

Engenharia de Software

Uma Abordagem Profissional

Sétima Edição



Roger S. Pressman

**Mc
Graw
Hill**





EDITORIA AFILIADA

P934c Pressman, Roger S.
Engenharia de software [recurso eletrônico] : uma
abordagem profissional / Roger S. Pressman ; tradução
Ariovaldo Griesi ; revisão técnica Reginaldo Arakaki, Julio
Arakaki, Renato Manzan de Andrade. – 7. ed. – Dados
eletrônicos. – Porto Alegre : AMGH, 2011.

Editedo também como livro impresso em 2011.
ISBN 978-85-8055-044-3

1. Engenharia de programas de computador. 2. Gestão de
projetos de softwares. I. Título.

CDU 004.41

Catalogação na publicação: Ana Paula M. Magnus – CRB-10/2052

Engenharia de Software

UMA ABORDAGEM PROFISSIONAL

SÉTIMA EDIÇÃO

Roger S. Pressman, Ph.D.

Tradução

Ariovaldo Griesi
Mario Moro Fecchio

Revisão Técnica

Reginaldo Arakaki
Professor Doutor do Departamento de Engenharia de Computação e Sistemas Digitais da EPUSP

Julio Arakaki
Professor Doutor do Departamento de Computação da PUC-SP

Renato Manzan de Andrade
Doutorando do Departamento de Engenharia de Computação e Sistemas Digitais da EPUSP

Versão impressa
desta obra: 2011



AMGH Editora Ltda.

2011

Obra originalmente publicada sob o título
Software Engineering: a Practitioner's Approach, 7th Edition
ISBN 0073375977 / 9780073375977

© 2011, The McGraw-Hill Companies, Inc., New York, NY, EUA

Preparação do original: *Mônica de Aguiar Rocha*
Leitura final: *Vera Lúcia Pereira*
Capa: *Triall Composição Editorial Ltda*, arte sobre capa original
Editora sênior: *Arysinha Jacques Affonso*
Assistente editorial: *César Crivelaro*
Diagramação: *Triall Composição Editorial Ltda*

Reservados todos os direitos de publicação em língua portuguesa à
AMGH Editora Ltda (AMGH Editora é uma parceria entre
Artmed® Editora S.A. e McGraw-Hill Education)
Av. Jerônimo de Ornelas, 670 - Santana
90040-340 Porto Alegre RS
Fone (51) 3027-7000 Fax (51) 3027-7070

É proibida a duplicação ou reprodução deste volume, no todo ou em parte,
sob quaisquer formas ou por quaisquer meios (eletrônico, mecânico, gravação,
fotocópia, distribuição na Web e outros) sem permissão expressa da Editora.

SÃO PAULO
Av. Embaixador Macedo Soares, 10.735 - Pavilhão 5 - Cond. Espace Center
Vila Anastácio 05095-035 São Paulo SP
Fone (11) 3665-1100 Fax (11) 3667-1333

SAC 0800 703-3444

IMPRESSO NO BRASIL
PRINTED IN BRAZIL

O AUTOR

Roger S. Pressman é uma autoridade reconhecida internacionalmente nas tecnologias em melhoria de processos de software e engenharia de software. Por mais de três décadas, trabalhou como engenheiro de software, gerente, professor, autor e consultor, concentrando-se nas questões da engenharia de software. Como profissional técnico e gerente nesta área, trabalhou no desenvolvimento de sistemas CAD/CAM para avançadas aplicações de engenharia e manufatura. Também ocupou cargos com responsabilidade pela programação científica e de sistemas.

Após receber o título de Ph.D. em engenharia da University of Connecticut, Pressman começou a dedicar-se à vida acadêmica ao se tornar professor-adjunto de Engenharia da Computação na University of Bridgeport e diretor do Centro de Projeto e Fabricação Apoiados por Computador (Computer-Aided Design and Manufacturing Center) dessa Universidade.

Atualmente, é presidente da R. S. Pressman & Associates, Inc., uma consultoria especializada em treinamento e métodos em engenharia de software. Atua como consultor-chefe e projetou e desenvolveu o *Essential Software Engineering*, um conjunto de vídeos em engenharia de software, e o *Process Advisor*, um sistema autodirigido para aperfeiçoamento de processos de software. Ambos são usados por milhares de empresas em todo o mundo. Mais recentemente, trabalhou em conjunto com a *EdistaLearning*, na Índia, desenvolvendo extenso treinamento em engenharia de software baseado na Internet.

Publicou vários artigos técnicos, escreve regularmente para periódicos do setor e é autor de sete livros técnicos. Além de *Engenharia de software: uma abordagem profissional*, foi coautor de *Web engineering* (McGraw-Hill), um dos primeiros livros a aplicar um conjunto personalizado de princípios e práticas de engenharia de software para o desenvolvimento de aplicações e sistemas baseados na Web. Também escreveu o premiado *A manager's guide to software engineering* (McGraw-Hill); *Making software engineering happen* (Prentice Hall), primeiro livro a tratar dos críticos problemas de gerenciamento associados ao aperfeiçoamento de processos de software, e *Software shock* (Dorset House), um tratado que se concentra em software e seu impacto sobre os negócios e a sociedade. O dr. Pressman participou dos comitês editoriais de uma série de periódicos do setor e, por muitos anos, foi editor da coluna "Manager" no *IEEE Software*.

É ainda um palestrante renomado, destacando-se em uma série das principais conferências do setor. É associado da IEEE e Tau Beta Pi, Phi Kappa Phi, Eta Kappa Nu e Pi Tau Sigma.

Vive no sul da Flórida com sua esposa, Barbara. Atleta por grande parte de sua vida, continua a ser um dedicado jogador de tênis (NTRP 4.5) e jogador de golfe com handicap de apenas um dígito. Nas horas vagas, escreveu dois romances, *The Aymara bridge* e *The Puppeteer*, e planeja começar um novo romance.

*Em memória de meu querido
pai, que viveu 94 anos
e ensinou-me, acima de tudo,
que a honestidade e a integridade
seriam os meus melhores guias na
jornada da vida.*

PREFÁCIO

Quando um software é bem-sucedido — atende às necessidades dos usuários, opera perfeitamente durante um longo período, é fácil de modificar e, mais fácil ainda, de utilizar —, ele é realmente capaz de mudar as coisas para melhor. Porém, quando um software falha — quando seus usuários estão insatisfeitos, quando é propenso a erros, quando é difícil modificá-lo e mais difícil ainda utilizá-lo —, fatos desagradáveis podem e, de fato, acontecem. Todos queremos construir softwares que facilitem o trabalho, evitando pontos negativos latentes nas tentativas mal-sucedidas. Para termos êxito, precisamos de disciplina no projeto e na construção do software. Precisamos de uma abordagem de engenharia.

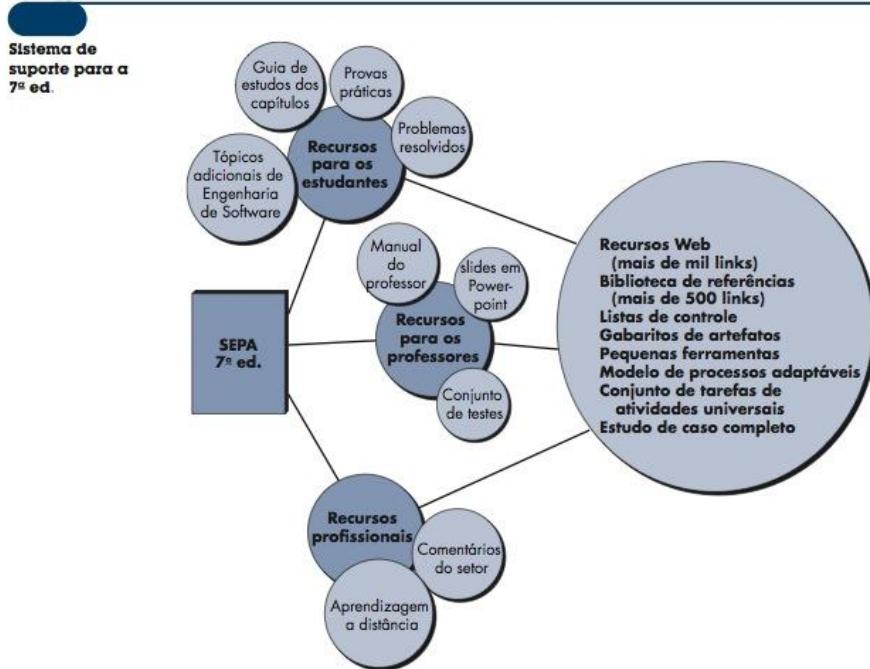
Já faz quase três décadas que a primeira edição deste livro foi escrita. Durante esse período, a engenharia de software evoluiu de algo obscuro praticado por um número relativamente pequeno de adeptos para uma disciplina de engenharia legítima. Hoje, é reconhecida como uma matéria digna de pesquisa séria, estudo consciente e debates acalorados. Neste segmento, o cargo preferido passou a ser o de engenheiro de software e não mais o de programador. Modelos de processos de software, métodos de engenharia de software, bem como ferramentas de software, vêm sendo adotados com sucesso em um amplo espectro de aplicações na indústria.

Apesar de gerentes e profissionais envolvidos com a área técnica reconhecerem a necessidade de uma abordagem mais disciplinada em relação ao software, eles continuam a discutir a maneira como essa disciplina deve ser aplicada. Muitos indivíduos e empresas desenvolvem software de forma desordenada, mesmo ao construírem sistemas dirigidos às mais avançadas tecnologias. Grande parte de profissionais e estudantes não estão cientes dos métodos modernos. E, como consequência, a qualidade do software que produzem é afetada. Além disso, a discussão e a controvérsia sobre a real natureza da abordagem de engenharia de software continuam. A engenharia de software é um estudo repleto de contrastes. A postura mudou, progressos foram feitos, porém, falta muito para essa disciplina atingir a maturidade.

A sétima edição do livro *Engenharia de software: uma abordagem profissional* é destinada a servir de guia para uma disciplina de engenharia em maturação. Assim como nas seis edições anteriores, esta é voltada tanto para estudantes no final do curso de graduação ou no primeiro ano de pós-graduação e para profissionais da área.

A sétima edição é muito mais do que uma simples atualização. O livro foi revisado e reestruturado para adquirir maior fluência em termos pedagógicos e enfatizar novos e importantes processos e práticas de engenharia de software. Além disso, um "sistema de suporte" revisado e atualizado, ilustrado na figura da página seguinte, fornece um conjunto completo de recursos para estudantes, professores e profissionais complementarem o conteúdo do livro. Tais recursos são apresentados como parte do site do livro em inglês (www.mhhe.com/pressman), especificamente projetado para este livro.

A Sétima Edição. Os 32 capítulos foram reorganizados em cinco partes, que diferem consideravelmente da sexta edição para melhor compartmentalizar tópicos e auxiliar os professores que talvez não tenham tempo suficiente para abordar o livro inteiro em um semestre.



A Parte 1, *O Processo*, apresenta uma série de visões diferentes de processo de software, considerando todos os modelos importantes e contemplando o debate entre as filosofias de processos ágeis e preceptivos. A Parte 2, *Modelagem*, fornece métodos de projeto e análise com ênfase em técnicas orientadas a objetos e modelagem UML. O projeto baseado em padrões, bem como o projeto para aplicações da Web, também é considerado. A Parte 3, *Gestão da Qualidade*, apresenta conceitos, procedimentos, técnicas e métodos que permitem a uma equipe de software avaliar a qualidade de software, revisar produtos gerados por engenharia de software, realizar procedimentos SQA e aplicar uma estratégia e tática de testes eficazes. Além disso, são considerados também métodos de modelagem e verificação formais. A Parte 4, *Gerenciamento de Projetos de Software*, aborda tópicos relevantes para os que planejam, gerenciam e controlam um projeto de desenvolvimento de software. A Parte 5, *Tópicos Avançados*, considera aperfeiçoamento de processos de software e tendências da engenharia de software. Preservando a tradição de edições anteriores, usa-se uma série de quadros ao longo do livro para apresentar as atribuições de uma equipe de desenvolvimento (fictícia), bem como fornecer material suplementar sobre métodos e ferramentas relevantes para os tópicos do capítulo. Dois novos apêndices fornecem tutoriais curtos sobre a filosofia de orientação a objetos e UML para os leitores que não estão familiarizados com esses itens.

A organização da edição em cinco partes possibilita ao professor "aglutinar" tópicos tomando como base o tempo disponível e a necessidade dos alunos. Por exemplo, cursos de um semestre podem ser montados baseando-se em uma ou mais das cinco partes; cursos que ofereçam uma visão geral da engenharia de software selecionariam capítulos de todas as cinco

partes; outro, sobre engenharia de software que enfatize análise e projeto, utilizaria tópicos das Partes 1 e 2; cursos de engenharia de software voltado para testes enfatizaria tópicos das Partes 1 e 3, com uma breve incursão na Parte 2; já "cursos voltados para administradores" concentrariam-se nas Partes 1 e 4. Organizada dessa maneira, a sétima edição oferece ao professor uma série de opções didáticas. Esse conteúdo é complementado pelos seguintes elementos.

Recursos para os estudantes. A ampla gama de instrumentos para os estudantes inclui um completo centro de aprendizagem on-line englobando guias de estudo capítulo por capítulo, testes práticos, soluções de problemas e uma série de recursos baseados na Web, incluindo listas de controle de engenharia de software, um conjunto em constante evolução de "pequenas ferramentas", um estudo de caso completo, gabaritos de artefatos e muitos outros. Há, ainda, mais de 1.000 *Referências na Web*, classificadas por categoria, que permitem ao estudante explorar a engenharia de software de forma mais detalhada, e uma *Biblioteca de Referências* com links de mais de 500 artigos, os quais fornecem uma extensa fonte de informações avançadas sobre engenharia de software.

Recursos para os professores. Desenvolveu-se uma ampla gama de recursos para os professores para suplementar a sétima edição. Entre eles: um completo *Guia do Instrutor on-line* (que também pode ser baixado) e materiais didáticos, como, por exemplo, um conjunto completo com mais de 700 slides em *PowerPoint* que pode ser usado em aulas, além de uma série de testes. Também estão disponíveis todos os recursos dirigidos aos estudantes (como as pequenas ferramentas, as Referências na Web, a Biblioteca de Referências que pode ser baixada) e os recursos para profissionais.

O *Guia do Instrutor* apresenta sugestões para a realização de vários tipos de cursos de engenharia de software, recomendações para uma variedade de projetos de software a ser realizados com um curso, soluções para problemas escolhidos e uma série de ferramentas didáticas úteis.

Recursos para os profissionais. No conjunto de recursos disponíveis para profissionais da área (bem como para estudantes e corpo docente), temos resumos e exemplos de documentos de engenharia de software e outros artefatos, um útil conjunto de listas de controle de engenharia de software, um catálogo de ferramentas de engenharia de software (CASE), uma completa coleção de recursos baseados na Web e um "modelo de processos adaptáveis", que compõe de forma detalhada as tarefas do processo de engenharia de software.

Quando associada ao seu sistema de suporte on-line, esta edição gera flexibilidade e profundidade do conteúdo que não poderiam ser atingidas por um livro-texto isolado.

Agradecimentos. Meu trabalho nas sete edições de *Engenharia de software: uma abordagem profissional* tem sido o mais longo e contínuo projeto técnico de minha vida. Mesmo quando termino a atividade de redação, informações extraídas da literatura técnica continuam a ser assimiladas e organizadas, bem como sugestões e críticas de leitores ao redor do mundo são avaliadas e catalogadas. Por essa razão, meus agradecimentos aos muitos autores de livros e artigos acadêmicos (tanto em papel quanto em meio eletrônico) que me forneceram visões, ideias e comentários adicionais ao longo de aproximadamente 30 anos.

Agradecimentos especiais a Tim Lethbridge, da University of Ottawa, que me ajudou no desenvolvimento de exemplos em UML e OCL e a desenvolver o estudo de caso que acompanha este livro, bem como Dale Skrien do Colby College, que desenvolveu o tutorial UML do Apêndice 1. Sua ajuda e comentários foram inestimáveis. Um agradecimento especial a Bruce Maxim, da University of Michigan-Dearborn, que me ajudou a desenvolver grande parte do conteúdo do site pedagógico que acompanha a edição em inglês. Por fim, gostaria de agradecer aos revisores da sétima edição: seus comentários profundos e críticas inteligentes foram valiosas.

Osman Balci,
Virginia Tech University
Max Fomitchev,
Penn State University
Jerry (Zeyu) Gao,
San Jose State University
Guillermo Garcia,
Universidad Alfonso X Madrid
Pablo Gervas,
Universidad Complutense de Madrid

SK Jain,
National Institute of Technology Hamirpur
Saeed Monemi,
Cal Poly Pomona
Ahmed Salem,
California State University
Vasudeva Varma,
IIIT Hyderabad

O conteúdo da sétima edição de *Engenharia de software: uma abordagem profissional* foi moldado por profissionais da área, professores universitários e estudantes que usaram edições anteriores do livro e se deram ao trabalho de enviar sugestões, críticas e ideias. Meus agradecimentos a cada um de vocês. Além disso, meus agradecimentos pessoais vão a muitas pessoas da indústria, distribuídas por todo o mundo, que me ensinaram tanto ou mais do que eu poderia ensinar-lhes.

