K. Reverse Engineering: A Roadmap. *International Conference on Software Engineering (ICSE)*. Limerick, Ireland, 2000, p. 47-60.
- MYERS, G. J., SANDLER, C., BADGETT, T. & THOMAS, T. M. *The Art of Software Testing*. 2. ed. New Jersey: John Wiley & Sons, 2004.
- NORTHROP, L. *Software Product Line Adoption Roadmap*. Pittsburgh: SEI, 2004.
- _____. *Software Product Lines Essentials*, 2008. SEI-Carnegie Mellon University. Disponível em <www.sei.cmu.edu/library/assets/spl-essentials.pdf>. Acesso em: abr. 2010.
- OCAMPO, A. & ARMBRUST, O. Software Process Lines and Standard Traceability Analysis. *WOCs 2009*. Tokyo, 2009.
- ORACLE. *Oracle Unified Method (OUM)*, 2009. Oracle Brief. Disponível em <www.oracle.com/consulting/library/briefs/oracle-unified-method.pdf>. Acesso em: fev. 2010.
- _____. *Oracle Unified Method (OUM) White Paper*, 2007. Disponível em <http://expobadge.com/dldev/partners/file/OUM_White_Paper.pdf>. Acesso em: jan. 2012.
- PALMER, S. R. & FELSING, J. M. *A Practical Guide to Feature-Driven Development*, Prentice Hall, 2002.
- PARK, R. *Software Size Measurement: A Framework for Counting Source Statements*. CMU, SEI. Pittsburgh: CMU-SEI, 1992.
- PARKINSON, C. N. Parkinson's Law. *The Economist*, 1995.
- PMI. *A Guide to the Project Management Body of Knowledge (PMBOK Guide)*. Project Management Institute Inc., 2004.
- PRESSMAN, R. S. *Software Engineering: A Practitioner's Approach*. 6. ed. McGraw-Hill Education, 2005.
- REENSKAUG, T. M. The Model-View-Controller (MVC) – Its Past and Present. *Java Zone*. Oslo, 2003.
- RIBU, K. *Estimating Object-Oriented Software Projects with Use Cases*. Master of Science Thesis, University of Oslo, Department of Informatics, 2001.
- ROBIOLO, G. & OROSCO, R. Employing Use Cases to Early Estimate Effort with Simpler Metrics. *Innovations in Systems and Software Engineering*, 4(1), 2008, p. 31-43.
- ROMBACH, D. *Integrated Software Process and Product Lines*. SPW 2005. Beijing, 2005.
- ROYCE, W. W. Managing the Development of Large Software Systems. *Proceedings of IEEE WESCON*, 1970, p. 1-9.
- SANTANA, F. L. *Problemas em Iniciativas de Melhoria de Processos de Software sob a Ótica de uma Teoria de Intervenção*. Dissertação de Mestrado, UFPE, Recife, 2007.
- SANTOS, L. B. & PRETZ, E. *Framework para Especialização de Modelos de Qualidade de Produtos de Software*. Monografia, Centro Universitário Feevale, 2009.
- SANTOS, R. N. & WAZLAWICK, R. S. Rastreabilidade Indutiva Aplicada a Artefatos de Software. *VI Experimental Software Engineering Latin American Workshop*, 2009.
- SCHAEFER, H. Metrics for Optimal Maintenance Management. *Proceedings Conference on Software Maintenance*. Washington: IEEE Computer Society Press, 1985, p. 114-9.
- SCHELL, J. *The Art of Game Design*. Elsevier, 2008.
- SCHNEIDER, G. & WINTERS, J. P. *Applying Use Cases: A Practical Guide*. Addison Wesley, 1998.
- SCHWABER, K. & BEEDLE, M. *Agile Software Development with Scrum*. Prentice-Hall, 2001.
- SCHWARTZ, M. Reverse-Engineering. *Computerworld*. 2001.
- SEI-CMU. *CMMI for Development Version 1.3*. Carnegie Mellon University, 2010.
- SETTIMI, R., CLELAND-HUANG, J., KHADRA, O. B., MODY, J., LUKASIK, W. & De PALMA, C. Supporting Software Evolution through Dynamically Retrieving Traces to UML Artifacts. *Proceedings of the Principles of Software Evolution*, 2004.
- SINGH, S. *O Último Teorema de Fermat*. Rio de Janeiro: Record, 1998.
- SOFTEX. *MPS.BR – Melhoria de Processo do Software Brasileiro – Guia Geral*. Softex, 2011.
- SOLINGEN, R. V. & BERGHOUT, E. *The Goal/Question/Metric Method: A Practical Guide for Quality Improvement Of Software Development*. Cambridge: McGraw-Hill, 1999.
- SOMMERVILLE, I. *Engenharia de Software*. 6. ed. Addison-Wesley, 2003.
- SPILLNER, A. The W Model: Strengthening the Bond Between Development and Test. *STAReast Conference*. Orlando, USA, 2002.
- STANDISH GROUP. *Chaos Report*, 1995. Disponível em <www.projectsmart.co.uk/docs/chaos-report.pdf>. Acesso em: mar. 2010.
- SURYN, W. & ABRAN, A. ISO/IEC SQuaRE. The Second Generation of Standards for Software Product Quality. *Proceedings of 7th IASTED International Conference on Software Engineering and Applications, SEA 2003*. Marina del Rey, CA, USA, 2003.

- SYMONS, C. R. Function Points Analysis: Difficulties and Improvements. *IEEE Transactions on Software Engineering*, 14(1), 1988, p. 2-11.
- TAKEUCHI, H. & NONAKA, I. The New Product Development Game. *Harvard Business Review*, 1986, p. 137-46.
- TAUSWORTHE, R. C. The Work Breakdown Structure in Software Project Management. *Journal of Systems and Software*, 1, 1980, p. 181-6.
- TEIXEIRA, H. V. *A Crise de Software*, 2010. Disponível em <<http://pt.scribd.com/doc/37503268/A-Crise-de-SW-Hugo-Vidal-Teixeira>>. Acesso em: ago. 2011.
- TOLFO, C. & WAZLAWICK, R. S. The Influence of Organizational Culture on the Adoption of Extreme Programming. *The Journal of Systems and Software*, 81(11), 2008, p. 1.955-67.
- TURLEY, F. *An Introduction to PRINCE2®*. Version 1.1, 2010.
- VAN LOON, H. *Process Assessment and ISO 15504*. Springer, 2007.
- VASCONCELOS, A. M., ROUILLER, A. C., MACHADO, C. A. & MEDEIROS, T. M. *Introdução à Engenharia de Software e à Qualidade de Software*. Lavras, MG: UFLA/FAEPE, 2006.
- VON MAYRHAUSER, A. & VANS, A. M. Program Comprehension During Software Maintenance and Evolution. *Computer*, 28(8), 1995, p. 44-55.
- VON STAA, A. *Acompanhamento de Projetos*. PUC-RJ, Departamento de Informática. Rio de Janeiro: PUC-RJ, 2003.
- WANGENHEIM, C. G. *Utilização do GQM no Desenvolvimento de Software*. São Leopoldo: Unisinos, 2000.
- WAZLAWICK, R. S. *Análise e Projeto de Sistemas de Informação Orientados a Objetos*. 2. ed. Rio de Janeiro: Elsevier, 2011.
- _____. *Metodologia de Pesquisa para Ciência da Computação*. Rio de Janeiro: Elsevier, 2009.
- WEISS, D. M. & LAY, C. T. *Software Product-Line Engineering: A Family-Based Software Development Process*. Reading: Addison-Wesley, 1999.
- WELLS, D. *The Rules of Extreme Programming*, 2009. Disponível em <www.extremeprogramming.org/rules.html>. Acesso em: mar. 2010.
- WEST, D. *Planning a Project with the Rational Unified Process*. Rational Software White Paper. TP 151, 2002.
- WIEGERS, K. E. Know Your Enemy: Introduction to Risk Management. In: K. E. Wieggers, *Practical Project Initiation: A Handbook with Tools*. Microsoft Press, 2007.
- XAVIER, C. M. *Gerência de Escopo em Projetos*, 2011. MBA em Gerência de Projetos. Disponível em <www.administracaovirtual.com/administracao/downloads/apostilas/Apostila-Gerencia-Escopo-Magno-FGV.pdf>. Acesso em: jan. 2012.
- YOURDON, E. *Structured Walkthroughs*. Yourdon Press, 1985.
- ZAHRAN, S. *Software Process Improvement: Practical Guidelines for Business Success*. Reading, MA, USA: Addison-Wesley, 1997.