Assim como as edições deste livro evoluíram, meus filhos, Matheus e Michael, cresceram e se tornaram homens. Sua maturidade, caráter e sucesso me inspiraram. Nada mais me deixou tão orgulhoso. E, finalmente, para você, Barbara, meu amor, agradeço por ter tolerado as várias horas que dediquei ao trabalho e por ter me estimulado para mais uma edição do "livro".

Roger S. Pressman

SUMÁRIO RESUMIDO

CAPÍTULO 1	Engenharia de Software	29
PARTE UM	PROCESSOS DE SOFTWARE	51
CAPÍTULO 2	Modelos de Processo	52
CAPÍTULO 3	Desenvolvimento Ágil	81
PARTE DOIS	MODELAGEM	107
CAPÍTULO 4	Princípios que Orientam a Prática	108
CAPÍTULO 5	Engenharia de Requisitos	126
CAPÍTULO 6	Modelagem de Requisitos: Cenários, Informações e Classes de Análise	150
CAPÍTULO 7	Modelagem de Requisitos: Fluxo, Comportamento, Padrões e Aplicações Baseadas na Web (WebApp)	181
CAPÍTULO 8	Conceitos de Projeto	206
CAPÍTULO 9	Projeto de Arquitetura	229
CAPÍTULO 10	Projeto de Componentes	257
CAPÍTULO 11	Projeto de Interfaces do Usuário	287
CAPÍTULO 12	Projeto Baseado em Padrões	316
CAPÍTULO 13	Projeto de WebApps	338
PARTE TRÊS	GESTÃO DA QUALIDADE	357
CAPÍTULO 14	Conceitos de Qualidade	358
CAPÍTULO 15	Técnicas de Revisão	373
CAPÍTULO 16	Garantia da Qualidade de Software	387
CAPÍTULO 17	Estratégias de Teste de Software	401
CAPÍTULO 18	Testando Aplicativos Convencionais	428
CAPÍTULO 19	Testando Aplicações Orientadas a Objeto	453
CAPÍTULO 20	Testando Aplicações para Web	468
CAPÍTULO 21	Modelagem Formal e Verificação	491
CAPÍTULO 22	Gestão de Configuração de Software	514
CAPÍTULO 23	Métricas de Produto	538
PARTE QUATRO	GERENCIAMENTO DE PROJETOS DE SOFTWARE	565
CAPÍTULO 24	Conceitos de Gerenciamento de Projeto	566
CAPÍTULO 25	Métricas de Processo e Projeto	583
CAPÍTULO 26	Estimativas de Projeto de Software	604
CAPÍTULO 27	Cronograma de Projeto	629

CAPÍTULO 28	Gestão de Risco 648
CAPÍTULO 29	Manutenção e Reengenharia 662

PARTE CINCO TÓPICOS AVANÇADOS 681

CAPÍTULO 30	Melhoria do Processo de Software 682
CAPÍTULO 31	Tendências Emergentes na Engenharia de Software 701
CAPÍTULO 32	Comentários Finais 721
APÊNDICE 1	Introdução à UML 722
APÊNDICE 2	Conceitos Orientados a Objeto 744
REFERÊNCIAS	751
ÍNDICE	773

SUMÁRIO

CAPÍTULO 1 ENGENHARIA DE SOFTWARE 29

- 1.1 A Natureza do Software 31
 - 1.1.1 Definindo software 32
 - 1.1.2 Campos de aplicação de software 34
 - 1.1.3 Software legado 36
- 1.2 A Natureza Única das Webapps 37
- 1.3 Engenharia de Software 38
- 1.4 O Processo de Software 40
- 1.5 A Prática da Engenharia de Software 42
 - 1.5.1 A essência da prática 42
 - 1.5.2 Princípios gerais 44
- 1.6 Mitos Relativos ao Software 45
- 1.7 Como Tudo Começou 47
- 1.8 Resumo 48
- Problemas e Pontos a Ponderar 49
- Leituras e Fontes de Informação Complementares 49

PARTE UM PROCESSOS DE SOFTWARE 51

CAPÍTULO 2 MODELOS DE PROCESSO 52

- 2.1 Um Modelo de Processo Genérico 53
 - 2.1.1 Definindo atividade metodológica 55
 - 2.1.2 Identificação de um conjunto de tarefas 55
 - 2.1.3 Padrões de processos 55
- 2.2 Avaliação e Aperfeiçoamento de Processos 58
- 2.3 Modelos de Processo Prescritivo 58
 - 2.3.1 O modelo cascata 59
 - 2.3.2 Modelos de processo incremental 61
 - 2.3.3 Modelos de processo evolucionário 62
 - 2.3.4 Modelos concorrentes 67
 - 2.3.5 Um comentário final sobre processos evolucionários 68
- 2.4 Modelos de Processo Especializado 69
 - 2.4.1 Desenvolvimento baseado em componentes 69
 - 2.4.2 O modelo de métodos formais 69
 - 2.4.3 Desenvolvimento de software orientado a aspectos 70
- 2.5 O Processo Unificado 71
 - 2.5.1 Breve histórico 72
 - 2.5.2 Fases do processo unificado 72
- 2.6 Modelos de Processo Pessoal e de Equipe 74
 - 2.6.1 Processo de Software Pessoal (PSP) 74
 - 2.6.2 Processo de Software em Equipe (TSP) 75
- 2.7 Tecnologia de Processos 76
- 2.8 Processo do Produto 77
- 2.9 Resumo 78
- Problemas e Pontos a Ponderar 78
- Leituras e Fontes de Informação Complementares 79

CAPÍTULO 3 DESENVOLVIMENTO ÁGIL 81

-
- 3.1 O que é Agilidade? 82
 - 3.2 Agilidade e o Custo das Mudanças 83
 - 3.3 O que é Processo Ágil? 84
 - 3.3.1 Princípios da agilidade 84
 - 3.3.2 A política do desenvolvimento ágil 85
 - 3.3.3 Fatores humanos 86
 - 3.4 Extreme Programming – XP (Programação Extrema) 87
 - 3.4.1 Valores da XP 87
 - 3.4.2 O processo XP 88
 - 3.4.3 Industrial XP 91
 - 3.4.4 O debate XP 92
 - 3.5 Outros Modelos de Processos Ágeis 93
 - 3.5.1 Desenvolvimento de Software Adaptativo (ASD) 94
 - 3.5.2 Scrum 95
 - 3.5.3 Método de Desenvolvimento de Sistemas Dinâmicos (DSDM) 96
 - 3.5.4 Crystal 97
 - 3.5.5 Desenvolvimento Dirigido a Funcionalidades (FDD) 98
 - 3.5.6 Desenvolvimento de Software Enxuto (ISD) 99
 - 3.5.7 Modelagem Ágil (AM) 99
 - 3.5.8 Processo Unificado Ágil (AUP) 101
 - 3.6 Um Conjunto de Ferramentas para o Processo Ágil 102
 - 3.7 Resumo 102
 - Problemas e Pontos a Ponderar 103
 - Leituras e Fontes de Informação Complementares 104

PARTE DOIS**MODELAGEM 107****CAPÍTULO 4 PRINCÍPIOS QUE ORIENTAM A PRÁTICA 108**

-
- 4.1 Conhecimento da Engenharia de Software 109
 - 4.2 Princípios Fundamentais 109
 - 4.2.1 Princípios que orientam o processo 110
 - 4.2.2 Princípios que orientam a prática 111
 - 4.3 Princípios das Atividades Metodológicas 112
 - 4.3.1 Princípios da comunicação 112
 - 4.3.2 Princípios de planejamento 114
 - 4.3.3 Princípios de modelagem 116
 - 4.3.4 Princípios de construção 120
 - 4.3.5 Princípios de disponibilização 122
 - 4.4 Resumo 123
 - Problemas e Pontos a Ponderar 123
 - Leituras e Fontes de Informação Complementares 124

CAPÍTULO 5 ENGENHARIA DE REQUISITOS 126

-
- 5.1 Engenharia de Requisitos 127
 - 5.2 Início do Processo de Engenharia de Requisitos 131
 - 5.2.1 Identificação de interessados 131
 - 5.2.2 Reconhecimento de diversos pontos de vista 131
 - 5.2.3 Trabalho na busca da colaboração 132
 - 5.2.4 Perguntas iniciais 132

5.3	Levantamento de Requisitos 133
5.3.1	Coleta colaborativa de requisitos 133
5.3.2	Disponibilização da função de qualidade 136
5.3.3	Cenários de uso 136
5.3.4	Artefatos do levantamento de requisitos 137
5.4	Desenvolvimento de Casos de Uso 137
5.5	Construção do modelo de análise 142
5.5.1	Elementos do modelo de análise 142
5.5.2	Padrões de análise 145
5.6	Negociação de Requisitos 145
5.7	Validação dos Requisitos 146
5.8	Resumo 147
	Problemas e Pontos a Ponderar 147
	Leituras e Fontes de Informação Complementares 148

CAPÍTULO 6 MODELAGEM DE REQUISITOS: CENÁRIOS, INFORMAÇÕES E CLASSES DE ANÁLISE 150

6.1	Ánalise de Requisitos 151
6.1.1	Filosofia e objetivos gerais 152
6.1.2	Regras práticas para a análise 152
6.1.3	Análise de domínio 153
6.1.4	Abordagens à modelagem de requisitos 154
6.2	Modelagem Baseada em Cenários 155
6.2.1	Criação de um caso de uso preliminar 155
6.2.2	Refinamento de um caso de uso preliminar 158
6.2.3	Criação de um caso de uso formal 159
6.3	Modelos UML que Complementam o Caso de Uso 161
6.3.1	Desenvolvimento de um diagrama de atividade 161
6.3.2	Diagramas de raias 162
6.4	Conceitos da Modelagem de Dados 163
6.4.1	Objetos de dados 163
6.4.2	Atributos de dados 163
6.4.3	Relacionamentos 164
6.5	Modelagem Baseada em Classes 166
6.5.1	Identificação de classes de análise 166
6.5.2	Especificação de atributos 169
6.5.3	Definição das operações 170
6.5.4	Modelagem CRC (Classe-Responsabilidade-Colaborador) 171
6.5.5	Associações e dependências 176
6.5.6	Pacotes de análise 178
6.6	Resumo 178
	Problemas e Pontos a Ponderar 179
	Leituras e Fontes de Informação Complementares 180

CAPÍTULO 7 MODELAGEM DE REQUISITOS: FLUXO, COMPORTAMENTO, PADRÓES E APLICAÇÕES BASEADAS NA WEB (WEBAPP) 181

7.1	Estratégias de Modelagem de Requisitos 182
7.2	Modelagem Orientada a Fluxos 182
7.2.1	Criação de um modelo de fluxo de dados 182
7.2.2	Criação de um modelo de fluxo de controle 185
7.2.3	A especificação de controles 185
7.2.4	A especificação de processos 186

7.3	Criação de um Modelo Comportamental	188
7.3.1	Identificação de eventos com o caso de uso	189
7.3.2	Representações de estados	189
7.4	Padrões Para a Modelagem de Requisitos	192
7.4.1	Descoberta de padrões de análise	192
7.4.2	Exemplo de padrão de requisitos: atuador-sensor	193
7.5	Modelagem de Requisitos para WebApps	197
7.5.1	Que nível de análise é suficiente?	197
7.5.2	Entrada da modelagem de requisitos	197
7.5.3	Saída da modelagem de requisitos	198
7.5.4	Modelo de conteúdo para WebApps	199
7.5.5	Modelo de interações para WebApps	200
7.5.6	Modelo funcional para WebApps	200
7.5.7	Modelos de configuração para WebApps	201
7.5.8	Modelo de navegação	202
7.6	Resumo	203
	Problemas e Pontos a Ponderar	204
	Leituras e Fontes de Informação Complementares	204

CAPÍTULO 8 CONCEITOS DE PROJETO 206

8.1	Projeto no Contexto da Engenharia de Software	207
8.2	O Processo de Projeto	209
8.2.1	Diretrizes e atributos da qualidade de software	209
8.2.2	A evolução de um projeto de software	211
8.3	Conceitos de Projeto	212
8.3.1	Abstração	212
8.3.2	Arquitetura	213
8.3.3	Padrões	214
8.3.4	Separação por interesses (por afinidades)	214
8.3.5	Modularidade	214
8.3.6	Encapsulamento de informações	215
8.3.7	Independência funcional	216
8.3.8	Refinamento	217
8.3.9	Aspectos	217
8.3.10	Refatoração	218
8.3.11	Conceitos de projeto orientado a objetos	218
8.3.12	Classes de projeto	218
8.4	O Modelo de Projeto	221
8.4.1	Elementos de projeto de dados	222
8.4.2	Elementos de projeto de arquitetura	222
8.4.3	Elementos de projeto de interfaces	222
8.4.4	Elementos de projeto de componentes	224
8.4.5	Elementos de projeto de implantação	224
8.5	Resumo	226
	Problemas e Pontos a Ponderar	226
	Leituras e Fontes de Informação Complementares	227

CAPÍTULO 9 PROJETO DA ARQUITETURA 229

9.1	Arquitetura de Software	230
9.1.1	O que é arquitetura?	230
9.1.2	Por que a arquitetura é importante?	231

9.1.3	Descrições de arquitetura	231
9.1.4	Decisões de arquitetura	232
9.2	Gêneros de Arquitetura	232
9.3	Estilos de Arquitetura	234
9.3.1	Uma breve taxonomia das estilos de arquitetura	235
9.3.2	Padrões de arquitetura	238
9.3.3	Organização e refinamento	239
9.4	Projeto da Arquitetura	239
9.4.1	Representação do sistema no contexto	240
9.4.2	Definição de arquétipos	241
9.4.3	Refinamento da arquitetura em componentes	242
9.4.4	Descrição das instâncias	243
9.5	Avaliação de Projetos de Arquiteturas Alternativas	244
9.5.1	Um método de análise dos prós e contras de uma arquitetura	245
9.5.2	Complexidade da arquitetura	246
9.5.3	Linguagens de descrição da arquitetura	247
9.6	Mapeamento de Arquitetura Utilizando Fluxo de Dados	247
9.6.1	Mapeamento de transformação	248
9.6.2	Refinamento do projeto da arquitetura	254
9.7	Resumo	255
	Problemas e Pontos a Ponderar	255
	Leituras e Fontes de Informação Complementares	256

CAPÍTULO 10 PROJETO DE COMPONENTES 257

10.1	O Que é Componente?	258
10.1.1	Uma visão orientada a objetos	258
10.1.2	A visão tradicional	260
10.1.3	Uma visão relacionada com processos	262
10.2	Projeto de Componentes Baseados em Classes	262
10.2.1	Princípios básicos de projeto	262
10.2.2	Diretrizes para o projeto de componentes	265
10.2.3	Coesão	266
10.2.4	Acoplamento	267
10.3	Condução de Projetos de Componentes	269
10.4	Projeto de Componentes para WebApps	274
10.4.1	Projeto de conteúdo para componentes	274
10.4.2	Projeto funcional para componentes	275
10.5	Projeto de Componentes Tradicionais	275
10.5.1	Notação gráfica em projeto	276
10.5.2	Notação tabular de projeto	277
10.5.3	Linguagem de projeto de programas	278
10.6	Desenvolvimento Baseado em Componentes	279
10.6.1	Engenharia de domínio	280
10.6.2	Qualificação, adaptação e composição de componentes	280
10.6.3	Análise e projeto para reutilização	282
10.6.4	Classificação e recuperação de componentes	283
10.7	Resumo	284
	Problemas e Pontos a Ponderar	285
	Leituras e Fontes de Informação Complementares	286

CAPÍTULO 11 PROJETO DE INTERFACES DO USUÁRIO 287

- 11.1 As Regras de Ouro 288
 - 11.1.1 Deixar o usuário no comando 288
 - 11.1.2 Reduzir a carga de memória do usuário 289
 - 11.1.3 Tornar a interface consistente 290
- 11.2 Análise e Projeto de Interfaces 291
 - 11.2.1 Modelos de análise e projeto de interfaces 291
 - 11.2.2 O processo 292
- 11.3 Análise de Interfaces 294
 - 11.3.1 Análise de usuários 294
 - 11.3.2 Análise e modelagem de tarefas 295
 - 11.3.3 Análise do conteúdo exibido 299
 - 11.3.4 Análise do ambiente de trabalho 300
- 11.4 Etapas no Projeto de Interfaces 300
 - 11.4.1 Aplicação das etapas para projeto de interfaces 301
 - 11.4.2 Padrões de projeto de interfaces do usuário 302
 - 11.4.3 Questões de projeto 303
- 11.5 Projeto de Interfaces para WebApp 306
 - 11.5.1 Princípios e diretrizes para projeto de interfaces 306
 - 11.5.2 Fluxo de trabalho de projeto de interfaces para WebApps 310
- 11.6 Avaliação de Projeto 311
- 11.7 Resumo 313
- Problemas e Pontos a Ponderar 313
- Leituras e Fontes de Informação Complementares 314