Índice remissivo

- Abordagem
 - proativa, 44
 - reativa, 44
- Abstração, 7
- Ação, 281
- Acesso fácil a especialistas, 71
- Acompanhamento de problemas, 211
- Adaptação, 56
- Adaptações obrigatórias, 325
- Adaptive software development* (ASD), 72
- Adequação funcional, 232
- AFP, pontos de função ajustados, 162
- Agenda, 54
- Agile unified process*, 105
- Alavancagem, 283
- Ambiente, 104
 - tecnologicamente rico, 71
- Análise
 - de negócio, 53
 - de riscos, 193
 - de valor-limite, 310
 - de viabilidade, 53
 - e *design*, 95
- Analista, 5, 81
- Anunciante, 55
- Aplicando COCOMO II para as fases do UP, 152
- Aprender, 73
- Aquisição, 18
- Áreas de negócio, 48
- Áreas de processo do CMMI-DEV V1, 3, 268
- Arquitetura inicial de sistema, 54
- Arquiteturas baseadas em componentes, 8
- Artefatos, 13, 84
- Atividades, 11, 13, 83
 - de negócio, 48, 49
- Ativos preeexistentes, 43
- Atualização *on-line*, 168
- Auditória, 19
 - de configuração, 225
- AUP, *agile unified process*, 105
- Ausência de testes no escopo, 53
- Autonomia, 52
- Autorregulação, 319
- Avaliação da qualidade, 247
- Base de ativos, 43
- Baseline e release*, 222
- BDUF (*big design up front*), 25
- Blocos de construção do RUP, 80
- Browseabilidade, 65
- Calibragem do modelo, 153
- Caminho(s)
 - crítico, 125
 - impossíveis, 306
 - independentes, 302
- Campeão do projeto, 55
- Capacidade, 268
 - de manutenção, 235
- Carga, 298
 - de trabalho da equipe, 49
- Cargos, 14
- Cascata, 22
 - com redução de risco, 34
 - com subprojetos, 22, 33
 - entrelaçado, 30
- Case (*computer aided software engineering*), 6
- Caso de uso, 40, 76
- Casos de teste, 304
- Centrado na arquitetura, 76
- Checklist* de riscos, 182
- Ciclo de vida, 12
 - cascata com subprojetos, 33
 - espiral, 36
- Ciclos de entrevistas com o usuário e o cliente, 35
- Classes aos desenvolvedores, 50
- Classificação
 - das atividades de manutenção, 320
 - de métricas, 213
- CLF (construir lista de funcionalidades), 47, 48
- CMMI – *capability maturity model integration*, 266
- COCOMO, 134
- COCOMO II, 138
- Codificar e consertar, 22, 23
- Colaborar, 73
- Comentários, 48
- Como demonstrar, 57
- Compatibilidade, 235
- Complexidade
 - ciclomática, 300
 - das funcionalidades, 49
 - de atores, 171
 - dos casos de uso, 171
 - percebida, 319
- Componentes, 40
- Compreensibilidade e explicabilidade, 65
- Comunicação
 - de dados, 165
 - de riscos, 201
- Comunicação, 53, 61
 - osmótica, 69
- Concepção, 12, 77, 78
 - e planejamento, 46
- Condução de projeto de software, 209
- Confiabilidade, 232
- Configuração de software, 221
- Construção, 12, 46, 77, 79
- Construir
 - lista de funcionalidades, 47, 48
 - por funcionalidade, 47
- Controle
 - de mudanças, 8, 224
 - de rastros, 221
 - de risco, 200
 - de versão, 222
- Conversão de arquitetura, 325
- Coordenador técnico, 56

- Coragem, 61
Cornerstone, 37
Could, 53
CPF (construir por funcionalidade), 47, 51
Criação do protótipo funcional, 54
Crise do software, 1
Cronograma, 63, 126
Crystal clear, 66
- Daily scrum*, 59
Declaração de escopo, 116
Decomposição, 7
Defeito, 290
Dependências
entre as funcionalidades em termos de classes, 49
entre atividades, 124
Depuração, 313
Desenvolvedor, 5, 56, 82
Desenvolver modelo abrangente, 47, 48
Desenvolvimento, 18
do núcleo de ativos, 41
do produto, 43
iterativo, 8
orientado a testes, 62
- Design*
arquitetural, 31
detalhado, 31
simples, 62
- Designer*, 5
- Detalhamento
da dimensão de processos, 262
de atividades, 15
dos fatores técnicos, 165
- Detalhar por funcionalidade, 47, 50
- Dez mandamentos da WBS, 124
- Diagnóstico, 279
- Diagramas de sequência, 48, 50
- Disciplinas, 11, 13, 86
- DMA (desenvolver modelo abrangente), 47
- Documentação, 19
de referência, 50
- Documento de atividade, 15
- Documento descritivo, 11
- Dpf (detalhar por funcionalidade), 47, 50
- Drivers*, 291
- DSDM – *dynamic systems development method*, 52
- Duração e custo de um projeto, 164
- Dynamic systems development method*, 52
- Early-learner*, 59
- Eficácia, 52
- Eficiência
de desempenho, 234
do usuário final, 167
- Elaboração, 77, 79
- Engenharia de software
definição de, 4
princípios da, 7
- Engenharia reversa, 329
de código, 330
de dados, 332
- Engenheiro de software, 5
- Enterprise unified process*, 108
- Entrada de dados *on-line*, 167
- Entrada e saída, 11, 13
- Entrega(s), 120
em estágios, 22, 38
evolucionária, 22, 40
frequentes, 52, 68
- Envio de versões, 224
Envolver o cliente, 29
Envolvimento, 52
Equipe coesa, 62
Equipe de funcionalidades, 50
Equipe de planejamento, 49
Equipe de processo, 11, 17
Erro, 290
Escopo da SPL, 43
Escrivão, 56
Esforço, 173
Especificação do *design*, 5
Especular, 73
Espiral, 35
Estabelecimento, 280
Estimação da duração, 118, 119
Estimação de KSLLOC, 130
Estimativas de esforço, 129
Estratégia de produção, 43
Estresse, 298
Estrutura analítica da iteração, 122
Estudar a documentação, 48
Estudo dirigido de domínio, 47, 50
EUP, *enterprise unified process*, 108
Evolução da engenharia de software, 6
- Executor, 5
- Facilidade
de instalação, 169
de operação, 169
para mudança, 170
- Facilitador, 56
- Fakey-fakey*, 58
- Falha, 290
- Fase de *design*, 29
arquitetural, 31
detalhado, 31
- Fase de implementação, 31
- Fase de teste
de integração, 31
de sistema, 32
de unidade, 31
- Fases, 12
- Fatores
de escala, 140
humanos, 284
técnicos, 172
- Fazer duas vezes, 29
- FDD, 46
ferramentas para, 52
- Feature-driven development*, 46
- Fechamento, 216
- Feedback*, 52, 61
- Ferramentas
para controle de versão, 225
para manutenção de software, 323
- Fim da fase, 25, 26
- Flexibilização, 7
- Focado em riscos, 77
- Foco, 70
- Folha de tempo, 210
- Formalidade, 8
- Formar a equipe de modelagem, 47
- Fornecimento, 18
- Full use case size*, 174
- Garantia de qualidade, 19
- Generalização, 7