CAPÍTULO 12 PROJETO BASEADO EM PADRÓES 316

- 12.1 Padrão de Projeto 317
 - 12.1.1 Tipos de padrões 318
 - 12.1.2 Estruturas de uso de padrões de projeto 320
 - 12.1.3 Descrição de padrões 320
 - 12.1.4 Linguagens e repositórios de padrões 321
- 12.2 Projeto de Software Baseado em Padrões 322
 - 12.2.1 Contexto do projeto baseado em padrões 322
 - 12.2.2 Pensando em termos de padrões 323
 - 12.2.3 Tarefas de projeto 324
 - 12.2.4 Construção de uma tabela para organização de padrões 325
 - 12.2.5 Erros comuns de projeto 326
- 12.3 Padrões de Arquitetura 327
- 12.4 Padrões de Projeto de Componentes 329
- 12.5 Padrões de Projeto para Interfaces do Usuário 331
- 12.6 Padrões de Projeto para WebApps 333
 - 12.6.1 Foco do projeto 333
 - 12.6.2 Granularidade do projeto 334
- 12.7 Resumo 335
- Problemas e Pontos a Ponderar 336
- Leituras e Fontes de Informação Complementares 336

CAPÍTULO 13 PROJETO DE WEBAPPS 338

- 13.1 Qualidade de Projeto em WebApps 339
- 13.2 Objetivos de Projeto 341
- 13.3 Uma Pirâmide de Projeto para WebApps 342

13.4	Projeto de Interfaces para WebApps	342
13.5	Projeto Estético	344
13.5.1	Problemas de layout	344
13.5.2	Questões de design gráfico	344
13.6	Projeto de Conteúdo	345
13.6.1	Objetos de conteúdo	345
13.6.2	Questões de projeto de conteúdo	345
13.7	Projeto Arquitetural	346
13.7.1	Arquitetura de conteúdo	347
13.7.2	Arquitetura de uma WebApp	349
13.8	Projeto da Navegação	350
13.8.1	Semântica de navegação	350
13.8.2	Sintaxe de navegação	351
13.9	Projeto dos Componentes	352
13.10	Método do Projeto de Hipermídia Orientada a Objetos (<i>Object-Oriented Hypermedia Design Method – OOHDMD</i>)	352
13.10.1	Projeto conceitual para o OOHDMD	352
13.10.2	Projeto da navegação para o OOHDMD	353
13.10.3	Projeto da interface abstrata e implementação	353
13.11	Resumo	354
	Problemas e Pontos a Ponderar	355
	Leituras e Fontes de Informação Complementares	356

PARTE TRÊS**GESTÃO DE QUALIDADE 357****CAPÍTULO 14 CONCEITOS DE QUALIDADE 358**

14.1	O Que é Qualidade?	359
14.2	Qualidade de Software	360
14.2.1	Dimensões de qualidade de Garvin	360
14.2.2	Fatores de qualidade de McCall	361
14.2.3	Fatores de qualidade ISO 9126	362
14.2.4	Fatores de qualidade desejados	363
14.2.5	A transição para uma visão quantitativa	364
14.3	O Dilema da Qualidade de Software	365
14.3.1	Software "bom o suficiente"	365
14.3.2	Custo da qualidade	366
14.3.3	Riscos	368
14.3.4	Negligência e responsabilidade civil	368
14.3.5	Qualidade e segurança	368
14.3.6	O impacto das ações administrativas	369
14.4	Alcançando a Qualidade do Software	370
14.4.1	Métodos de engenharia de software	370
14.4.2	Técnicas de gerenciamento de software	370
14.4.3	Controle de qualidade	370
14.4.4	Garantia da qualidade	370
14.5	Resumo	371
	Problemas e Pontos a Ponderar	371
	Leituras e Fontes de Informação Complementares	372

CAPÍTULO 15 TÉCNICAS DE REVISÃO 373

15.1	Impacto de Defeitos de Software nos Custos	374
15.2	Amplificação e Eliminação de Defeitos	375

15.3	Métricas de Revisão e seu Emprego	376
15.3.1	Análise de métricas	377
15.3.2	Eficácia das custas de revisões	377
15.4	Revisões: Um Espectro de Formalidade	379
15.5	Revisões Informais	380
15.6	Revisões Técnicas Formais	381
15.6.1	Uma reunião de revisão	381
15.6.2	Relatório de revisão e manutenção de registros	382
15.6.3	Diretrizes de revisão	382
15.6.4	Revisões por amostragem	384
15.7	Resumo	385
	Problemas e Pontos a Ponderar	385
	Leituras e Fontes de Informação Complementares	386

CAPÍTULO 16 GARANTIA DA QUALIDADE DE SOFTWARE 387

16.1	Problemas de Background	388
16.2	Elementos de Garantia da Qualidade de Software	388
16.3	Tarefas, Metas e Métricas da SQA	390
16.3.1	Tarefas da SQA	390
16.3.2	Metas, atributos e métricas	391
16.4	Abordagens Formais da SQA	392
16.5	Estatística da Garantia da Qualidade de Software	393
16.5.1	Um exemplo genérico	393
16.5.2	Seis sigma para engenharia de software	394
16.6	Confiabilidade do Software	395
16.6.1	Medidas de confiabilidade e disponibilidade	395
16.6.2	Proteção do software	396
16.7	Os Padrões de Qualidade Iso 9000	397
16.8	O Plano de Sqa	398
16.9	Resumo	399
	Problemas e Pontos a Ponderar	399
	Leituras e Fontes de Informação Complementares	400

CAPÍTULO 17 ESTRATÉGIAS DE TESTE DE SOFTWARE 401

17.1	Uma Abordagem Estratégica do Teste de Software	402
17.1.1	Verificação e validação	402
17.1.2	Organizando o teste de software	403
17.1.3	Estratégia de teste de software — a visão ampla	404
17.1.4	Critérios para conclusão do teste	405
17.2	Problemas Estratégicos	406
17.3	Estratégias Para Software Convencional	407
17.3.1	Teste de unidade	407
17.3.2	Teste de integração	409
17.4	Estratégias de Teste Para Software Orientado a Objeto	415
17.4.1	Teste de unidade no contexto OO	415
17.4.2	Teste de integração no contexto OO	415
17.5	Estratégias de Teste Para WebApps	416
17.6	Teste de Validação	416
17.6.1	Critério de teste de validação	417
17.6.2	Revisão da configuração	417
17.6.3	Teste alfa e beta	417

17.7	Teste de Sistema 418
17.7.1	Teste de recuperação 419
17.7.2	Teste de segurança 419
17.7.3	Teste por esforço 419
17.7.4	Teste de desempenho 420
17.7.5	Teste de disponibilização 420
17.8	A Arte da Depuração 421
17.8.1	O processo de depuração 421
17.8.2	Considerações psicológicas 422
17.8.3	Estratégias de depuração 423
17.8.4	Correção do erro 424
17.9	Resumo 425
	Problemas e Pontos a Ponderar 425
	Leituras e Fontes de Informação Complementares 426

CAPÍTULO 18 TESTANDO APlicativos CONVENCIONAIS 428

18.1	Fundamentos do Teste de Software 429
18.2	Visões Interna e Externa do Teste 430
18.3	Teste Caixa Branca 431
18.4	Teste do Caminho Básico 431
18.4.1	Notação de grafo de fluxo 432
18.4.2	Caminhos de programa independentes 433
18.4.3	Derivação de casos de teste 435
18.4.4	Matrizes de grafos 436
18.5	Teste de Estrutura de Controle 437
18.5.1	Teste de condição 437
18.5.2	Teste de fluxo de dados 438
18.5.3	Teste de ciclo 438
18.6	Teste Caixa Preta 439
18.6.1	Métodos de teste com base em grafo 440
18.6.2	Particionamento de equivalência 441
18.6.3	Análise de valor limite 442
18.6.4	Teste de matriz ortogonal 442
18.7	Teste Baseado em Modelos 445
18.8	Teste Para Ambientes, Arquiteturas e Aplicações Especializados 446
18.8.1	Testando GUIs 446
18.8.2	Teste de arquiteturas cliente-servidor 446
18.8.3	Testando a documentação e os recursos de ajuda 447
18.8.4	Teste para sistema em tempo real 448
18.9	Padrões para Teste de Software 449
18.10	Resumo 450
	Problemas e Pontos a Ponderar 451
	Leituras e Fontes de Informação Complementares 452

CAPÍTULO 19 TESTANDO APlicações ORIENTADAS A OBJETO 453

19.1	Ampliando a Versão do Teste 454
19.2	Testando Modelos de Análise Orientada a Objeto (OOA) e Projeto Orientado a Objeto (OOD) 455
19.2.1	Exatidão dos modelos de OOA e OODs 455
19.2.2	Consistência dos modelos orientados a objeto 455
19.3	Estratégias de Teste Orientado a Objeto 457
19.3.1	Teste de unidade em contexto orientado a objeto 457

19.3.2	Teste de integração em contexto orientado a objeto	457
19.3.3	Teste de validação em contexto orientado a objeto	458
19.4	Métodos de Teste Orientados a Objeto	458
19.4.1	A implicações no projeto de casos de teste dos conceitos orientados a objeto	459
19.4.2	Aplicabilidade dos métodos convencionais de projeto de casos de teste	459
19.4.3	Teste baseado em falhas	459
19.4.4	Casos de teste e a hierarquia de classe	460
19.4.5	Projeto de teste baseado em cenário	460
19.4.6	Teste da estrutura superficial e estrutura profunda	462
19.5	Métodos de Teste Aplicáveis no Nível de Classe	462
19.5.1	Teste aleatório para classes orientadas a objeto	462
19.5.2	Teste de partição em nível de classe	463
19.6	Projeto de Caso de Teste Interclasse	464
19.6.1	Teste de múltiplas classes	464
19.6.2	Testes derivados de modelos comportamentais	465
19.7	Resumo	466
	Problemas e Pontos a Ponderar	467
	Leituras e Fontes de Informação Complementares	467

CAPÍTULO 20 TESTANDO APLICAÇÕES PARA WEB 468

20.1	Conceitos de Teste para WebApps	469
20.1.1	Dimensões da qualidade	469
20.1.2	Erros em um ambiente WebApp	469
20.1.3	Estratégia de teste	470
20.1.4	Planejamento de teste	470
20.2	O Processo de Teste – uma Visão Geral	471
20.3	Teste de Conteúdo	472
20.3.1	Objetivos do teste de conteúdo	472
20.3.2	Teste de base de dados	473
20.4	Teste da Interface de Usuário	474
20.4.1	Estratégia de teste de interface	474
20.4.2	Testando mecanismos de interface	475
20.4.3	Testando semânticas de interface	477
20.4.4	Testes de usabilidade	477
20.4.5	Testes de compatibilidade	478
20.5	Teste no Nível de Componente	479
20.6	Testes de Navegação	480
20.6.1	Testando a sintaxe de navegação	481
20.6.2	Testando as semânticas de navegação	481
20.7	Teste de Configuração	482
20.7.1	Tópicos no lado do servidor	483
20.7.2	Tópicos no lado do cliente	483
20.8	Teste de Segurança	484
20.9	Teste de Desempenho	485
20.9.1	Objetivos do teste de desempenho	485
20.9.2	Teste de carga	486
20.9.3	Teste de esforço (stress)	486
20.10	Resumo	487
	Problemas e Pontos a Ponderar	488
	Leituras e Fontes de Informação Complementares	489

CAPÍTULO 21 MODELAGEM FORMAL E VERIFICAÇÃO 491

- 21.1 Estratégia Sala Limpa 492
- 21.2 Especificação Funcional 493
 - 21.2.1 Especificação caixa preta 495
 - 21.2.2 Especificação caixa de estado 495
 - 21.2.3 Especificação caixa clara 495
- 21.3 Projeto Sala Limpa 496
 - 21.3.1 Refinamento de projeto 496
 - 21.3.2 Verificação de projeto 497
- 21.4 Teste Sala Limpa 498
 - 21.4.1 Teste estatístico de uso 498
 - 21.4.2 Certificação 499
- 21.5 Conceitos de Métodos Formais 500
- 21.6 Aplicando Notação Matemática para Especificação Formal 503
- 21.7 Linguagens de Especificação Formal 504
 - 21.7.1 Object Constraint Language (OCL) 505
 - 21.7.2 A linguagem de especificação Z 508
- 21.8 Resumo 510
- Problemas e Pontos a Ponderar 511
- Leituras e Fontes de Informação Complementares 512

CAPÍTULO 22 GESTÃO DE CONFIGURAÇÃO DE SOFTWARE 514

- 22.1 Gestão de Configuração de Software 515
 - 22.1.1 Um cenário SCM 515
 - 22.1.2 Elementos de um sistema de gestão de configuração 516
 - 22.1.3 Referenciais 517
 - 22.1.4 Itens de configuração de software 518
- 22.2 O Repositório SCM 519
 - 22.2.1 O papel do repositório 519
 - 22.2.2 Características gerais e conteúdo 519
 - 22.2.3 Características SCM 520
- 22.3 O Processo SCM 521
 - 22.3.1 Identificação de objetos na configuração de software 522
 - 22.3.2 Controle de versão 523
 - 22.3.3 Controle de alterações 524
 - 22.3.4 Auditoria de configuração 526
 - 22.3.5 Relatório de status 527
- 22.4 Gestão de Configuração para WebApps 528
 - 22.4.1 Problemas dominantes 528
 - 22.4.2 Objetos de configuração de WebApp 529
 - 22.4.3 Gestão de conteúdo 530
 - 22.4.4 Gestão de alterações 531
 - 22.4.5 Controle de versão 534
 - 22.4.6 Auditoria e relatório 534
- 22.5 Resumo 535
- Problemas e Pontos a Ponderar 536
- Leituras e Fontes de Informação Complementares 537

CAPÍTULO 23 MÉTRICAS DE PRODUTO 538

- 23.1 Estrutura para Métricas de Produto 539
 - 23.1.1 Medidas, métricas e indicadores 539

23.1.2	O desafio das métricas de produto	539
23.1.3	Princípios de medição	540
23.1.4	Medição de software orientada a objetivo	541
23.1.5	Atributos de métricas efetivas de software	542
23.2	Métricas para o Modelo de Requisitos	543
23.2.1	Métricas baseadas em função	543
23.2.2	Métricas para qualidade de especificação	546
23.3	Métricas para o Modelo de Projeto	547
23.3.1	Métricas de projeto da arquitetura	547
23.3.2	Métricas para projeto orientado a objeto	549
23.3.3	Métricas orientadas a classe — o conjunto de métricas CK	551
23.3.4	Métricas orientadas a classe — o conjunto de métricas MOOD	553
23.3.5	Métricas orientadas a objeto propostas por Lorenz e Kidd	554
23.3.6	Métricas de projeto em nível de componente	554
23.3.7	Métricas orientadas a operação	556
23.3.8	Métricas de projeto de interface de usuário	557
23.4	Métricas de Projeto para WebApps	557
23.5	Métricas para Código-Fonte	559
23.6	Métricas para Teste	560
23.6.1	Métricas de Halstead aplicadas ao teste	561
23.6.2	Métricas para teste orientado a objeto	561
23.7	Métricas para Manutenção	562
23.8	Resumo	563
	Problemas e Pontos a Ponderar	563
	Leituras e Fontes de Informação Complementares	564

PARTE QUATRO GERENCIAMENTO DE PROJETOS DE SOFTWARE 565**CAPÍTULO 24 CONCEITOS DE GERENCIAMENTO DE PROJETO 566**

24.1	O Espectro de Gerenciamento	567
24.1.1	Pessoal	567
24.1.2	O produto	567
24.1.3	O processo	568
24.1.4	O projeto	568
24.2	Pessoas	568
24.2.1	Os interessados [comprometidos]	569
24.2.2	Líderes de equipe	569
24.2.3	Equipe de software	570
24.2.4	Equipes ágeis	572
24.2.5	Itens de comunicação e coordenação	573
24.3	O Produto	574
24.3.1	Escopo de software	574
24.3.2	Decomposição do problema	574
24.4	O Processo	575
24.4.1	Combinando o produto e o processo	575
24.4.2	Decomposição do processo	576
24.5	O Projeto	577
24.6	O Princípio W ^{HH}	578
24.7	Práticas Vitais	579
24.8	Resumo	579
	Problemas e Pontos a Ponderar	580
	Leituras e Fontes de Informação Complementares	581

CAPÍTULO 25 MÉTRICAS DE PROCESSO E PROJETO 583

- 25.1 Métricas no domínio de processo e projeto 584
 - 25.1.1 Métricas de processo e aperfeiçoamento do processo de software 584
 - 25.1.2 Métricas de projeto 586
- 25.2 Medidas de Software 586
 - 25.2.1 Métricas orientadas a tamanho 588
 - 25.2.2 Métricas orientadas a função 589
 - 25.2.3 Reconciliando métricas LOC e FP 589
 - 25.2.4 Métricas orientadas a objeto 591
 - 25.2.5 Métricas orientadas a casos de uso 592
 - 25.2.6 Métricas de projeto WebApp 592
- 25.3 Métricas para Qualidade do Software 594
 - 25.3.1 Medição da qualidade 594
 - 25.3.2 Eficiência na remoção de defeitos 595
- 25.4 Integrando Métricas Dentro do Processo de Software 596
 - 25.4.1 Argumentos favoráveis a métricas de software 597
 - 25.4.2 Estabelecendo uma linha de base 597
 - 25.4.3 Coleta, cálculo e avaliação de métricas 597
- 25.5 Métricas para Pequenas Organizações 598
- 25.6 Estabelecendo um Programa de Métricas de Software 599
- 25.7 Resumo 601
- Problemas e Pontos a Ponderar 601
- Leituras e Fontes de Informação Complementares 602

CAPÍTULO 26 ESTIMATIVAS DE PROJETO DE SOFTWARE 604

- 26.1 Observações e Estimativas 605
- 26.2 O Processo de Planejamento do Projeto 606
- 26.3 Escopo e Viabilidade do Software 606
- 26.4 Recursos 607
 - 26.4.1 Recursos humanos 608
 - 26.4.2 Recursos de software reutilizáveis 608
 - 26.4.3 Recursos de ambiente 608
- 26.5 Estimativa do Projeto de Software 609
- 26.6 Técnicas de Decomposição 610
 - 26.6.1 Dimensionamento do software 610
 - 26.6.2 Estimativa baseada em problema 611
 - 26.6.3 Um exemplo de estimativa baseada em LOC 612
 - 26.6.4 Um exemplo de estimativa baseada em FP 613
 - 26.6.5 Estimativas baseadas em processo 614
 - 26.6.6 Um exemplo de estimativa baseada em processo 615
 - 26.6.7 Estimativa com casos de uso 616
 - 26.6.8 Exemplo de estimativa baseada em caso de uso 617
 - 26.6.9 Reconciliando estimativas 617
- 26.7 Modelos Empíricos de Estimativa 618
 - 26.7.1 Estrutura dos modelos de estimativa 619
 - 26.7.2 O modelo COCOMO II 619
 - 26.7.3 A equação do software 621
- 26.8 Estimativa para Projetos Orientados a Objeto 622
- 26.9 Técnicas Especializadas de Estimativa 622
 - 26.9.1 Estimativa para desenvolvimento ágil 622
 - 26.9.2 Estimativa para projetos para WebApp 623

26.10 A Decisão Fazer/Comprar	624
26.10.1 Criando uma árvore de decisões	624
26.10.2 Terceirização	625
26.11 Resumo	627
Problemas e Pontos a Ponderar	627
Leituras e Fontes de Informação Complementares	628

CAPÍTULO 27 CRONOGRAMA DE PROJETO 629

27.1 Conceitos Básicos	630
27.2 Cronograma de Projeto	631
27.2.1 Princípios básicos	632
27.2.2 Relação entre pessoas e esforço	632
27.2.3 Distribuição de esforço	634
27.3 Definindo um Conjunto de Tarefas para o Projeto de Software	635
27.3.1 Exemplo de conjunto de tarefas	635
27.3.2 Refinamento das ações de engenharia de software	636
27.4 Definindo uma Rede de Tarefas	636
27.5 Cronograma	637
27.5.1 Gráfico de Gantt	638
27.5.2 Acompanhando o cronograma	638
27.5.3 Acompanhando o progresso de um projeto orientado a objeto	640
27.5.4 Cronograma para projetos para WebApp	641
27.6 Análise de Valor Agregado	643
27.8 Resumo	645
Problemas e Pontos a Ponderar	645
Leituras e Fontes de Informação Complementares	646

CAPÍTULO 28 GESTÃO DE RISCOS 648

28.1 Estratégias de Riscos Reativa versus Proativa	649
28.2 Riscos de Software	649
28.3 Identificação do Risco	650
28.3.1 Avaliando o risco geral do projeto	651
28.3.2 Componentes e fatores de risco	651
28.4 Previsão de Risco	652
28.4.1 Desenvolvendo uma tabela de risco	653
28.4.2 Avaliando o impacto do risco	655
28.5 Refinamento do Risco	656
28.6 Mitigação, Monitoração e Controle de Riscos (RMMM)	657
28.7 O Plano RMMM	658
28.8 Resumo	660
Problemas e Pontos a Ponderar	660
Leituras e Fontes de Informação Complementares	661

CAPÍTULO 29 MANUTENÇÃO E REENGRENHARIA 662

29.1 Manutenção de Software	663
29.2 Superdinilidade do Software	664
29.3 Reengenharia	665
29.4 Reengenharia de Processo de Negócio	665
29.4.1 Processos de negócio	665
29.4.2 Um modelo de BPR	666

29.5	Reengenharia de Software 667
29.5.1	Um modelo de processo de reengenharia de software 668
29.5.2	Atividades de reengenharia de software 669
29.6	Engenharia Reversa 670
29.6.1	Engenharia reversa para entender os dados 672
29.6.2	Engenharia reversa para entender o processamento 672
29.6.3	Engenharia reversa das interfaces de usuário 673
29.7	Reestruturação 674
29.7.1	Reestruturação do código 674
29.7.2	Reestruturação de dados 674
29.8	Engenharia Direta 675
29.8.1	Engenharia direta para arquiteturas cliente-servidor 676
29.8.2	Engenharia direta para arquiteturas orientadas a objeto 677
29.9	A Economia da Reengenharia 677
29.10	Resumo 678
	Problemas e Pontos a Ponderar 679
	Leituras e Fonte de Informação Complementares 679

PARTE CINCO TÓPICOS AVANÇADOS 681

CAPÍTULO 30 MELHORIA DO PROCESSO DE SOFTWARE 682

30.1	O Que É SPI? 683
30.1.1	Abordagens para SPI 683
30.1.2	Modelos de maturidade 685
30.1.3	A SPI é para todos? 685
30.2	O Processo de SPI 686
30.2.1	Avaliação e análise de lacunas 686
30.2.2	Educação e treinamento 687
30.2.3	Seleção e justificação 688
30.2.4	Instalação/migração 689
30.2.5	Mensuração 689
30.2.6	Gerenciamento de risco para SPI 689
30.2.7	Fatores críticos de sucesso 690
30.3	A CMMI 691
30.4	A CMM das Pessoas 695
30.5	Outras Estruturas SPI 696
30.6	Retorno sobre Investimento em SPI 697
30.7	Tendências da SPI 698
30.8	Resumo 698
	Problemas e Pontos a Ponderar 699
	Leituras e Fontes de Informação Complementares 699

CAPÍTULO 31 TENDÊNCIAS EMERGENTES NA ENGENHARIA DE SOFTWARE 701

31.1	Evolução da Tecnologia 702
31.2	Observando Tendências na Engenharia de Software 703
31.3	Identificando as "Tendências Leves" 704
31.3.1	Administrando a complexidade 705
31.3.2	Software aberto 706
31.3.3	Requisitos emergentes 707
31.3.4	O mix de talentos 707
31.3.5	Blocos básicos de software 708

31.3.6	Mudando as percepções de "valor"	708
31.3.7	Código aberto	709
31.4	Direções da Tecnologia	710
31.4.1	Tendências de processo	710
31.4.2	O grande desafio	711
31.4.3	Desenvolvimento colaborativo	712
31.4.4	Engenharia de requisitos	713
31.4.5	Desenvolvimento de software controlado por modelo	714
31.4.6	Projeto pós-moderno	714
31.4.7	Desenvolvimento baseado em teste	715
31.5	Tendências Relacionadas com Ferramentas	716
31.5.1	Ferramentas que respondem a tendências leves	717
31.5.2	Ferramentas que lidam com as tendências tecnológicas	718
31.6	Resumo	719
	Problemas e Pontos a Ponderar	719
	Leituras e Fontes de Informação Complementares	720

CAPÍTULO 32 COMENTÁRIOS FINAIS 721

32.1	A Importância do Software – Revisitada	722
32.2	Profissionais e a Maneira como Construem Sistemas	722
32.3	Novos Modos de Representar as Informações	723
32.4	A Visão no Longo Prazo	724
32.5	A Responsabilidade do Engenheiro de Software	725
32.6	Comentário Final	726

APÊNDICE 1 INTRODUÇÃO À UML 727**APÊNDICE 2 CONCEITOS ORIENTADOS AO OBJETO 744****REFERÉNCIAS 751****ÍNDICE 773**

SOFTWARE E ENGENHARIA DE SOFTWARE

1

CONCEITOS-	
-CHAVE	
atividades de apoio	40
campos de aplicação	34
características de software	32
engenharia de software	38
metodologia	40
mitos de software	45
prática	45
princípios	44
processo de software	40
software legado	36
WebApps	37

Ele tinha a aparência clássica de um executivo sênior de uma grande empresa de software — cerca de 40 anos de idade, as têmporas levemente grisalhas, elegante e atlético, olhos penetrantes enquanto falava. Mas o que ele disse me deixou em choque: "O software está morto".

Fiquei surpreso e então sorri. "Você está brincando, não é mesmo? O mundo é comandado pelo software e sua empresa tem lucrado imensamente com isso... Ele não está morto! Está vivo e crescendo."

Balançando a cabeça enfática e negativamente, acrescentou: "Não, ele está morto... Pelo menos da forma que, um dia, o conhecemos".

Assenti e disse: "Continue".

Ele falava batendo na mesa para enfatizar: "A visão da velha escola de software — você o compra, é seu proprietário e é o responsável pelo seu gerenciamento — está chegando ao fim. Atualmente, com a Web 2.0 e a computação pervasiva, que vem surgindo com força, estamos por ver uma geração completamente diferente de software. Ele será distribuído via Internet e parecerá estar residente nos dispositivos do computador de cada usuário... Porém, estará residente em um servidor bem distante".

PANORAMA

O que é? Software de computador é o produto que profissionais de software desenvolvem e ao qual dão suporte no longo prazo. Abrange programas executáveis em um computador de qualquer porte ou arquitetura, conteúdos (apresentados à medida que os programas são executados), informações descritivas tanto na forma impressa (*hard copy*) como na virtual, abrangendo praticamente qualquer mídia eletrônica. A engenharia de software abrange um processo, um conjunto de métodos (práticas) e um leque de ferramentas que possibilitam aos profissionais desenvolverem software de altíssima qualidade.

Quem realiza? Os engenheiros de software criam e dão suporte a ele e, praticamente, todos do mundo industrializado o utilizam, direta ou indiretamente.

Por que ele é importante? Software é importante porque afeta a quase todos os aspectos de nossas vidas e tornou-se perversivo (incorporado) no comércio, na cultura e em nossas atividades cotidianas. A engenharia de software é importante

porque ela nos capacita para o desenvolvimento de sistemas complexos dentro do prazo e com alta qualidade.

Quais são as etapas envolvidas? Cria-se software para computadores da mesma forma que qualquer produto bem-sucedido: aplicando-se um processo adaptável e ágil que conduza a um resultado de alta qualidade, atendendo às necessidades daqueles que usarão o produto. Aplica-se uma abordagem de engenharia de software.

Qual é o artefato? Do ponto de vista de um engenheiro de software, é um conjunto de programas, conteúdo (dados) e outros artefatos que são software. Porém, do ponto de vista do usuário, o artefato consiste em informações resultantes que, de alguma forma, tornam a vida dele melhor.

Como garantir que o trabalho foi feito corretamente? Leia o restante deste livro, selecione as ideias que são aplicáveis ao software que você desenvolver e use-as em seu trabalho.

Eu tive de concordar: "Assim, sua vida será muito mais simples. Vocês, meus amigos, não terão de se preocupar com cinco versões diferentes do mesmo aplicativo em uso por dezenas de milhares de usuários espalhados".

Ele sorriu e acrescentou: "Absolutamente. Apenas a versão mais atual residindo em nossos servidores. Quando fizermos uma modificação ou correção, forneceremos funcionalidade e conteúdo atualizados para todos os usuários. Todos terão isso instantaneamente!".

Fiz uma careta: "Mas, da mesma forma, se houver um erro, todos o terão instantaneamente".

Ele riu: "É verdade, por isso estamos redobrando nossos esforços para aplicarmos a engenharia de software de uma forma ainda melhor. O problema é que temos de fazer isso 'rapidamente', pois o mercado tem se acelerado, em todas as áreas de aplicação".

Recostei-me e, com minhas mãos por trás da cabeça, disse: "Você sabe o que falam por aí: você pode ter algo rápido, você pode ter algo correto ou você pode ter algo barato. Escolha dois!".

"Eu fico com rápido e correto", disse, levantando-se.

Também me levantei, concluindo: "Então, nós realmente precisamos de engenharia de software".

"Sei disso", afirmou, enquanto se afastava. "O problema é: temos de convencer ainda outra geração de 'techies'* de que isso é verdadeiro!".

O software está *realmente* morto? Se estivesse, você não estaria lendo este livro!

Software de computadores continua a ser a tecnologia única mais importante no cenário mundial. E é também um ótimo exemplo da lei das consequências não intencionais. Há cinquenta anos, ninguém poderia prever que o software iria se tornar uma tecnologia indispensável para negócios, ciência e engenharia; que software iria viabilizar a criação de novas tecnologias (por exemplo, engenharia genética e nanotecnologia), a extensão de tecnologias existentes (por exemplo, telecomunicações) e a mudança radical nas tecnologias mais antigas (por exemplo, indústria gráfica); que software se tornaria a força motriz por trás da revolução do computador pessoal; que produtos de pacotes de software seriam comprados pelos consumidores em lojas de bairro; que software evoluiria lentamente de produto para serviço, na medida que empresas de software "sob encomenda" oferecessem funcionalidade imediata (*just-in-time*), via um navegador Web; que uma companhia de software iria se tornar a maior e mais influente do que quase todas as companhias da era industrial; que uma vasta rede comandada por software, denominada Internet, iria evoluir e modificar tudo: de pesquisa em bibliotecas a compras feitas pelos consumidores, incluindo discurso político, hábitos de namoros de jovens e de adultos não tão jovens.

Ninguém poderia prever que o software seria incorporado em sistemas de todas as áreas: transportes, medicina, telecomunicações, militar, industrial, entretenimento, máquinas de escritório... A lista é quase infinitável. E se você acredita na lei das consequências não intencionais, há muitos efeitos que ainda não somos capazes de prever.

Ninguém poderia prever que milhões de programas de computador teriam de ser corrigidos, adaptados e ampliados à medida que o tempo passasse. A realização dessas atividades de "manutenção" absorve mais pessoas e recursos do que todo o esforço aplicado na criação de um novo software.

Conforme aumenta a importância do software, a comunidade da área tenta desenvolver tecnologias que tornem mais fácil, mais rápido e mais barato desenvolver e manter programas de computador de alta qualidade. Algumas dessas tecnologias são direcionadas a um campo de aplicação específico (por exemplo, projeto e implementação de sites); outras são focadas em um campo de tecnologia (por exemplo, sistemas orientados a objetos ou programação orientada a aspectos); e ainda outras são de bases amplas (por exemplo, sistemas operacionais como o

* N. de R.T.: fanáticos por tecnologia.

"Ideias e descobertas tecnológicas são as forças propulsoras do crescimento econômico."

The Wall Street Journal

Linux). Entretanto, nós ainda temos de desenvolver uma tecnologia de software que faça tudo isso, sendo que a probabilidade de surgir tal tecnologia no futuro é pequena. Ainda assim, as pessoas apostam seus empregos, seu conforto, sua segurança, seu entretenimento, suas decisões e suas próprias vidas em software. Tomara que estejam certas.

Este livro apresenta uma estrutura que pode ser utilizada por aqueles que desenvolvem software — pessoas que devem fazê-lo corretamente. A estrutura abrange um processo, um conjunto de métodos e uma gama de ferramentas que chamaremos de *engenharia de software*.

1.1 A NATUREZA DO SOFTWARE

PONTO-CHAVE

Software é tanto um produto como um veículo que distribui um produto.

Hoje, o software assume um duplo papel. Ele é um produto e, ao mesmo tempo, o veículo para distribuir um produto. Como produto, fornece o potencial computacional representado pelo hardware ou, de forma mais abrangente, por uma rede de computadores que podem ser acessados por hardware local. Independentemente de residir em um celular ou operar dentro de um mainframe, software é um transformador de informações — produzindo, gerenciando, adquirindo, modificando, exibindo ou transmitindo informações que podem ser tão simples quanto um único bit ou tão complexas quanto uma apresentação multimídia derivada de dados obtidos de dezenas de fontes independentes. Como veículo de distribuição do produto, o software atua como a base para o controle do computador (sistemas operacionais), a comunicação de informações (redes) e a criação e o controle de outros programas (ferramentas de software e ambientes).

O software distribui o produto mais importante de nossa era — a *informação*. Ele transforma dados pessoais (por exemplo, transações financeiras de um indivíduo) de modo que possam ser mais úteis num determinado contexto; gerencia informações comerciais para aumentar a competitividade; fornece um portal para redes mundiais de informação (Internet) e os meios para obter informações sob todas as suas formas.

O papel desempenhado pelo software tem passado por grandes mudanças ao longo dos últimos cinquenta anos. Aperfeiçoamentos significativos no desempenho do hardware, mudanças profundas nas arquiteturas computacionais, vasto aumento na capacidade de memória e armazenamento, e uma ampla variedade de exóticas opções de entrada e saída, tudo isso resultou em sistemas computacionais mais sofisticados e complexos. Sofisticação e complexidade podem produzir resultados impressionantes quando um sistema é bem-sucedido, porém, também podem trazer enormes problemas para aqueles que precisam desenvolver sistemas robustos.