- Gerência, 19, 43
de configuração, 19
de mudança, 224
Gerenciamento, 64
de configuração e mudança, 219
de mudança e configuração, 101
de projeto, 86
de projetos segundo o PMBOK, 205
de requisitos, 8
de riscos, 8
do programa de SPI, 284
Gerente, 83
de projeto, 5, 55, 204
Gestão da qualidade, 241
GQM (*goal/question/metric*), 247
Grafo de fluxo, 301
Grupos de artefatos, 84

Histórias de usuário, 63

ID, 57
Identificação
das dependências entre atividades, 124
de riscos, 181
do protótipo funcional, 54
dos recursos necessários e custo, 124
dos responsáveis por atividade, 124
IMP, 57
Implantação, 55, 100
Implementação, 31, 96
Implementar classes e métodos, 51
Infraestrutura, 19
Iniciação, 278
Inicialização e encerramento, 73
Início da fase de planejamento e requisitos, 25
Inspecionar o código, 51
Inspecções Fagan, 243
Instalação de um programa de melhoria de qualidade, 241
Integração contínua, 62
Interações frequentes, 72
Inteligência artificial, 7
Interfaces (assinatura) das classes e métodos, 51
Intermediador, 55
Interpretação e classificação dos requisitos como funções, 158
ISO/IEC 15504 – spice, 259
ISO/IEC, 90003, 251
Item de configuração de software, 220
Iteração de *design* e construção, 55
Iteração do modelo funcional, 54
Iterativo e incremental, 77

Jogo de planejamento, 62
Jogos, 7

KSLOC, 130

Late-learner, 58
Lei da complexidade crescente, 319
Lei da conservação da estabilidade organizacional, 319
Lei da conservação da familiaridade, 319
Lei da mudança contínua, 318
Lei da qualidade decrescente, 320
Lei do crescimento contínuo, 319
Lei do sistema realimentado, 320
Lei fundamental da evolução de programas, 319
Leis de Lehman, 318
Levantamento dos requisitos, 29
Líder de equipe, 56

Limitações, 307
Linha de processo de software, 287
Linhas de código, 132
Linhas de produto de software, 22, 40

Manutenção, 18
adaptativa, 321
corretiva, 320
e evolução de software, 317
perfectiva, 321
preventiva, 322
Máquina de estados, 48
Marcos, 25, 120
Maturidade, 268
Medição
da qualidade, 245
de velocidade, 178
em engenharia de software, 212
em teste, 312
Melhoria, 19
de processo de software, 276
reflexiva, 68
Melhorias, 325
Metáfora, 62
Método *cleanroom*, 244
Método Fuchs, 174
Métodos ágeis, 6
Métodos de contagem para casos de uso detalhados, 174
Métodos e atributos, 48
Middle-learner, 59
Migração entre plataformas, 325
Milestones, 25
Mitos, 2, 3
Modelagem
de negócio, 88
visual, 8
Modelo ACT, 326
Modelo avançado, 138
Modelo básico, 135
Modelo cascata, 25
com redução de risco, 34
com subprojetos, 33
entrelaçado, 30
Modelo codificar e consertar (*code and fix*), 23
Modelo de manutenção de CII, 327
Modelo de objetos, 50
Modelo de processo, 12
Modelo de prototipação evolucionária, 37
Modelo de qualidade de *dromey*, 236
Modelo de qualidade *square*, 229
Modelo entrega evolucionária, 40
Modelo entregas em estágios, 38
Modelo espiral, 35
Modelo intermediário, 136
Modelo orientado a cronograma, 22, 39
Modelo *Sashimi*, 30
Modelo *Scrum*, 30, 56
Modelo V, 22, 31
Modelo W, 22, 32
Modelos ágeis, 22, 45
Modelos de estimativa de esforço de manutenção, 326
Modelos de processo prescritivos, 21
Modelos FP, 329
Modelos orientados a ferramentas, 22, 40
Módulos, 40
Monitoramento de riscos, 198
MPS.BR, 271
Múltiplas condições, 304

- Multiplicadores de esforço, 143
do *early design model*, 150
Múltiplos locais, 169
Must, 53
- Necessidade de manutenção e evolução de software, 318
Never-never, 59
Níveis de capacidade, 268
Níveis de maturidade, 268
 CMMI, 270
Níveis de teste de funcionalidade, 292
Nome, 57
Norma NBR ISO/IEC 12207, 18
Normas técnicas, 11
Notas, 57
Núcleo de ativos, 43
Número de iterações, 120
Número dos ciclos iterativos, 119
- Objetividade, 52
Objetos, 40
Open unified process, 107
OpenUp – open unified process, 107
OpenWiki, 17
Operação, 18
Oracle unified method, 109
Orientações e aprovação do usuário, 55
Otimização de performance, 325
OUM, oracle unified method, 109
- Pacote de *design*, 51
Padrões de codificação, 62
Padronização, 7
Papéis de analista, 81
Papéis de desenvolvedor, 82
Papéis de gerente, 83
Papéis de responsáveis e participantes, 11
Papéis de testador, 82
Papéis no DSDM, 55
Papéis, 81
Particionamento de equivalência, 309
Participantes, 13, 14
Passos, 15
 de atividades de negócio, 48
Perfis, 14, 56
Performance, 166
PH, 57
Planejamento, 115
 de iteração, 121
 de projeto com ciclos iterativos, 117
 de um programa de métricas, 214
Planejar por funcionalidade, 47, 49
Planejar, controlar e monitorar o teste, 29
Plano de contingência, 197
Plano de desenvolvimento, 50
Plano de gerência de riscos, 180
Plano de produção, 43
Plano de redução de impacto de risco, 197
Planos de mitigação de riscos, 194
Plateau, 59
Políticas de compartilhamento de itens, 223
Pontos de caso de uso, 170
 ajustados, 173
 não ajustados, 172
Pontos de função, 156
 ajustados, 162
 não ajustados, 160
Pontos de histórias, 177
- Pontos de variação, 41
Posse coletiva, 62
Post-architecture model, 143
PPF (planejar por funcionalidade), 47, 49
Práticas de elicitação de requisitos, 35
Práticas e objetivos específicos e genéricos, 267
Práticas XP, 62
Previsibilidade, 53
Prince2, 207
Princípio 80/20, 52
Princípio de Pareto, 52, 53
Procedimentos, 15
Processamento complexo, 168
Processamento de dados distribuído, 166
Processo, 11, 12
 Processo de adaptação, 18, 19
 Processo de avaliação, 261
 Processo de desenvolvimento de software, 11
 Processo de manutenção, 322
 Processo unificado, 222
 caracterização do, 76
 fases do, 77
 Processos de apoio, 18, 19
 Processos fundamentais, 18
 Processos organizacionais, 18
 Product backlog, 56, 57
 Product owner, 56
 Produzir documentação, 29
 Programação em pares, 62
 Programação extrema, 60
 Programador, 5
 Programadores líderes, 49
 Projects in controlled environments, 2, 207
 Projeto, 12
 Promover à versão atual (*build*), 51
 Prototipação evolucionária, 22, 37
 Prototipagem, 55
 Protótipos de interface com o usuário, 34
 Prova de correção de programas, 314
- Qualidade de processo, 251
Qualidade de produto, 229
Qualidades do software em uso, 236
Questionário EPML, 154
- Rastreabilidade, 8, 221
Rational unified process, 80
Rational unified process-systems engineering, 111
Recursos
 consumíveis, 14
 físicos, 14
 humanos, 14
 não consumíveis, 14
 necessários e custo, 124
Recursos, 11, 13
Rede Pert, 125
Reengenharia, 329
Refatoração, 62
Refinar o modelo de objetos abrangente, 48
Registro de artefatos, 212
Regra(s), 15
 8-80, 123
 dos 100%, 123
 dos níveis e do número total de atividades, 123
 de codificação, 65
 de *design*, 64
 de gerenciamento, 64
 de planejamento, 63
 de teste, 66

- Relacionamentos de itens de configuração de software, 220
Relatórios, 84
Remoção de módulos sujeitos a erros, 324
Reparação de defeitos, 323
Repositório, 223
Requisição de modificação, 40
Requisitos, 92
 de qualidade, 246
 e orientações
 de gerenciamento, 253
 para ações corretivas, 257
 para realização de projetos, 255
 relacionados a recursos, 254
 sistêmicos, 252
Resistência, 298
Resolução de problemas, 19
Respeito, 61
Responsáveis, 13, 14
 por atividade, 124
Restrições, 15
 de produção, 43
 de produto, 43
Reuniões em pé, 62
Reusabilidade, 169
Reversibilidade, 52
Revisão conjunta, 19
Revisão de negócio, 55
Revisão do protótipo, 54
Revisão e avaliação, 215
Risco(s), 40, 179
 de projeto, 182
 relacionados a pessoas, 182
 tecnológicos, 182
Ritmo sustentável, 62
RUP, *rational unified process*, 80
RUP-SE, *rational unified process-systems engineering*, 111

Sashimi, 22, 30
Scope increase, 59
Scrum, 56
 funcionamento geral do, 60
Scrum master, 56
Scrum team, 56
Segurança, 235
 pessoal, 69
Seleção de projetos, 115
Sequência de desenvolvimento, 49
Serviços, 40
Should, 53
Simplicidade, 61
Sistema de gerenciamento de configuração, 72
Sistemas de informação, 7
SLOC, 130
SMPEEM, 329
Software
 básico, 6
 científico e de engenharia, 7
 comercial, 7
 de tempo real, 6
 do ponto de vista da engenharia, tipos de, 6
 embutido ou embarcado, 7
 engineering institute 3 (SEI), 18
 pessoal, 7

Sprint, 57
Sprint backlog, 57
Sprint burndown, 58
Storyboards com o usuário, 35
Stubs, 291
Sub-rotinas, 40
Suporte a usuários, 324
Swebok, 6

Taxa de trabalho invariante, 319
Taxa de transações, 167
TDD, desenvolvimento orientado a testes, 311
Template, 13
Template e exemplo de documento de atividade, 15
Termo de abertura, 116
Testabilidade, 65
Testador, 56, 82
Teste, 99, 289, 291
 Teste automatizado, 72
 Teste de aceitação, 296
 Teste de ciclo de negócio, 297
 Teste de instalação, 299
 Teste de integração, 31, 294
 Teste de interface com usuário, 298
 Teste de performance, 298
 Teste de recuperação de falha, 299
 Teste de regressão, 297
 Teste de segurança, 299
 Teste de sistema, 32, 295
 Teste de unidade, 31, 51, 292
 Teste e depuração, 291
 Teste estrutural, 299
 Teste funcional, 308
 Testes de aceitação, 62
 Testes suplementares, 297
Throw-away, 37
Tipos de atividades de manutenção e suas métricas, 323
Transformando pontos de função em KSLOC, 131
Transição, 78, 80
Treinamento, 19, 55

UFP, pontos de função não ajustados, 160
UML (*unified modeling language*), 8
Usabilidade, 234
Uso do sistema, 166

V model, 31
Validação, 19, 291
Verificação, 19, 291
 contínua da qualidade, 8
Versões
 de itens de configuração de software, 221
 pequenas, 62
Virtual scrum board, 57
Visionário, 55

Walkthrough, 242
Waterfall, 25
WBS – estrutura analítica da iteração, 122
WFM, 25
Workflows, 85
Would, 53

XP, extreme programming, 60