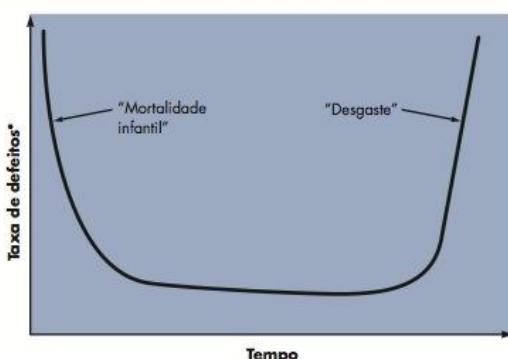
Atualmente, uma enorme indústria de software tornou-se fator dominante nas economias do mundo industrializado. Equipes de especialistas em software, cada qual concentrando-se numa parte da tecnologia necessária para distribuir uma aplicação complexa, substituíram o programador solitário de antigamente. Ainda assim, as questões levantadas por esse programador solitário continuam as mesmas feitas hoje, quando os modernos sistemas computacionais são desenvolvidos:¹

- Por que concluir um software leva tanto tempo?
- Por que os custos de desenvolvimento são tão altos?
- Por que não conseguimos encontrar todos os erros antes de entregarmos o software aos clientes?
- Por que gastamos tanto tempo e esforço mantendo programas existentes?
- Por que continuamos a ter dificuldade em medir o progresso enquanto o software está sendo desenvolvido e mantido?

¹ Em um excelente livro de ensaio sobre o setor de software, Tom DeMarco [DeM95] contesta. Ele afirma: "Em vez de perguntar por que software custa tanto, precisamos começar perguntando: 'O que fizemos para tornar possível que o software atual custe tão pouco?' A resposta a essa pergunta nos ajudará a continuarmos com o extraordinário nível de realização que sempre tem distinguido a indústria de software".

FIGURA 1.1

Curva de defeitos para hardware



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

1.1.1 Definindo software

Hoje em dia, a maior parte dos profissionais e muitos outros indivíduos do público em geral acham que entendem de software. Mas entendem mesmo?

Uma descrição de software em um livro-texto poderia ser a seguinte:



Software consiste em: (1) instruções (programas de computador) que, quando executadas, fornecem características, funções e desempenho desejados; (2) estruturas de dados que possibilitam aos programas manipular informações adequadamente; e (3) informação descritiva, tanto na forma impressa como na virtual, descrevendo a operação e o uso dos programas.

Sem dúvida, poderíamos dar outras definições mais completas.

Mas, provavelmente, uma definição mais formal não melhoraria, de forma considerável, a compreensão do que é software. Para conseguir isso, é importante examinar as características do software que o tornam diverso de outras coisas que os seres humanos constroem. Software é mais um elemento de sistema lógico do que físico. Dessa forma, tem características que são consideravelmente diferentes daquelas do hardware:

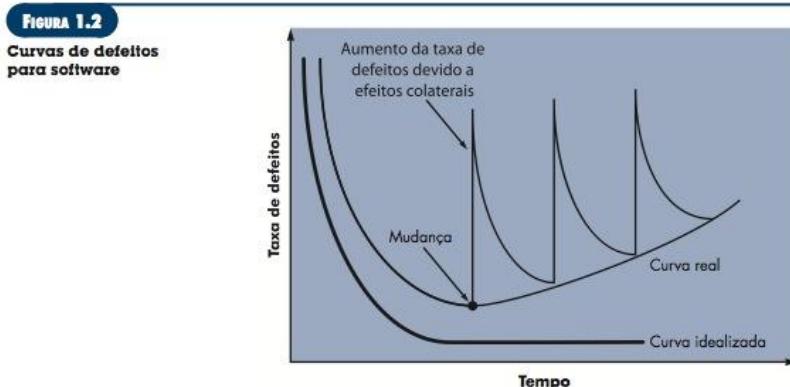
1. *Software é desenvolvido ou passa por um processo de engenharia; ele não é fabricado no sentido clássico.*

Embora existam algumas similaridades entre o desenvolvimento de software e a fabricação de hardware, as duas atividades são fundamentalmente diferentes. Em ambas, alta qualidade é obtida por meio de bom projeto, entretanto, a fase de fabricação de hardware pode gerar problemas de qualidade inexistentes (ou facilmente corrigíveis) para software. Ambas as atividades são dependentes de pessoas, mas a relação entre pessoas envolvidas e trabalho realizado é completamente diferente (ver Capítulo 24). Ambas requerem a construção de um "produto", entretanto, as abordagens são diferentes. Os custos de software concentram-se no processo de engenharia. Isso significa que projetos de software não podem ser geridos como se fossem projetos de fabricação.

* N. de R.T.: os defeitos do software nem sempre se manifestam como falha, geralmente devido a tratamentos dos erros decorrentes destes defeitos pelo software. Esses conceitos serão precisamente mais detalhados e diferenciados nos capítulos sobre qualidade. Neste ponto, optou-se por traduzir *failure rate* por taxa de defeitos, sem prejuízo para a assimilação dos conceitos apresentados pelo autor neste capítulo.

PONTO-CHAVE

Software é um processo de engenharia, não é fabricação.



2. Software não "se desgasta".

A Figura 1.1 representa a taxa de defeitos em função do tempo para hardware. Essa relação, normalmente denominada "curva da banheira", indica que o hardware apresenta taxas de defeitos relativamente altas no início de sua vida (geralmente, atribuídas a defeitos de projeto ou de fabricação); os defeitos são corrigidos e a taxa cai para um nível estável (felizmente, bastante baixo) por certo período. Entretanto, à medida que o tempo passa, a taxa aumenta novamente, conforme os componentes de hardware sofrem os efeitos cumulativos de poeira, vibração, impactos, temperaturas extremas e vários outros males ambientais. Resumindo, o hardware começa a desgastar-se.



Caso queira reduzir a deterioração do software, terá de fazer um projeto melhor de software (Capítulos 8 a 13).

Software não é suscetível aos males ambientais que fazem com que o hardware se desgaste. Portanto, teoricamente, a curva da taxa de defeitos para software deveria assumir a forma da "curva idealizada", mostrada na Figura 1.2. Defeitos ainda não descobertos irão resultar em altas taxas logo no início da vida de um programa. Entretanto, esses serão corrigidos e a curva se achatá como mostrado. A curva idealizada é uma simplificação grosseira de modelos de defeitos reais para software. Porém, a implicação é clara: software não se desgasta, mas sim se deteriora!

Essa aparente contradição pode ser elucidada pela curva real apresentada na Figura 1.2. Durante sua vida², o software passará por alterações. À medida que estas ocorram, é provável que sejam introduzidos erros, fazendo com que a curva de taxa de defeitos se acentue, conforme mostrado na "curva real" (Figura 1.2). Antes que a curva possa retornar à taxa estável original, outra alteração é requisitada, fazendo com que a curva se acentue novamente. Lentamente, o nível mínimo da taxa começa a aumentar — o software está deteriorando devido à modificação.

Outro aspecto de desgaste ilustra a diferença entre hardware e software. Quando um componente de hardware se desgasta, ele é substituído por uma peça de reposição. Não existem peças de reposição de software. Cada defeito de software indica um erro no projeto ou no processo pelo qual o projeto foi traduzido em código de máquina executável. Portanto, as tarefas de manutenção de software, que envolvem solicitações de mudanças, implicam em complexidade consideravelmente maior do que a de manutenção de hardware.

² De fato, desde o momento em que o desenvolvimento começa, e muito antes da primeira versão ser entregue, podem ser solicitadas mudanças por uma variedade de diferentes interessados.



Os métodos de engenharia de software tentam reduzir ao máximo a magnitude das elevações (picos) e a inclinação da curva real da Figura 1.2.

"Ideias são a matéria-prima para construção de ideias."
Jason Zebehazy

- 3.** Embora a indústria caminhe para a construção com base em componentes, a maioria dos softwares continua a ser construída de forma personalizada (sob encomenda).

À medida que a disciplina da engenharia evolui, uma coleção de componentes de projeto padronizados é criada. Parafusos padronizados e circuitos integrados de linha são apenas dois dos milhares de componentes padronizados utilizados por engenheiros mecânicos e elétricos ao projetarem novos sistemas. Os componentes reutilizáveis foram criados para que o engenheiro possa se concentrar nos elementos realmente inovadores de um projeto, isto é, nas partes do projeto que representam algo novo. No mundo do hardware, a reutilização de componentes é uma parte natural do processo de engenharia. No mundo do software, é algo que, em larga escala, apenas começou a ser alcançado.

Um componente de software deve ser projetado e implementado de modo que possa ser reutilizado em muitos programas diferentes. Os modernos componentes reutilizáveis encapsulam tanto dados quanto o processamento aplicado a eles, possibilitando criar novas aplicações a partir de partes reutilizáveis.³ Por exemplo, as atuais interfaces interativas com o usuário são construídas com componentes reutilizáveis que possibilitam criar janelas gráficas, menus "pull-down" (suspenso e retráteis) e uma ampla variedade de mecanismos de interação. Estruturas de dados e detalhes de processamento necessários para a construção da interface ficam em uma biblioteca de componentes reutilizáveis para a construção de interfaces.

1.1.2 Campos de aplicação de software

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

Software de sistema — conjunto de programas feito para atender a outros programas. Certos softwares de sistema (por exemplo, compiladores, editores e utilitários para gerenciamento de arquivos) processam estruturas de informação complexas, porém, determinadas.⁴ Outras aplicações de sistema (por exemplo, componentes de sistema operacional, drivers, software de rede, processadores de telecomunicações) processam dados amplamente indeterminados. Em ambos os casos, a área de software de sistemas é caracterizada por "pesada" interação com o hardware do computador; uso intenso por múltiplos usuários; operação concorrente que requer escala da ordem, compartilhamento de recursos e gestão de processos sofisticada; estruturas de dados complexas e múltiplas interfaces externas.

Software de aplicação — programas sob medida que solucionam uma necessidade específica de negócio. Aplicações nessa área processam dados comerciais ou técnicos de uma forma que facilite operações comerciais ou tomadas de decisão administrativas/técnicas. Além das aplicações convencionais de processamento de dados, o software de aplicação é usado para controlar funções de negócio em tempo real (por exemplo, processamento de transações em pontos de venda, controle de processos de fabricação em tempo real).

Software científico/de engenharia — tem sido caracterizado por algoritmos "number crunching" (para "processamento numérico pesado"). As aplicações vão da astronomia à vulcanologia, da análise de tensões na indústria automotiva à dinâmica orbital de ônibus espaciais, e da biologia molecular à fabricação automatizada. Entretanto, aplicações modernas dentro da área de engenharia/científica estão se afastando dos algoritmos numéricos convencionais. Projeto com o auxílio de computador, simulação de sistemas e outras aplicações interativas começaram a ter características de sistemas em tempo real e até mesmo de software de sistemas.

³ O desenvolvimento com base em componentes é discutido no Capítulo 10.

⁴ Um software é determinado se a ordem e o *timing* (periodicidade, frequência, medidas de tempo) de entradas, processamento e saídas forem previsíveis. É indeterminado, se ordem e *timing* de entradas, processamento e saídas não puderem ser previstos antecipadamente.

WebRef

Uma das mais impressionantes bibliotecas de shareware/fweware (software compartilhado/livre) pode ser encontrada em shareware.onet.com

Software embutido — residente num produto ou sistema e utilizado para implementar e controlar características e funções para o usuário final e para o próprio sistema. Executa funções limitadas e específicas (por exemplo, controle do painel de um forno micro-ondas) ou fornece função significativa e capacidade de controle (por exemplo, funções digitais de automóveis, tal como controle do nível de combustível, painéis de controle e sistemas de freios).

Software para linha de produtos — projetado para prover capacidade específica de utilização por muitos clientes diferentes. Pode focalizar um mercado limitado e particularizado (por exemplo, produtos para controle de estoques) ou direcionar-se para mercados de consumo de massa (por exemplo, processamento de texto, planilhas eletrônicas, computação gráfica, multimídia, entretenimento, gerenciamento de bancos de dados e aplicações financeiras pessoais e comerciais).

Aplicações para a Web — chamadas de "WebApps", essa categoria de software centralizada em redes abrange uma vasta gama de aplicações. Em sua forma mais simples, as WebApps podem ser pouco mais que um conjunto de arquivos de hipertexto interconectados, apresentando informações por meio de texto e informações gráficas limitadas. Entretanto, com o aparecimento da Web 2.0, elas têm evoluído e se transformado em sofisticados ambientes computacionais que não apenas fornecem recursos especializados, funções computacionais e conteúdo para o usuário final, como também estão integradas a bancos de dados corporativos e aplicações comerciais.

"Não existe computador que tenha bom senso."
Marvin Minsky

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

Milhões de engenheiros de software em todo o mundo trabalham arduamente em projetos de software em uma ou mais dessas categorias. Em alguns casos, novos sistemas estão sendo construídos, mas em muitos outros, aplicações já existentes estão sendo corrigidas, adaptadas e aperfeiçoadas. Não é incomum para um jovem engenheiro de software trabalhar num programa mais velho que ele! Gerações passadas de pessoal de software deixaram um legado em cada uma das categorias discutidas. Espera-se que o legado a ser deixado por essa geração facilite o trabalho de futuros engenheiros de software. Ainda assim, novos desafios (Capítulo 31) têm surgido no horizonte:

Computação mundial aberta — o rápido crescimento de redes sem fio pode, em breve, conduzir a uma verdadeira computação distribuída e pervasiva (ampliada, compartilhada e incorporada nos ambientes domésticos e comerciais). O desafio para os engenheiros de software será o de desenvolver sistemas e software aplicativo que permitam que dispositivos móveis, computadores pessoais e sistemas corporativos se comuniquem através de extensas redes.

Netsourcing (recursos via Internet) — a Internet está se tornando, rapidamente, tanto um mecanismo computacional, como um provedor de conteúdo. O desafio para os engenheiros de software consiste em arquitetar aplicações simples (isto é, planejamento financeiro pessoal) e sofisticadas que forneçam benefícios aos mercados mundiais de usuários finais visados.

Software aberto — uma tendência crescente que resulta na distribuição de código-fonte para aplicações de sistemas (por exemplo, sistemas operacionais, bancos de dados e ambientes de desenvolvimento), de forma que muitas pessoas possam contribuir para seu desenvolvimento. O desafio para os engenheiros de software consiste em construir um código-fonte autodescritivo, porém, mais importante ainda, será desenvolver técnicas que permitam que tanto clientes quanto desenvolvedores saibam quais alterações foram feitas e como se manifestam dentro do software.

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

Anônimo

Cada um desses desafios obedecerá, sem dúvida, à lei das consequências não intencionais, produzindo efeitos (para executivos, engenheiros de software e usuários finais) que, hoje, não podem ser previstos. Entretanto, os engenheiros de software podem se preparar, iniciando um processo que seja ágil e suficientemente adaptável para assimilar as profundas mudanças na tecnologia e nas regras comerciais, que certamente virão na próxima década.

1.1.3 Software legado

Centenas de milhares de programas de computador caem em um dos sete amplos campos de aplicação discutidos na subseção anterior. Alguns deles são software de ponta — recém-lançados para indivíduos, indústria e governo. Outros programas são mais antigos, em alguns casos *muito* mais antigos.

Esses programas mais antigos — frequentemente denominados *software legado* — têm sido foco de contínua atenção e preocupação desde os anos 1960. Dayani-Fard e seus colegas [Day99] descrevem software legado da seguinte maneira:

Sistemas de software legado... Foram desenvolvidos décadas atrás e têm sido continuamente modificados para se adequar a mudanças dos requisitos de negócio e a plataformas computacionais. A proliferação de tais sistemas está causando dores de cabeça para grandes organizações que os consideram dispendiosos de manter e arriscados de evoluir.

Liu e seus colegas [Liu98] ampliam essa descrição observando que "muitos sistemas legados permanecem dando suporte para funções de negócios vitais e são 'indispensáveis' para o mesmo". Por isso, um software legado é caracterizado pela longevidade e criticidade de negócios.

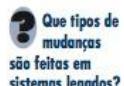
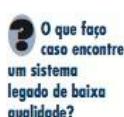
Infelizmente, algumas vezes, há uma característica adicional que pode estar presente em um software legado: *baixa qualidade*.⁵ Os sistemas legados, algumas vezes, têm projetos não expansíveis, código intrincado, documentação pobre ou inexistente, casos de testes e resultados que nunca foram arquivados, um histórico de modificações mal administrado — a lista pode ser bem longa. Ainda assim, esses sistemas dão suporte a "funções vitais de negócio e são indispensáveis para ele". O que fazer então?

A única resposta razoável talvez seja: *Não faça nada*, pelo menos até que o sistema legado tenha que passar por alguma modificação significativa. Se o software legado atende às necessidades de seus usuários e roda de forma confiável, ele não está "quebrado" e não precisa ser "consertado". Entretanto, com o passar do tempo, esses sistemas, frequentemente, evoluem por uma ou mais das seguintes razões:

- O software deve ser adaptado para atender às necessidades de novos ambientes ou de novas tecnologias computacionais.
- O software deve ser aperfeiçoado para implementar novos requisitos de negócio.
- O software deve ser expandido para torná-lo interoperável com outros bancos de dados ou com sistemas mais modernos.
- O software deve ser rearquitetado para torná-lo viável dentro de um ambiente de rede.

Quando essas modalidades de evolução ocorrerem, um sistema legado deve passar por re-engenharia (Capítulo 29) para que permaneça viável no futuro. O objetivo da engenharia de software moderna é o de "elaborar metodologias baseadas na noção de evolução"; isto é, na noção de que os sistemas de software modificam-se continuamente, novos sistemas são construídos a partir dos antigos e... Todos devem interoperar e cooperar um com o outro" [Day99].

⁵ Nesse caso, a qualidade é julgada pensando-se em termos de engenharia de software moderna — um critério um tanto injusto, já que alguns conceitos e princípios da engenharia de software moderna talvez não tenham sido bem entendidos na época em que o software legado foi desenvolvido.



1.2 A NATUREZA ÚNICA DAS WEBAPPS

"Quando notarmos qualquer tipo de estabilidade, a Web terá se transformado em algo completamente diferente."

Louis Monier

Nos primórdios da World Wide Web (por volta de 1990 a 1995), os sites eram formados por nada mais que um conjunto de arquivos de hipertexto lincados que apresentavam informações usando texto e gráficos limitados. Com o tempo, o aumento da HTML, via ferramentas de desenvolvimento (por exemplo, XML, Java), tornou possível aos engenheiros da Internet oferecerem capacidade computacional juntamente com as informações. Nasclaram, então, os *sistemas e aplicações baseados na Web*⁶ (refiro-me a eles coletivamente como aplicações WebApps). Atualmente, as WebApps evoluíram para sofisticadas ferramentas computacionais que não apenas oferecem funções especializadas (*stand-alone functions*) ao usuário final, como também foram integradas aos bancos de dados corporativos e às aplicações de negócios.

Conforme observado na Seção 1.1.2, WebApps são apenas uma dentre uma série de diferentes categorias de software. Ainda assim, pode-se afirmar que elas são diferentes. Powell [Pow98] sugere que sistemas e aplicações baseados na Web "envolvem uma mistura de publicação impressa e desenvolvimento de software, de marketing e computação, de comunicações internas e relações externas e de arte e tecnologia". Os seguintes atributos são encontrados na grande maioria das WebApps:



Que características diferenciam as WebApps de outros softwares?

Uso intensivo de redes. Uma WebApp reside em uma rede e deve atender às necessidades de uma comunidade diversificada de clientes. A rede possibilita acesso e comunicação mundiais (isto é, a Internet) ou acesso e comunicação mais limitados (por exemplo, uma Intranet corporativa).

Simultaneidade. Um grande número de usuários pode acessar a WebApp ao mesmo tempo. Em muitos casos, os padrões de utilização entre os usuários finais variam amplamente.

Carga não previsível. O número de usuários da WebApp pode variar, em ordem de grandeza, de um dia para outro. Uma centena de usuários pode conectar-se na segunda-feira e 10.000 na quinta.

Desempenho. Se um usuário de uma WebApp tiver de esperar muito (para acesso, processamento no servidor, formatação e exibição no cliente), talvez ele procure outra opção.

Disponibilidade. Embora a expectativa de 100% de disponibilidade não seja razoável, usuários de WebApps populares normalmente exigem acesso 24 horas por dia, 7 dias por semana, 365 dias por ano. Usuários na Austrália ou Ásia podem requerer acesso quando aplicações de software domésticas tradicionais na América do Norte estejam off-line para manutenção.

Orientadas a dados. A função principal de muitas WebApps é usar hipermedias para apresentar texto, gráficos, áudio e vídeo para o usuário final. Além disso, as WebApps são comumente utilizadas para acessar informações em bancos de dados que não são parte integrante do ambiente baseado na Web (por exemplo, comércio eletrônico e/ou aplicações financeiras).

Sensibilidade no conteúdo. A qualidade e a natureza estética do conteúdo são fatores importantes que determinam a qualidade de uma WebApp.

Evolução contínua. Diferentemente de softwares de aplicação convencionais que evoluem ao longo de uma série de versões planejadas e cronologicamente espaçadas, as WebApps evoluem continuamente. Não é incomum algumas delas (especificamente seu conteúdo) serem atualizadas segundo uma escala minuto a minuto ou seu conteúdo ser computado independentemente para cada solicitação.

⁶ No contexto deste livro, o termo *aplicação Web* (WebApp) engloba tudo, de uma simples página Web que possa ajudar um consumidor a processar o pagamento do aluguel de um automóvel a um amplo site que fornece serviços de viagem completos para executivos e turistas. Dentro dessa categoria, estão sites completos, funcionalidade especializada dentro de sites e aplicações para processamento de informações residentes na Internet ou em uma Intranet ou Extranet.

Imediatismo. Embora *immediatismo* — a imperativa necessidade de colocar rapidamente um software no mercado — seja uma característica de diversos campos de aplicação, as WebApps normalmente apresentam um tempo de colocação no mercado que pode consistir de poucos dias ou semanas.⁷

Segurança. Pelo fato de estarem disponíveis via acesso à Internet, torna-se difícil, se não impossível, limitar o número dos usuários finais que podem acessar as WebApps. A fim de proteger conteúdos sensíveis e oferecer modos seguros de transmissão de dados, fortes medidas de segurança devem ser implementadas ao longo da infraestrutura que suporta uma WebApp e dentro da própria aplicação.

Estética. Parte inegável do apelo de uma WebApp consiste na sua aparência e na impressão que desperta. Quando uma aplicação for desenvolvida para o mercado ou para vender produtos ou ideias, a estética pode ser tão importante para o sucesso quanto o projeto técnico.

Pode-se argumentar que outras categorias de aplicação discutidas na Seção 1.1.2 podem exibir alguns dos atributos citados. Entretanto, as WebApps quase sempre apresentam todos esses atributos.

1.3 ENGENHARIA DE SOFTWARE

Para desenvolver software que esteja preparado para enfrentar os desafios do século vinte e um, devemos perceber uns poucos fatos reais:

PONTO-CHAVE

Entenda o problema antes de elaborar uma solução.

PONTO-CHAVE

Projetar é uma atividade fundamental na engenharia de software.

PONTO-CHAVE

Qualidade e facilidade de manutenção são resultantes de um projeto bem feito.

- Software tornou-se profundamente incorporado em praticamente todos os aspectos de nossas vidas e, consequentemente, o número de pessoas interessadas nos recursos e nas funções oferecidas por uma determinada aplicação⁸ tem crescido significativamente. Quando uma aplicação ou um sistema embutido estão para ser desenvolvidos, muitas vezes devem ser ouvidas. E, algumas vezes, parece que cada uma delas possui uma ideia ligeiramente diferente de quais funções ou recursos o software deve oferecer. Depreende-se, portanto, que se deve fazer *um esforço concentrado para compreender o problema antes de desenvolver uma solução de software*.
- Os requisitos de tecnologia de informação demandados por indivíduos, empresas e órgãos governamentais estão se tornando cada vez mais complexos a cada ano. Atualmente, equipes numericamente grandes desenvolvem programas de computador que antigamente eram desenvolvidos por um único indivíduo. Software sofisticado, outrora implementado em um ambiente computacional independente e previsível, hoje em dia está incorporado em tudo, de produtos eletrônicos de consumo a equipamentos médicos e sistemas de armamentos. A complexidade desses novos produtos e sistemas baseados em computadores demanda uma maior atenção para com as interações de todos os elementos do sistema. Depreende-se, portanto, que *projetar tornou-se uma atividade-chave (fundamental)*.
- Indivíduos, negócios e governos dependem, de forma crescente, de software para decisões estratégicas e táticas, assim como para controle e para operações cotidianas. Se o software falhar, as pessoas e as principais empresas poderão vivenciar desde pequenos inconvenientes a falhas catastróficas. Depreende-se, portanto, que *um software deve apresentar qualidade elevada*.
- À medida que o valor de uma aplicação específica aumente, a probabilidade é de que sua base de usuários e longevidade também cresçam. À medida que sua base de usuários e seu tempo em uso forem aumentando, a demanda por adaptação e aperfeiçoamento também irá aumentar. Conclui-se, portanto, que *um software deve ser passível de manutenção*.

⁷ Com o uso de ferramentas modernas, sofisticadas, páginas de sites podem ser produzidas em apenas algumas horas.

⁸ Neste livro, mais à frente, chamarei tais pessoas de “interessados”.



Essas simples constatações nos conduzem a uma única conclusão: *software, em todas as suas formas e em todos os seus campos de aplicação, deve passar pelos processos de engenharia.* E isso nos leva ao tópico principal deste livro — *engenharia de software*.

Embora centenas de autores tenham criado suas definições pessoais de engenharia de software, uma definição proposta por Fritz Bauer [Nau69] na conferência sobre o tema serve de base para discussão:

[Engenharia de software é] o estabelecimento e o emprego de sólidos princípios de engenharia de modo a obter software de maneira econômica, que seja confiável e funcione de forma eficiente em máquinas reais.

Picamos tentados a acrescentar algo a essa definição.⁹ Ela diz pouco sobre os aspectos técnicos da qualidade de software; ela não trata diretamente da necessidade de satisfação do cliente ou da entrega do produto dentro do prazo; ela não faz menção à importância das medições e métricas; ela não declara a importância de um processo eficaz. Ainda assim, a definição de Bauer nos fornece uma base. Quais são os “sólidos princípios de engenharia” que podem ser aplicados ao desenvolvimento de software? Como criar software “economicamente viável” e de modo “confiável”? O que é necessário para desenvolver programas de computador que funcionem “de forma eficiente” não apenas em uma, mas sim em várias e diferentes “máquinas reais”? Essas são as questões que continuam a desafiar os engenheiros de software.

A IEEE [IEE93a] desenvolveu uma definição mais abrangente ao afirmar o seguinte:

Engenharia de software: (1) A aplicação de uma abordagem sistemática, disciplinada e quantificável no desenvolvimento, na operação e na manutenção de software; isto é, a aplicação de engenharia ao software. (2) O estudo de abordagens como definido em (1).

Entretanto, uma abordagem “sistematica, disciplinada e quantificável”, aplicada por uma equipe de desenvolvimento de software, pode ser pesada para outra. Precisamos de disciplina, mas também precisamos de adaptabilidade e agilidade.

A engenharia de software é uma tecnologia em camadas. Referindo-se à Figura 1.3, qualquer abordagem de engenharia (inclusive engenharia de software) deve estar fundamentada em um comprometimento organizacional com a qualidade. A gestão da qualidade total Seis Sigma e filosofias similares¹⁰ promovem uma cultura de aperfeiçoamento contínuo de processos, e é esta cultura que, no final das contas, leva ao desenvolvimento de abordagens cada vez mais efetivas na engenharia de software. A pedra fundamental que sustenta a engenharia de software é o foco na qualidade.

A base para a engenharia de software é a camada de *processos*. O processo de engenharia de software é a liga que mantém as camadas de tecnologia coesas e possibilita o desenvolvimento de software de forma racional e dentro do prazo. O processo define uma metodologia que deve ser estabelecida para a entrega efetiva de tecnologia de engenharia de software. O processo de

⁹ Para inúmeras definições adicionais de *engenharia de software*, consulte: www.answers.com/topic/software-engineering#wp-note-13.

¹⁰ A gestão da qualidade e metodologias relacionadas são discutidas no Capítulo 14 e ao longo da Parte 3 deste livro.

PONTO-CHAVE

A engenharia de software engloba um processo, métodos de gerenciamento e desenvolvimento de software, bem como ferramentas.



PONTO-CHAVE

A engenharia de software engloba um processo, métodos de gerenciamento e desenvolvimento de software, bem como ferramentas.

WebRef

CrossTalk é um jornal que divulga informações práticas a respeito de processo, métodos e ferramentas. Pode ser encontrado no endereço: www.stsc.hill.af.mil.

software constitui a base para o controle do gerenciamento de projetos de software e estabelece o contexto no qual são aplicados métodos técnicos, são produzidos produtos derivados (modelos, documentos, dados, relatórios, formulários etc.), são estabelecidos marcos, a qualidade é garantida e mudanças são geridas de forma apropriada.

Os *métodos* da engenharia de software fornecem as informações técnicas para desenvolver software. Os métodos envolvem uma ampla gama de tarefas, que incluem: comunicação, análise de requisitos, modelagem de projeto, construção de programa, testes e suporte.

Os métodos da engenharia de software baseiam-se em um conjunto de princípios básicos que governam cada área da tecnologia e inclui atividades de modelagem e outras técnicas descriptivas.

As *ferramentas* da engenharia de software fornecem suporte automatizado ou semiautomatizado para o processo e para os métodos. Quando as ferramentas são integradas, de modo que as informações criadas por uma ferramenta possam ser usadas por outra, é estabelecido um sistema para o suporte ao desenvolvimento de software, denominado *engenharia de software com o auxílio do computador*.

1.4 O PROCESSO DE SOFTWARE

 **Quais são os elementos de um processo de software?**

"Um processo define quem está fazendo o quê, quando e como para atingir determinado objetivo."

Ivar Jacobson,
Grady Booch
e James Rumbaugh

Processo é um conjunto de atividades, ações e tarefas realizadas na criação de algum produto de trabalho (*work product*). Uma *atividade* esforça-se para atingir um objetivo amplo (por exemplo, comunicar-se com os interessados) e é utilizada independentemente do campo de aplicação, do tamanho do projeto, da complexidade de esforços ou do grau de rigor com que a engenharia de software será aplicada. Uma *ação* (por exemplo, projeto de arquitetura) envolve um conjunto de tarefas que resultam num artefato de software fundamental (por exemplo, um modelo de projeto de arquitetura). Uma *tarefa* se concentra em um objetivo pequeno, porém, bem definido (por exemplo, realizar um teste de unidades) e produz um resultado tangível.

No contexto da engenharia de software, um *processo* não é uma prescrição rígida de como desenvolver um software. Ao contrário, é uma abordagem adaptável que possibilita às pessoas (a equipe de software) realizar o trabalho de selecionar e escolher o conjunto apropriado de ações e tarefas. A intenção é a de sempre entregar software dentro do prazo e com qualidade suficiente para satisfazer àqueles que patrocinaram sua criação e àqueles que irão utilizá-lo.

Uma *metodologia (framework) de processo* estabelece o alicerce para um processo de engenharia de software completo, por meio da identificação de um pequeno número de *atividades estruturais* aplicáveis a todos os projetos de software, independentemente de tamanho ou complexidade. Além disso, a metodologia de processo engloba um conjunto de *atividades de apoio (umbrella activities — abertas)* aplicáveis em todo o processo de software. Uma metodologia de processo genérica para engenharia de software compreende cinco atividades:

 **Quais são as cinco atividades genéricas de metodologia de processo?**

Comunicação. Antes de iniciar qualquer trabalho técnico, é de vital importância comunicar-se e colaborar com o cliente (e outros interessados)¹¹. A intenção é compreender os objetivos das partes interessadas para com o projeto e fazer o levantamento das necessidades que ajudarão a definir as funções e características do software.

Planejamento. Qualquer jornada complicada pode ser simplificada caso exista um mapa. Um projeto de software é uma jornada complicada, e a atividade de planejamento cria um "mapa" que ajuda a guiar a equipe na sua jornada. O mapa — denominado *plano de projeto de software* — define o trabalho de engenharia de software, descrevendo as tarefas técnicas a ser

¹¹ *Interessado* é qualquer um que tenha interesse no êxito de um projeto — executivos, usuários finais, engenheiros de software, o pessoal de suporte etc. Rob Thomsett ironiza: "Interessado [stakeholder, em inglês] é uma pessoa que está segurando [holding, em inglês] uma estaca [stake, em inglês] grande e pontiaguda. Se você não cuidar de seus interessados, você sabe exatamente onde irá parar essa estaca."

"Einstein argumentou que devia haver uma explicação simplificada da natureza, pois Deus não é caprichoso ou arbitrário. Tal fé não conforta o engenheiro de software. Grande parte da complexidade com a qual terá de lidar é arbitrária."

Fred Brooks

conduzidas, os riscos prováveis, os recursos que serão necessários, os produtos resultantes a ser produzidos e um cronograma de trabalho.

Modelagem. Independentemente de ser um paisagista, um construtor de pontes, um engenheiro aeronáutico, um carpinteiro ou um arquiteto, trabalha-se com modelos todos os dias. Cria-se um "esboço" da coisa, de modo que se possa ter uma ideia do todo — qual será o seu aspecto em termos de arquitetura, como as partes constituintes se encaixarão e várias outras características. Se necessário, refina-se o esboço com mais detalhes, numa tentativa de compreender melhor o problema e como resolvê-lo. Um engenheiro de software faz a mesma coisa criando modelos para melhor entender as necessidades do software e o projeto que irá atender a essas necessidades.

Construção. Essa atividade combina geração de código (manual ou automatizada) e testes necessários para revelar erros na codificação.

Emprego. O software (como uma entidade completa ou como um incremento parcialmente efetivado) é entregue ao cliente, que avalia o produto entregue e fornece feedback, baseado na avaliação.

Essas cinco atividades metodológicas genéricas podem ser utilizadas para o desenvolvimento de programas pequenos e simples, para a criação de grandes aplicações para a Internet e para a engenharia de grandes e complexos sistemas baseados em computador. Os detalhes do processo de software serão bem diferentes em cada um dos casos, mas as atividades metodológicas permanecerão as mesmas.

Para muitos projetos de software, as atividades metodológicas são aplicadas iterativamente conforme o projeto se desenvolve. Ou seja, **comunicação, planejamento, modelagem, construção e emprego** são aplicados repetidamente quantas forem as iterações do projeto, sendo que cada iteração produzirá um *incremento de software*. Este disponibilizará uma parte dos recursos e funcionalidades do software. A cada incremento, o software torna-se mais e mais completo.

As atividades metodológicas do processo de engenharia de software são complementadas por uma série de *atividades de apoio*; em geral, estas são aplicadas ao longo de um projeto, ajudando a equipe a gerenciar, a controlar o progresso, a qualidade, as mudanças e o risco. As atividades de apoio típicas são:

Controle e acompanhamento do projeto — possibilita que a equipe avalie o progresso em relação ao plano do projeto e tome as medidas necessárias para cumprir o cronograma.

Administração de riscos — avalia riscos que possam afetar o resultado ou a qualidade do produto/projeto.

Garantia da qualidade de software — define e conduz as atividades que garantem a qualidade do software.

Revisões técnicas — avaliam artefatos da engenharia de software, tentando identificar e eliminar erros antes que se propaguem para a atividade seguinte.

Medição — define e coleta medidas (do processo, do projeto e do produto). Auxilia na entrega do software de acordo com os requisitos; pode ser usada com as demais atividades (metodológicas e de apoio).

Gerenciamento da configuração de software — gerencia os efeitos das mudanças ao longo do processo.

Gerenciamento da reusabilidade — define critérios para o reúso de artefatos (inclusive componentes de software) e estabelece mecanismos para a obtenção de componentes reutilizáveis.

PONTO-CHAVE

Atividades de apoio ocorrem ao longo do processo de software e se concentram, principalmente, no gerenciamento, acompanhamento e controle do projeto.

PONTO-CHAVE

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

Preparo e produção de artefatos de software — engloba as atividades necessárias para criar artefatos como, por exemplo, modelos, documentos, logs, formulários e listas.

Cada uma dessas atividades universais será discutida de forma aprofundada mais adiante.

Anteriormente, declarei que o processo de engenharia de software não é rígido nem deve ser seguido à risca. Mais que isso, ele deve ser ágil e adaptável (ao problema, ao projeto, à equipe e à cultura organizacional). Portanto, o processo adotado para um determinado projeto pode ser muito diferente daquele adotado para outro. Entre as diferenças, temos:

- fluxo geral de atividades, ações e tarefas e suas interdependências;
- grau pelo qual ações e tarefas são definidas dentro de cada atividade da metodologia;
- grau pelo qual artefatos de software são identificados e exigidos;
- modo de aplicar as atividades de garantia de qualidade;
- modo de aplicar as atividades de acompanhamento e controle do projeto;
- grau geral de detalhamento e rigor da descrição do processo;
- grau de envolvimento com o projeto (por parte do cliente e de outros interessados);
- nível de autonomia dada à equipe de software;
- grau de prescrição da organização da equipe.

A Parte I deste livro examina o processo de software com grau de detalhamento considerável. Os *modelos de processo prescritivo* (Capítulo 2) abordam detalhadamente a definição, a identificação e a aplicação de atividades e tarefas do processo. A intenção é melhorar a qualidade do sistema, tornar os projetos mais gerenciáveis, tornar as datas de entrega e os custos mais previsíveis e orientar as equipes de engenheiros de software conforme realizam o trabalho de desenvolvimento de um sistema. Infelizmente, por vezes, tais objetivos não são alcançados. Se os modelos prescritivos forem aplicados de forma dogmática e sem adaptações, poderão aumentar a burocracia associada ao desenvolvimento de sistemas computacionais e, inadvertidamente, criarão dificuldades para todos os envolvidos.

Os *modelos ágeis de processo* (Capítulo 3) ressaltam a "agilidade" do projeto, seguindo princípios que conduzem a uma abordagem mais informal (porém, não menos eficiente) para o processo de software. Tais modelos de processo geralmente são caracterizados como "ágeis", porque enfatizam a flexibilidade e adaptabilidade. Eles são apropriados para vários tipos de projetos e são particularmente úteis quando aplicações para a Internet são projetadas.



"Sinto que uma receita consiste em apenas um tema com o qual um cozinheiro inteligente pode brincar, cada vez, com uma variação."

Madame Benoit



1.5 A PRÁTICA DA ENGENHARIA DE SOFTWARE

WebRef

Uma variedade de citações provocativas sobre a prática da engenharia de software pode ser encontrada em www.literateprogramming.com

A Seção 1.4 apresentou uma introdução a um modelo de processo de software genérico composto por um conjunto de atividades que estabelecem uma metodologia para a prática da engenharia de software. As atividades genéricas da metodologia — **comunicação, planejamento, modelagem, construção e emprego** —, bem como as atividades de apoio, estabelecem um esquema para o trabalho da engenharia de software. Mas como a prática da engenharia de software se encaixa nisso? Nas seções seguintes, você adquirirá um conhecimento básico dos princípios e conceitos genéricos que se aplicam às atividades de uma metodologia.¹²

1.5.1 A essência da prática

Em um livro clássico, *How to Solve It*, sobre os modernos computadores, George Polya [Pol45] apontou em linhas gerais a essência da solução de problemas e, consequentemente, a essência da prática da engenharia de software:

1. *Compreender o problema* (comunicação e análise).
2. *Planejar uma solução* (modelagem e projeto de software).

¹² Você deve rever seções relevantes contidas neste capítulo à medida que métodos de engenharia de software e atividades de apoio específicas forem discutidos posteriormente neste livro.



Pode-se afirmar que a abordagem de Polya é simplesmente questão de bom senso. É verdade. Mas é espantoso quanto frequentemente o bom senso é incomum no mundo do software.

3. Executar o plano (geração de código).

4. Examinar o resultado para ter precisão (testes e garantia da qualidade).

No contexto da engenharia de software, essas etapas de bom senso conduzem a uma série de questões essenciais [adaptado de Pol45]:

Compreenda o problema. Algumas vezes é difícil de admitir, porém, a maioria de nós é arrogante quando nos é apresentado um problema. Ouvimos por alguns segundos e então pensamos: "Ah, sim, estou entendendo, vamos começar a resolver este problema". Infelizmente, compreender nem sempre é assim tão fácil. Vale a pena despender um pouco de tempo respondendo a algumas questões simples:

- *Quem tem interesse na solução do problema?* Ou seja, quem são os interessados?
- *Quais são as incógnitas?* Que dados, funções e recursos são necessários para resolver apropriadamente o problema?
- *O problema pode ser compartmentalizado?* É possível representá-lo em problemas menores que talvez sejam mais fáceis de ser compreendidos?
- *O problema pode ser representado graficamente?* É possível criar um modelo analítico?

"Há um grao de descoberta na solução de qualquer problema."

George Polya

Planeje a solução. Agora você entende o problema (ou assim pensa) e não vê a hora de começar a codificar. Antes de fazer isso, relaxe um pouco e faça um pequeno projeto:

- *Você já viu problemas similares anteriormente?* Existem padrões que são reconhecíveis em uma potencial solução? Existe algum software que implemente os dados, as funções e características necessárias?
- *Algum problema similar já foi resolvido?* Em caso positivo, existem elementos da solução que podem ser reutilizados?
- *É possível definir subproblemas?* Em caso positivo, existem soluções aparentes e imediatas para eles?
- *É possível representar uma solução de maneira que conduza a uma implementação efetiva?* É possível criar um modelo de projeto?

Execute/leve adiante o plano. O projeto elaborado que criamos serve como um mapa para o sistema que se quer construir. Podem surgir desvios inesperados e é possível que se descubra um caminho ainda melhor à medida que se prossiga, porém, o "planejamento" nos permitirá que continuemos sem nos perder.

- *A solução se adéqua ao plano?* O código-fonte pode ser atribuído ao modelo de projeto?
- *Cada uma das partes componentes da solução está provavelmente correta?* O projeto e o código foram revistos, ou, melhor ainda, provas da correção foram aplicadas ao algoritmo?

Examine o resultado. Não se pode ter certeza de que uma solução seja perfeita, porém, pode-se assegurar que um número de testes suficiente tenha sido realizado para revelar o maior número de erros possível.

- *É possível testar cada parte componente da solução?* Foi implementada uma estratégia de testes razoável?
- *A solução produz resultados que se adéquam aos dados, às funções e características necessários?* O software foi validado em relação a todas as solicitações dos interessados?

Não é surpresa que grande parte dessa metodologia consiste no bom senso. De fato, é razoável afirmar que uma abordagem de bom senso à engenharia de software jamais o levará ao mau caminho.



Antes de iniciar um projeto, certifique-se de que o software tem um propósito para a empresa e de que seus usuários reconheçam seu valor.

1.5.2 Princípios gerais

O dicionário define a palavra *princípio* como "uma importante afirmação ou lei subjacente em um sistema de pensamento". Ao longo deste livro serão discutidos princípios em vários níveis de abstração. Alguns se concentram na engenharia de software como um todo, outros consideram uma atividade de metodologia genérica específica (por exemplo, **comunicação**) e outros ainda destacam as ações de engenharia de software (por exemplo, projeto de arquitetura) ou tarefas técnicas (por exemplo, redigir um cenário de uso). Independentemente do seu nível de enfoque, os princípios ajudam a estabelecer um modo de pensar para a prática segura da engenharia de software — esta é a razão porque são importantes.

David Hooker [Hoo96] propôs sete princípios que se concentram na prática da engenharia de software como um todo. Eles são reproduzidos nos parágrafos a seguir:¹³

Primeiro princípio: a razão de existir

Um sistema de software existe por uma única razão: gerar valor a seus usuários. Todas as decisões deveriam ser tomadas tendo esse princípio em mente. Antes de especificar uma necessidade de um sistema, antes de indicar alguma parte da funcionalidade de um sistema, antes de determinar as plataformas de hardware ou os processos de desenvolvimento, pergunte a si mesmo: "Isso realmente agrupa valor real ao sistema?". Se a resposta for "não", não o faça. Todos os demais princípios se apoiam neste primeiro.

Segundo princípio: KISS (Keep It Simple, Stupid!, ou seja: Faça de forma simples, tapado!)

O projeto de software não é um processo casual; há muitos fatores a ser considerados em qualquer esforço de projeto — *todo projeto deve ser o mais simples possível, mas não tão simples assim*. Esse princípio contribui para um sistema mais fácil de compreender e manter. Isso não significa que características, até mesmo as internas, devam ser descartadas em nome da simplicidade.

De fato, frequentemente os projetos mais elegantes são os mais simples, o que não significa "rápido e malfeito" — na realidade, simplificar exige muita análise e trabalho durante as iterações, sendo que o resultado será um software de fácil manutenção e menos propenso a erros.

Terceiro princípio: mantenha a visão

Uma visão clara é essencial para o sucesso. Sem ela, um projeto se torna ambíguo. Sem uma integridade conceitual, corre-se o risco de transformar o projeto numa colcha de retalhos de projetos incompatíveis, unidos por parafusos inadequados... Comprometer a visão arquitetônica de um sistema de software debilita e até poderá destruir sistemas bem projetados. Ter um arquiteto responsável e capaz de manter a visão clara e de reforçar a adequação ajuda a assegurar o êxito de um projeto.

Quarto princípio: o que um produz outros consomem

Raramente um sistema de software de força industrial é construído e utilizado de forma isolada. De uma maneira ou de outra, alguém mais irá usar, manter, documentar ou, de alguma forma, dependerá da capacidade de entender seu sistema. Portanto, *sempre especifique, projete e implemente ciente de que alguém mais terá de entender o que você está fazendo*. O público para qualquer produto de desenvolvimento de software é potencialmente grande. Especifique tendo em vista os usuários; projete, tendo em mente os implementadores; e codifique considerando aqueles que terão de manter e estender o sistema. Alguém terá de depurar o código que você escreveu e isso o fará um usuário de seu código; facilitando o trabalho de todas essas pessoas você agrupa maior valor ao sistema.

¹³ Reproduzido com a permissão do autor [Hoo96]. Hooker define padrões para esses princípios em <http://c2.com/cgi/wiki?SevenPrinciplesOfSoftwareDevelopment>.



Um software de valor mudará ao longo de sua vida. Por essa razão, um software deve ser desenvolvido para fácil manutenção.

Quinto princípio: esteja aberto para o futuro

Um sistema com tempo de vida mais longo tem mais valor. Nos ambientes computacionais de hoje, em que as especificações mudam de um instante para outro e as plataformas de hardware se tornam rapidamente obsoletas, a vida de um software, em geral, é medida em meses. Entretanto, verdadeiros sistemas de software com força industrial devem durar um período muito maior — e, para isso, devem estar preparados para se adaptar a mudanças. Sistemas que obtêm sucesso são aqueles que foram projetados dessa forma desde seu princípio.

Jamais faça projetos limitados, sempre pergunte “e se” e prepare-se para todas as possíveis respostas, criando sistemas que resolvam o problema geral, não apenas aquele específico.¹⁴ Isso muito provavelmente conduziria à reutilização de um sistema inteiro.

Sexto princípio: planeje com antecedência, visando a reutilização

A reutilização economiza tempo e esforço,¹⁵ alcançar um alto grau de reúso é indiscutivelmente a meta mais difícil de ser atingida ao se desenvolver um sistema de software. A reutilização de código e projetos tem sido proclamada como o maior benefício do uso de tecnologias orientadas a objetos, entretanto, o retorno desse investimento não é automático. Alavancar as possibilidades de reutilização (oferecida pela programação orientada a objetos [ou convencional]) requer planejamento e capacidade de fazer previsões. Existem várias técnicas para levar a cabo a reutilização em cada um dos níveis do processo de desenvolvimento do sistema. *Planejar com antecedência para o reúso reduz o custo e aumenta o valor tanto dos componentes reutilizáveis quanto dos sistemas aos quais eles serão incorporados.*

Sétimo princípio: pense!

Este último princípio é, provavelmente, aquele que é mais menosprezado. *Pensar bem e de forma clara antes de agir quase sempre produz melhores resultados.* Quando se analisa alguma coisa, provavelmente esta sairá correta. Ganhá-se também conhecimento de como fazer correto novamente. Se você realmente analisar algo e mesmo assim o fizer da forma errada, isso se tornará uma valiosa experiência. Um efeito colateral da análise é aprender a reconhecer quando não se sabe algo, e até que ponto poderá buscar o conhecimento. Quando a análise clara fez parte de um sistema, seu valor aflora. Aplicar os seis primeiros princípios requer intensa reflexão, para a qual as recompensas potenciais são enormes.

Se todo engenheiro de software e toda a equipe de software simplesmente seguissem os sete princípios de Hooker, muitas das dificuldades enfrentadas no desenvolvimento de complexos sistemas baseados em computador seriam eliminadas.

1.6 MITOS RELATIVOS AO SOFTWARE

“Na falta de padrões significativos substituindo o folclore, surge uma nova indústria como a do software.”
Tom DeMarco

Os mitos criados em relação ao software — crenças infundadas sobre o software e sobre o processo usado para criá-lo — remontam aos primórdios da computação. Os mitos possuem uma série de atributos que os tornam insidiosos. Por exemplo, eles parecem ser, de fato, afirmações razoáveis (algumas vezes contendo elementos de verdade), têm uma sensação intuitiva e frequentemente são promulgados por praticantes experientes “que entendem do riscado”.

Atualmente, a maioria dos profissionais versados na engenharia de software reconhece os mitos por aquilo que eles representam — atitudes enganosas que provocaram sérios problemas tanto para gerentes quanto para praticantes da área. Entretanto, antigos hábitos e atitudes são difíceis de ser modificados e resquícios de mitos de software permanecem.

¹⁴ Esse conselho pode ser perigoso se levado a extremos. Projetar para o “problema geral” algumas vezes requer compromissos de desempenho e pode tornar ineficientes as soluções específicas.

¹⁵ Embora isso seja verdade para aqueles que reutilizam o software em futuros projetos, a reutilização pode ser cara para aqueles que precisem projetar e desenvolver componentes reutilizáveis. Estudos indicam que o projeto e o desenvolvimento de componentes reutilizáveis pode custar de 25 a 200% mais que o próprio software. Em alguns casos, o diferencial de custo não pode ser justificado.

WebRef

Software Project Managers Network, no endereço www.spmn.com, pode ajudá-lo a dissipar esses e outros mitos.

Mitos de gerenciamento. Gerentes com responsabilidade sobre software, assim como gerentes da maioria das áreas, frequentemente estão sob pressão para manter os orçamentos, evitar deslizes nos cronogramas e elevar a qualidade. Como uma pessoa que está se afogando e se agarra a uma tábua, um gerente de software muitas vezes se agarra à crença num mito do software, para aliviar a pressão (mesmo que temporariamente).

Mito: Já temos um livro que está cheio de padrões e procedimentos para desenvolver software. Ele não supre meu pessoal com tudo que eles precisam saber?

Realidade: O livro com padrões pode muito bem existir, mas ele é usado? Os praticantes da área estão cientes de que ele existe? Esse livro reflete a prática moderna da engenharia de software? É completo? É adaptável? Está alinhado para melhorar o tempo de entrega, mantendo ainda o foco na qualidade? Em muitos casos, a resposta para todas essas perguntas é "não".

Mito: Se o cronograma atrasar, poderemos acrescentar mais programadores e ficarmos em dia (algumas vezes denominado conceito da "horda mongol").

Realidade: O desenvolvimento de software não é um processo mecânico como o de fábrica. Nas palavras de Brooks [Bro95]: "acrescentar pessoas num projeto de software atrasado só o tornará mais atrasado ainda". A princípio, essa afirmação pode parecer um contrassenso, no entanto, o que ocorre é que, quando novas pessoas entram, as que já estavam terão de gastar tempo situando os recém-chegados, reduzindo, consequentemente, o tempo destinado ao desenvolvimento produtivo. Pode-se adicionar pessoas, mas somente de forma planejada e bem coordenada.

Mito: Se eu decidir terceirizar o projeto de software, posso simplesmente relaxar e deixar essa empresa realizá-lo.

Realidade: Se uma organização não souber gerenciar e controlar projetos de software, ela irá, invariavelmente, enfrentar dificuldades ao terceirizá-los.

Mitos dos clientes. O cliente solicitante do software computacional pode ser uma pessoa na mesa ao lado, um grupo técnico do andar de baixo, de um departamento de marketing/vendas, ou uma empresa externa que encomendou o projeto por contrato. Em muitos casos, o cliente acredita em mitos sobre software porque gerentes e profissionais da área pouco fazem para corrigir falsas informações. Mitos conduzem a falsas expectativas (do cliente) e, em última instância, à insatisfação com o desenvolvedor.

Mito: Uma definição geral dos objetivos é suficiente para começar a escrever os programas — podemos preencher detalhes posteriormente.

Realidade: Embora nem sempre seja possível uma definição ampla e estável dos requisitos, uma definição de objetivos ambígua é receita para um desastre. Requisitos não ambíguos (normalmente derivados da iteratividade) são obtidos somente pela comunicação contínua e eficaz entre cliente e desenvolvedor.

Mito: Os requisitos de software mudam continuamente, mas as mudanças podem ser facilmente assimiladas, pois o software é flexível.

Realidade: É verdade que os requisitos de software mudam, mas o impacto da mudança varia dependendo do momento em que ela foi introduzida. Quando as mudanças dos requisitos são solicitadas cedo (antes do projeto ou da codificação terem começado), o impacto sobre os custos é relativamente pequeno. Entretanto, conforme o tempo passa, ele aumenta rapidamente — recursos foram comprometidos, uma estrutura de projeto foi estabelecida e mudar pode causar uma revolução que exija recursos adicionais e modificações fundamentais no projeto.



Esforce-se ao máximo para compreender o que deve fazer antes de começar. Você pode não chegar a todos os detalhes, mas quanto mais você souber, menor será o risco.



Toda vez que pensar:
"não temos tempo
para engenharia de
software", pergunte a
si mesmo, "teremos
tempo para fazer de
novo?".

Mitos dos profissionais da área. Mitos que ainda sobrevivem nos profissionais da área têm resistido por mais de 50 anos de cultura de programação. Durante seus primórdios, a programação era vista como uma forma de arte. Modos e atitudes antigos dificilmente morrem.

Mito: Uma vez feito um programa e o colocado em uso, nosso trabalho está terminado.

Realidade: Uma vez alguém já disse que "o quanto antes se começar a codificar, mais tempo levará para terminá-lo". Levantamentos indicam que entre 60 e 80% de todo o esforço será despendido após a entrega do software ao cliente pela primeira vez.

Mito: Até que o programa entre em funcionamento, não há maneira de avaliar sua qualidade.

Realidade: Um dos mecanismos de garantia da qualidade de software mais eficaz pode ser aplicado desde a concepção de um projeto — a revisão técnica. Revisões de software (descritas no Capítulo 15) são um "filtro de qualidade" que mostram ser mais eficientes do que testes para encontrar certas classes de defeitos de software.

Mito: O único produto passível de entrega é o programa em funcionamento.

Realidade: Um programa funcionando é somente uma parte de uma configuração de software que inclui muitos elementos. Uma variedade de produtos derivados (por exemplo, modelos, documentos, planos) constitui uma base para uma engenharia bem-sucedida e, mais importante, uma orientação para suporte de software.

Mito: A engenharia de software nos fará criar documentação volumosa e desnecessária e, invariavelmente, irá nos retardar.

Realidade: A engenharia de software não trata de criação de documentos, trata da criação de um produto de qualidade. Melhor qualidade conduz à redução do retrabalho, e menos retrabalho resulta em maior rapidez na entrega.

Muitos profissionais de software reconhecem a falácia dos mitos que acabamos de descrever. Lamentavelmente, métodos e atitudes habituais fomentam tanto gerenciamento quanto mediadas técnicas deficientes, mesmo quando a realidade exige uma abordagem melhor. Ter ciência das realidades do software é o primeiro passo para buscar soluções práticas na engenharia de software.

1.7 Como Tudo Começou

Todo projeto de software é motivado por alguma necessidade de negócios — a necessidade de corrigir um defeito em uma aplicação existente; a necessidade de adaptar um "sistema legado" a um ambiente de negócios em constante transformação; a necessidade de estender as funções e os recursos de uma aplicação existente, ou a necessidade de criar um novo produto, serviço ou sistema.

No início de um projeto de software, a necessidade do negócio é, com frequência, expressa informalmente como parte de uma simples conversa. A conversa apresentada no quadro a seguir é típica.

Exceto por uma rápida referência, o software mal foi mencionado como parte da conversação. Ainda assim, o software irá decretar o sucesso ou o fracasso da linha de produtos *CasaSegura*. A empreitada de engenharia terá êxito apenas se o software para a linha *CasaSegura* tiver êxito; e o mercado irá aceitar o produto apenas se o software incorporado atender adequadamente às necessidades (ainda não declaradas) do cliente. Acompanharemos a evolução da engenharia do software *CasaSegura* em vários dos capítulos que estão por vir.

CASASEGURA¹⁶



Como começa um projeto

Cena: Sala de reuniões da CPI Corporation, empresa [fictícia] que fabrica produtos de consumo para uso doméstico e comercial.

Atores: Mal Golden, gerente sênior, desenvolvimento do produto; Lisa Perez, gerente de marketing; Lee Warren, gerente de engenharia; Joe Camalleri, vice-presidente executivo, desenvolvimento de negócios.

Conversa:

Joe: Lee, ouvi dizer que o seu pessoal está construindo algo. Do que se trata? Um tipo de caixa sem fio de uso amplo e genérico?

Lee: Trata-se de algo bem legal... Aproximadamente do tamanho de uma caixa de fósforos, conectável a todo tipo de sensor, como uma câmera digital — ou seja, é conectável a quase tudo. Usa o protocolo sem fio 802.11g, permitindo que acessemos saídas de dispositivos sem o emprego de fios. Acreditamos que nos levará a uma geração de produtos inteiramente nova.

Joe: Você concorda, Mal?

Mal: Sim. Na verdade, com as vendas tão em baixa quanto neste ano, precisamos de algo novo. Lisa e eu fizemos uma pequena pesquisa de mercado e acreditamos que conseguimos uma linha de produtos que poderá ser ampla.

Joe: Ampla em que sentido?

Mal (evitando comprometimento direto): Conte a ele sobre nossa ideia, Lisa.

Lisa: Trata-se de uma geração completamente nova na linha de "produtos de gerenciamento doméstico". Chamamos esses produtos que criamos de CasaSegura. Eles usam uma nova interface sem fio e oferecem a pequenos empresários e proprietários de casas um sistema que é controlado por seus PCs, envolvendo segurança doméstica, sistemas de vigilância, controle de eletrodomésticos e dispositivos. Por exemplo, seria possível diminuir a temperatura do aparelho de ar condicionado enquanto você está voltando para casa, esse tipo de coisa.

Lee (reagindo sem pensar): O departamento de engenharia fez um estudo de viabilidade técnica dessa ideia, Joe. É possível fazê-lo com um baixo custo de fabricação. A maior parte dos componentes do hardware é encontrada no mercado; o software é um problema, mas não é nada que não possamos resolver.

Joe: Interessante, mas eu perguntei sobre o levantamento final.

Mal: Os PCs estão em mais de 70% das lareiras, se formos capazes de estabelecer um preço baixo para essa coisa, ela poderia se tornar um produto "revolucionário". Ninguém mais tem nosso dispositivo sem fio... Ele é exclusivo! Estaremos 2 anos à frente de nossos concorrentes... E as receitas? Algo em torno de 30 a 40 milhões de dólares no segundo ano...

Joe (sorrindo): Vamos levar isso adiante. Estou interessado.

1.8 RESUMO

Software é o elemento-chave na evolução de produtos e sistemas baseados em computador e uma das mais importantes tecnologias no cenário mundial. Ao longo dos últimos 50 anos, o software evoluiu de uma ferramenta especializada em análise de informações e resolução de problemas para uma indústria propriamente dita. Mesmo assim, ainda temos problemas para desenvolver software de boa qualidade dentro do prazo e orçamento estabelecidos.

Softwares — programas, dados e informações descritivas — contemplam uma ampla gama de áreas de aplicação e tecnologia. O software legado continua a representar desafios especiais àqueles que precisam fazer sua manutenção.

As aplicações e os sistemas baseados na Internet passaram de simples conjuntos de conteúdo informativo para sofisticados sistemas que apresentam funcionalidade complexa e conteúdo multimídia. Embora essas WebApps possuam características e requisitos exclusivos, elas não deixam de ser um tipo de software.

A engenharia de software engloba processos, métodos e ferramentas que possibilitam a construção de sistemas complexos baseados em computador dentro do prazo e com qualidade. O processo de software incorpora cinco atividades estruturais: comunicação, planejamento, modelagem, construção e emprego; e elas se aplicam a todos os projetos de software. A prática

16 O projeto CasaSegura será usado ao longo deste livro para ilustrar o funcionamento interno de uma equipe de projeto à medida que ela constrói um produto de software. A empresa, o projeto e as pessoas são inteiramente fictícias, porém as situações e os problemas são reais.

da engenharia de software é uma atividade de resolução de problemas que segue um conjunto de princípios básicos.

Inúmeros mitos em relação ao software continuam a levar gerentes e profissionais para o mau caminho, mesmo com o aumento do conhecimento coletivo de software e das tecnologias necessárias para construí-los. À medida que for aprendendo mais sobre a engenharia de software, você começará a compreender porque esses mitos devem ser derrubados toda vez que se deparar com eles.

PROBLEMAS E PONTOS A PONDERAR

- 1.1.** Cite pelo menos cinco outros exemplos de como a lei das consequências não intencionais se aplica ao software.
- 1.2.** Forneça uma série de exemplos (positivos e negativos) que indiquem o impacto do software em nossa sociedade.
- 1.3.** Desenvolva suas próprias respostas às cinco perguntas colocadas no início da Seção 1.1. Discuta-as com seus colegas.
- 1.4.** Muitas aplicações modernas mudam com frequência — antes de serem apresentadas ao usuário final e só então a primeira versão ser colocada em uso. Sugira algumas maneiras de construir software para impedir a deterioração decorrente de mudanças.
- 1.5.** Considere as sete categorias de software apresentadas na Seção 1.1.2. Você acha que a mesma abordagem em relação à engenharia de software pode ser aplicada a cada uma delas? Justifique sua resposta.
- 1.6.** A Figura 1.3 coloca as três camadas de engenharia de software acima de uma camada intitulada “foco na qualidade”. Isso implica um programa de qualidade organizacional como o de gestão da qualidade total. Pesquise um pouco a respeito e crie um sumário dos princípios básicos de um programa de gestão da qualidade total.
- 1.7.** A engenharia de software é aplicável quando as WebApps são construídas? Em caso positivo, como poderia ser modificada para atender às características únicas das WebApps?
- 1.8.** À medida que o software invade todos os setores, riscos ao público (devido a programas com imperfeições) passam a ser uma preocupação cada vez maior. Crie um cenário o mais catastrófico possível, porém realista, cuja falha de um programa de computador poderia causar um grande dano (em termos econômico ou humano).
- 1.9.** Descreva uma estrutura de processos com suas próprias palavras. Ao afirmarmos que atividades de modelagem se aplicam a todos os projetos, isso significa que as mesmas tarefas são aplicadas a todos os projetos, independentemente de seu tamanho e complexidade? Justifique.
- 1.10.** As atividades de apoio ocorrem ao longo do processo de software. Você acredita que elas são aplicadas de forma homogênea ao longo do processo ou algumas delas são concentradas em uma ou mais atividades de metodologia?
- 1.11.** Acrescente dois outros mitos à lista apresentada na Seção 1.6 e declare a realidade que acompanha tais mitos.

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES¹⁷

Há literalmente milhares de livros sobre o tema software. A grande maioria trata de linguagens de programação ou aplicações de software, porém poucos tratam do software em si.

¹⁷ A seção *Leitura e fontes de informação adicionais* apresentada no final de cada capítulo apresenta uma breve visão geral de publicações que podem ajudar a expandir o seu entendimento dos principais tópicos apresentados no capítulo. Há um site bem abrangente para dar suporte ao livro *Engenharia de Software: uma abordagem prática* no endereço www.mhhe.com/pressman. Entre os diversos tópicos contemplados no site estão recursos de engenharia de software capítulo a capítulo e dicas de sites que poderão complementar o material apresentado em cada capítulo.

Pressman e Herron (*Software Shock*, Dorset House, 1991) apresentaram uma discussão preliminar (dirigida ao grande público) sobre software e a maneira pela qual os profissionais o desenvolvem. O best-seller de Negroponte (*Being Digital*, Alfred A. Knopf, Inc., 1995) dá uma visão geral da computação e seu impacto global no século XXI. DeMarco (*Why Does Software Cost So Much?* Dorset House, 1995) produziu um conjunto de divertidos e perspicazes ensaios sobre software e o processo pelo qual ele é desenvolvido.

Minasi (*The Software Conspiracy: Why Software Companies Put out Faulty Products, How They Can Hurt You, and What You Can Do*, McGraw-Hill, 2000) argumenta que o "flagelo moderno" dos bugs de software pode ser eliminado e sugere maneiras para concretizar isso. Compaine (*Digital Divide: Facing a Crisis or Creating a Myth*, MIT Press, 2001) defende que a "separação" entre aqueles que têm acesso a fontes de informação (por exemplo, a Web) e aqueles que não o têm está diminuindo, à medida que avançamos na primeira década deste século. Livros de Greenfield (*Everyware: The Dawning Age of Ubiquitous Computing*, New Riders Publishing, 2006) e Loke (*Context-Aware Pervasive Systems: Architectures for a New Breed of Applications*, Auerbach, 2006) introduzem o conceito de software "aberto" e preveem um ambiente sem fio no qual o software deve se adaptar às exigências que surgem em tempo real.

O estado atual da engenharia de software e do processo de software pode ser mais bem determinado a partir de publicações como *IEEE Software*, *IEEE Computer*, *CrossTalk* e *IEEE Transactions on Software Engineering*. Periódicos do setor como *Application Development Trends* e *Cutter IT Journal* normalmente contêm artigos sobre tópicos da engenharia de software. A disciplina é "sintetizada" todos os anos no *Proceeding of the International Conference on Software Engineering*, patrocinado pelo IEEE e ACM e é discutida de forma aprofundada em periódicos como *ACM Transactions on Software Engineering and Methodology*, *ACM Software Engineering Notes* e *Annals of Software Engineering*. Dezenas de milhares de sites são dedicados à engenharia de software e ao processo de software.

Foram publicados vários livros sobre o processo de desenvolvimento de software e sobre a engenharia de software nos últimos anos. Alguns fornecem uma visão geral de todo o processo, ao passo que outros se aprofundam em tópicos específicos importantes em detrimento dos demais. Entre as ofertas mais populares (além deste livro, é claro!), temos:

- Abran, A. e J. Moore, *SWEBOK: Guide to the Software Engineering Body of Knowledge*, IEEE, 2002.
- Andersson, E. et al., *Software Engineering for Internet Applications*, The MIT Press, 2006.
- Christensen, M. e R. Thayer, *A Project Manager's Guide to Software Engineering Best Practices*, IEEE-CS Press (Wiley), 2002.
- Glass, R., *Fact and Fallacies of Software Engineering*, Addison-Wesley, 2002.
- Jacobson, I., *Object-Oriented Software Engineering: A Use Case Driven Approach*, 2.ª ed., Addison-Wesley, 2008.
- Jalote, P., *An Integrated Approach to Software Engineering*, Springer, 2006.
- Fleeger, S., *Software Engineering: Theory and Practice*, 3.ª ed., Prentice-Hall, 2005.
- Schach, S., *Object-Oriented and Classical Software Engineering*, 7.ª ed., McGraw-Hill, 2006.
- Sommerville, I., *Software Engineering*, 8.ª ed., Addison-Wesley, 2006.
- Tsui, F. e O. Karam, *Essentials of Software Engineering*, Jones & Bartlett Publishers, 2006.

Foram publicados diversos padrões de engenharia de software pelo IEEE, pela ISO e suas organizações de padronização ao longo das últimas décadas. Moore (*The Road Map to Software Engineering: A Standards-Based Guide*, Wiley-IEEE Computer Society Press, 2006) disponibiliza uma pesquisa útil de padrões relevantes e como aplicá-los a projetos reais.

Uma ampla gama de fontes de informação sobre engenharia de software e processo de software se encontra à disposição na Internet. Uma lista atualizada de referências relevantes para o processo para o software pode ser encontrada no site www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm.