

9<sup>a</sup> EDIÇÃO

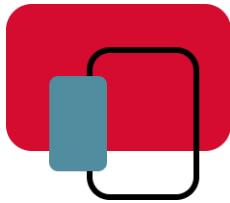
# Engenharia de Software

UMA ABORDAGEM PROFISSIONAL



**ROGER S. PRESSMAN**

**BRUCE R. MAXIM**



## Aviso!

---

Todo esforço foi feito para garantir a qualidade editorial desta obra, agora em versão digital. Destacamos, contudo, que diferenças na apresentação do conteúdo podem ocorrer em função de restrições particulares às versões impressa e digital e das características técnicas específicas de cada dispositivo de leitura.

Este eBook é uma versão da obra impressa, podendo conter referências a este formato (p. ex.: "Circule conforme indicado no exemplo 1", "Preencha o quadro abaixo", etc.). Buscamos adequar todas as ocorrências para a leitura do conteúdo na versão digital, porém alterações e inserções de texto não são permitidas no eBook. Por esse motivo, recomendamos a criação de notas. Em caso de divergências, entre em contato conosco através de [nossa site](#).

**ROGER S. PRESSMAN**

**BRUCE R. MAXIM**

# **Engenharia de Software**

**UMA ABORDAGEM PROFISSIONAL**

**9<sup>a</sup> EDIÇÃO**

Versão impressa desta obra: 2021



---

AMGH Editora Ltda.

Porto Alegre  
2021

**Tradução**  
Francisco Araújo da Costa

**Revisão técnica**

**Reginaldo Arakaki**

Professor do Departamento de Engenharia de Computação e Sistemas Digitais da Escola Politécnica da Universidade de São Paulo (EPUSP). Mestre e Doutor em Engenharia Elétrica pela Universidade de São Paulo (USP).

**Julio Arakaki**

Professor Assistente do Departamento de Computação da Pontifícia Universidade Católica de São Paulo (PUC-SP). Coordenador do Curso de Ciência da Computação da PUC-SP. Mestre e Doutor em Engenharia Mecatrônica e Sistemas Mecânicos pela EPUSP.

**Renato Manzan de Andrade**

Auxiliar de ensino da USP. Pesquisador do Laboratório de Tecnologia de Software (LTS). Mestre em Engenharia Elétrica pela USP. Doutorando do Departamento de Engenharia de Computação e Sistemas Digitais da EPUSP.

Obra originalmente publicada sob o título Software engineering: a practitioner's approach, 9th edition

ISBN 9781259872976 / 1259872971

Original edition copyright © 2020 by McGraw-Hill Global Education Holdings, LLC, New York, New York 10121. All rights reserved.

Portuguese language translation copyright © 2021, by AMGH Editora Ltda., a Grupo A Educação S.A. company. All rights reserved.

**Gerente editorial:** *Arysinha Jacques Affonso*

**Colaboraram nesta edição:**

**Editora:** *Simone de Fraga*

**Arte sobre a capa original:** *Márcio Monticelli*

**Preparação de originais:** *Carine Garcia Prates*

**Projeto gráfico e editoração:** *Clic Editoração Eletrônica Ltda.*

**Produção digital:** *HM Digital Digital.*



---

P934e Pressman, Roger S.

Engenharia de software: uma abordagem profissional

[recurso eletrônico] / Roger S. Pressman, Bruce R. Maxim;

[tradução: Francisco Araújo da Costa; revisão técnica:

Reginaldo Arakaki, Julio Arakaki, Renato Manzan de

Andrade]. – 9. ed. – Porto Alegre: AMGH, 2021.  
E-pub.

Editado também como livro impresso em  
2021.

ISBN 978-65-5804-011-8

1. Engenharia de software. 2. Gestão de  
projetos de  
softwares. I. Maxim, Bruce R. II. Título.

CDU 004.41

---

Catalogação na publicação: Karin Lorien Menoncin – CRB 10/2147

Reservados todos os direitos de publicação à  
AMGH EDITORA LTDA., uma parceria entre GRUPO A  
EDUCAÇÃO S.A. e McGRAW-HILL EDUCATION

Rua Ernesto Alves, 150 – Bairro Floresta  
90220-190 – Porto Alegre – RS  
Fone: (51) 3027-7000

SÃO PAULO  
Rua Doutor Cesário Mota Jr., 63 – Vila Buarque  
01221-020 – São Paulo – SP  
Fone: (11) 3221-9033

SAC 0800 703-3444 – [www.grupoa.com.br](http://www.grupoa.com.br)

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.

*Para Barbara, Matt, Mike, Shiri, Adam, Lily e Maya.*

— Roger S. Pressman

*Para a minha família, que me apoia em tudo que faço.*

— Bruce R. Maxim

# Os autores

---



Cortesia de Roger Pressman

fabricação de acessórios para a linha Tesla de veículos elétricos.

É autor de 10 livros, incluindo dois romances, e de ensaios técnicos e gerenciais. Participou da comissão editorial dos periódicos *IEEE Software* e *The Cutter IT Journal* e foi editor da coluna Manager na *IEEE Software*.

É palestrante renomado. Apresentou tutoriais na International Conference on Software Engineering e em diversos outros congressos do setor. Foi associado da ACM, IEEE, Tau Beta Pi, Phi Kappa Phi, Eta Kappa Nu e Pi Tau Sigma.

**Roger S. Pressman** é um consultor e autor reconhecido internacionalmente. Por quase cinco décadas, trabalhou como engenheiro de *software*, gerente, professor, escritor, consultor e empresário.

Foi presidente da R. S. Pressman & Associates, Inc., consultoria na qual desenvolveu um conjunto de técnicas e ferramentas que melhoraram a prática de engenharia de *software*. É também o fundador e diretor de tecnologia da EVANNEX®, empresa de pós-venda automotivo especializada na criação e

**Bruce R. Maxim** trabalhou como engenheiro de *software*, gerente de projeto, professor, escritor e consultor por mais de 30 anos. Sua pesquisa inclui engenharia de software, projeto da experiência do

usuário, projeto de *games*, inteligência artificial e educação em engenharia.



Michigan Creative/  
UM-Dearborn

desenvolvimento de *software* industrial.

Sua experiência profissional inclui o gerenciamento de sistemas de pesquisa de informações em uma faculdade de medicina, direção educacional da computação em um campus de medicina e atuação como programador estatístico. Trabalhou como diretor-chefe de tecnologia em uma empresa de desenvolvimento de games.

Recebeu vários prêmios por distinção no ensino, um por notável serviço comunitário e outro por distinção na governança do corpo docente. É associado da Sigma Xi, Upsilon Pi Epsilon, Pi Mu Epsilon, Association of Computing Machinery, IEEE Computer Society, American Society for Engineering Education, Society of Women Engineers e International Game Developers Association.

# Prefácio

---

Quando um *software* é bem-sucedido – ou seja, quando atende às necessidades dos usuários, opera perfeitamente durante um longo período, é fácil de modificar e mais fácil ainda de utilizar –, ele pode mudar, e de fato muda, as coisas para melhor. Entretanto, quando um *software* é falho – quando seus usuários estão insatisfeitos, quando é propenso a erros, quando é difícil de modificar e mais difícil ainda de utilizar –, coisas desagradáveis podem acontecer, e de fato acontecem. Todos queremos construir *software* que facilite o trabalho, evitando as falhas que se escondem nos esforços malsucedidos. Para termos êxito, precisamos de disciplina no projeto e na construção do *software*. Precisamos de uma abordagem de engenharia.

Já se passaram quase quatro décadas que a primeira edição deste livro foi escrita. De lá para cá, a engenharia de *software* evoluiu de algo obscuro, praticado por um número relativamente pequeno de adeptos, para uma disciplina de engenharia legítima. Hoje, ela é reconhecida como um assunto digno de pesquisa séria, estudo meticoloso e debates acalorados. O engenheiro de *software* tomou o lugar do programador como cargo mais procurado. Modelos de processos de *software* e métodos de engenharia de *software*, bem como ferramentas de *software*, são adotados com sucesso em muitos segmentos industriais.

Embora gerentes e profissionais da área reconheçam a necessidade de uma abordagem mais disciplinada ao *software*, eles continuam a discutir a melhor forma de fazê-lo. Muitos profissionais e empresas desenvolvem *software* de forma desordenada, até na construção de sistemas para avançadas tecnologias. Grande parte dos profissionais e estudantes não está ciente dos métodos modernos, e isso afeta a qualidade do *software*.

produzido. Continuam também a discussão e a controvérsia sobre a real natureza da abordagem de engenharia de *software*. A engenharia de *software* é um estudo repleto de contrastes. A postura mudou, houve progressos, mas falta muito para essa disciplina atingir a maturidade.

## **Noividades da 9<sup>a</sup> edição**

---

O objetivo desta 9<sup>a</sup> edição de *Engenharia de software: uma abordagem profissional* é ser um guia para uma disciplina de engenharia em fase de amadurecimento. Assim como as edições que a precederam, esta é voltada tanto para estudantes quanto para praticantes, servindo também como guia para profissionais da área e como introdução abrangente para estudantes no final do curso de graduação ou no primeiro ano de pós-graduação.

A 9<sup>a</sup> edição é muito mais do que uma simples atualização. O livro foi revisado e reestruturado para melhorar seu fluxo pedagógico e enfatizar novos e importantes processos e práticas da engenharia de *software*. Além disso, aprimoramos ainda mais o “sistema de apoio” que acompanha o livro, fornecendo um conjunto complementar de recursos para estudantes, instrutores e profissionais.

Os leitores das edições anteriores de *Engenharia de software: uma abordagem profissional* observarão que o número de páginas foi reduzido. Nossa objetivo foi a concisão. Queríamos fortalecer o livro do ponto de vista pedagógico e torná-lo menos intimidante para o leitor que deseja percorrê-lo de ponta a ponta. Há uma história atribuída a Blaise Pascal, famoso físico e matemático francês, que é mais ou menos assim: Pascal escreveu uma carta muito comprida para um amigo e terminou com a seguinte frase: “Queria ter lhe escrito uma carta mais curta, mas não tive tempo”. Trabalhando para tornar a 9<sup>a</sup> edição mais concisa, aprendemos a valorizar as palavras de Pascal.

Os 30 capítulos desta 9<sup>a</sup> edição estão organizados em cinco partes. Essa organização divide melhor os assuntos e ajuda os

professores sem tempo hábil para concluir o livro em um semestre.

A Parte I, *O processo de software*, apresenta diferentes visões, considerando diversas estruturas e modelos de processo importantes e contemplando o debate entre as filosofias de processos ágeis e prescritivos. A Parte II, *Modelagem*, fornece métodos de projeto e análise com ênfase em técnicas orientadas a objetos e modelagem UML. Também são considerados o projeto baseado em padrões e o projeto para aplicativos móveis. A discussão sobre o projeto da experiência do usuário foi expandida nesta seção. A Parte III, *Qualidade e Segurança*, apresenta conceitos, procedimentos, técnicas e métodos que permitem que uma equipe de *software* avalie a qualidade do *software*, revise produtos gerados por engenharia de *software*, realize procedimentos para a garantia de qualidade de *software* (SQA) e aplique estratégias e táticas de teste eficazes. Além disso, apresentamos práticas de segurança de *software* que podem ser inseridas em modelos incrementais de desenvolvimento de *software*. A Parte IV, *Gerenciamento de projetos de software*, aborda tópicos relevantes para aqueles que planejam, gerenciam e controlam um projeto de desenvolvimento de *software*. A Parte V, *Tópicos avançados*, considera o aperfeiçoamento de processos de *software* e tendências da engenharia de *software*. São usadas caixas de texto para apresentar as atribulações de uma equipe (fictícia) de desenvolvimento de *software* e fornecer conteúdo complementar sobre métodos e ferramentas relevantes para os tópicos do capítulo.

A organização em cinco partes permite que o instrutor “agrupue” o conteúdo, considerando o tempo disponível e a necessidade dos alunos. Um curso de um semestre pode ser baseado em uma ou mais das cinco partes. Um curso de pesquisa sobre engenharia de *software* selecionaria capítulos de todas as partes. Um curso de engenharia de *software* que enfatize a análise e o projeto selecionaria tópicos das Partes I e II. Um curso de engenharia de *software* voltado para testes selecionaria tópicos das Partes I e III, com uma breve incursão na Parte II. Um “curso de gerenciamento”

enfatizaria as Partes I e IV. Organizado dessa maneira, o livro oferece ao professor diferentes opções didáticas

## **Recursos adicionais**

Diversos recursos estão disponíveis em língua inglesa no *site* do Grupo A para o instrutor. O arquivo *Professional Resources* oferece centenas de referências *online* categorizadas que permitem que os estudantes explorem a engenharia de *software* em maiores detalhes, junto com uma biblioteca de referências com *links* para recursos *online*, o que gera uma fonte de informações avançadas sobre engenharia de *software* de grande profundidade. Também estão incluídos um *Instructor's guide* completo e materiais pedagógicos complementares, além de centenas de *slides* de PowerPoint que podem ser utilizados em aulas e palestras.

O *Instructor's guide to software engineering: a professional approach* apresenta sugestões sobre como realizar diversos tipos de cursos de engenharia de *software*, recomendações para diversos projetos de *software* a serem desenvolvidos em paralelo ao curso, soluções para uma série de problemas e uma grande quantidade de materiais pedagógicos auxiliares úteis.

Aliada ao seu sistema de apoio online, esta nona edição de *Engenharia de software: uma abordagem profissional* oferece um nível de flexibilidade e conteúdo aprofundado que seria impossível de produzir apenas com um livro-texto.

Bruce Maxim assumiu a responsabilidade de desenvolver novos conteúdos para a 9a edição de *Engenharia de software: uma abordagem profissional*, enquanto Roger Pressman atuou como editor-chefe e fez contribuições pontuais.

## **Agradecimentos**

Agradecimentos especiais a Nancy Mead, do *software Engineering Institute*, da Carnegie Mellon University, que escreveu o capítulo sobre engenharia de segurança de *software*; Tim Lethbridge, da

Universidade de Ottawa, que nos ajudou no desenvolvimento de exemplos em UML e OCL e desenvolveu o estudo de caso que acompanha este livro; Dale Skrien, do Colby College, que desenvolveu o tutorial UML do Apêndice 1; William Grosky, da Universidade de Michigan-Dearborn, que desenvolveu o panorama da ciência de dados do Apêndice 2 ao lado de Terry Ruas, seu aluno; e à nossa colega australiana Margaret Kellow, por atualizar os materiais pedagógicos *online* que acompanham este livro. Além disso, gostaríamos de agradecer a Austin Krauss por seus *insights* sobre desenvolvimento de *software* no setor de games a partir da sua perspectiva como engenheiro de *software* sênior.

### **Agradecimentos especiais**

BRM: Estou muito agradecido pela oportunidade de trabalhar com Roger na nona edição deste livro. Enquanto fazia isso, meu filho, Benjamin, se tornou gerente de engenharia de *software*, enquanto Katherine, minha filha, usou sua formação artística para criar as figuras que aparecem neste livro. Estou muito contente de ver os adultos que se tornaram e adoro o meu tempo com as suas filhas (Isla, Emma e Thelma). Agradeço muito à minha esposa, Norma, pelo apoio entusiasmado que me deu quando preenchi meu tempo livre ao trabalhar nesta obra.

RSP: Assim como as edições deste livro evoluíram, meus filhos, Mathew e Michael, cresceram e se tornaram homens. Sua maturidade, caráter e sucesso me inspiraram. Após muitos anos seguindo nossas próprias carreiras, agora nós três trabalhamos juntos na empresa que fundamos em 2012. Nada jamais me deixou tão orgulhoso. Ambos têm suas próprias filhas, Maya e Lily, que dão início a mais uma geração. Por fim, à minha esposa, Barbara, ofereço meu amor e minha gratidão por ter tolerado as várias horas que dediquei ao trabalho e por ter me incentivado a fazer mais uma edição “do livro”.

*Bruce R. Maxim  
Roger S. Pressman*

# Sumário

---

**1** Software e engenharia de *software*

## **PARTE I O PROCESSO DE SOFTWARE**

---

**2** Modelos de processo

**3** Agilidade e processo

**4** Modelo de processo recomendado

**5** Aspectos humanos da engenharia de *software*

## **PARTE II MODELAGEM**

---

**6** Princípios que orientam a prática

**7** Entendendo os requisitos

**8** Modelagem de requisitos: Uma abordagem recomendada

**9** Conceitos de projeto

**10** Projeto de arquitetura: Uma abordagem recomendada

**11** Projeto de componentes

**12** Projeto da experiência do usuário

**13** Projeto para mobilidade

**14** Projeto baseado em padrões

## **PARTE III    QUALIDADE E SEGURANÇA**

---

- 15** Conceitos de qualidade
- 16** Revisões: Uma abordagem recomendada
- 17** Garantia da qualidade de *software*
- 18** Engenharia de segurança de *software*
- 19** Teste de *software* – Nível de componentes
- 20** Teste de *software* – Nível de integração
- 21** Teste de *software* – Testes especializados para mobilidade
- 22** Gestão de configuração de *software*
- 23** Métricas e análise de *software*

## **PARTE IV    GERENCIAMENTO DE PROJETOS DE SOFTWARE**

---

- 24** Conceitos de gerenciamento de projeto
- 25** Criando um plano de *software* viável
- 26** Gestão de riscos
- 27** Uma estratégia para suporte de *software*

## **PARTE V    TÓPICOS AVANÇADOS**

---

- 28** Melhoria do processo de *software*
- 29** Tendências emergentes na engenharia de *software*

## **30** Comentários finais

Apêndice 1 - Introdução à UML

Apêndice 2 - Ciência de dados para engenheiros de  
*software*

Referências

# **Sumário detalhado**

---

## **1 SOFTWARE E ENGENHARIA DE SOFTWARE**

---

1.1 A natureza do *software*

1.1.1 Definição de *software*

1.1.2 Domínios de aplicação de *software*

1.1.3 *Software* legado

1.2 Definição da disciplina

1.3 O processo de *software*

1.3.1 A metodologia do processo

1.3.2 Atividades de apoio

1.3.3 Adaptação do processo

1.4 A prática da engenharia de *software*

1.4.1 A essência da prática

1.4.2 Princípios gerais

1.5 Como tudo começa

1.6 Resumo

Problemas e pontos a ponderar

## **PARTE I      O PROCESSO DE *SOFTWARE***

---

### **2    MODELOS DE PROCESSO**

---

- 2.1 Um modelo de processo genérico
  - 2.2 Definição de uma atividade metodológica
  - 2.3 Identificação de um conjunto de tarefas
  - 2.4 Avaliação e aperfeiçoamento de processos
  - 2.5 Modelos de processo prescritivo
    - 2.5.1 O modelo cascata
    - 2.5.2 Modelo de processo de prototipação
    - 2.5.3 Modelo de processo evolucionário
    - 2.5.4 Modelo de Processo Unificado
  - 2.6 Produto e processo
  - 2.7 Resumo
- Problemas e pontos a ponderar

### **3    AGILIDADE E PROCESSO**

---

- 3.1 O que é agilidade?
- 3.2 Agilidade e o custo das mudanças
- 3.3 O que é processo ágil?
  - 3.3.1 Princípios da agilidade
  - 3.3.2 A política do desenvolvimento ágil

### 3.4 *Scrum*

- 3.4.1 Equipes e artefatos do *Scrum*
- 3.4.2 Reunião de planejamento do *sprint*
- 3.4.3 Reunião diária do *Scrum*
- 3.4.4 Reunião de revisão do *sprint*
- 3.4.5 Retrospectiva do *sprint*
- 3.5 Outros *frameworks* ágeis
  - 3.5.1 O *framework* XP
  - 3.5.2 Kanban
  - 3.5.3 DevOps
- 3.6 Resumo
- Problemas e pontos a ponderar

## 4 MODELO DE PROCESSO RECOMENDADO

---

- 4.1 Definição dos requisitos
- 4.2 Projeto de arquitetura preliminar
- 4.3 Estimativa de recursos
- 4.4 Construção do primeiro protótipo
- 4.5 Avaliação do protótipo
- 4.6 Decisão go/no-go
- 4.7 Evolução do protótipo
  - 4.7.1 Escopo do novo protótipo
  - 4.7.2 Construção de novos protótipos
  - 4.7.3 Teste dos novos protótipos
- 4.8 Disponibilização do protótipo
- 4.9 Manutenção do *software*

#### 4.10 Resumo

Problemas e pontos a ponderar

---

## **5 ASPECTOS HUMANOS DA ENGENHARIA DE SOFTWARE**

---

5.1 Características de um engenheiro de *software*

5.2 A psicologia da engenharia de *software*

5.3 A equipe de *software*

5.4 Estruturas de equipe

5.5 O impacto das mídias sociais

5.6 Equipes globais

5.7 Resumo

Problemas e pontos a ponderar

---

## **PARTE II MODELAGEM**

---

## **6 PRINCÍPIOS QUE ORIENTAM A PRÁTICA**

---

6.1 Princípios fundamentais

6.1.1 Princípios que orientam o processo

6.1.2 Princípios que orientam a prática

6.2 Princípios das atividades metodológicas

6.2.1 Princípios da comunicação

6.2.2 Princípios do planejamento

6.2.3 Princípios da modelagem

- 6.2.4 Princípios da construção
- 6.2.5 Princípios da disponibilização
- 6.3 Resumo
- Problemas e pontos a ponderar

## **7 ENTENDENDO OS REQUISITOS**

---

- 7.1 Engenharia de requisitos
  - 7.1.1 Concepção
  - 7.1.2 Levantamento
  - 7.1.3 Elaboração
  - 7.1.4 Negociação
  - 7.1.5 Especificação
  - 7.1.6 Validação
  - 7.1.7 Gerenciamento de requisitos
- 7.2 Estabelecimento da base de trabalho
  - 7.2.1 Identificação de envolvidos
  - 7.2.2 Reconhecimento de diversos pontos de vista
  - 7.2.3 Trabalho em busca da colaboração
  - 7.2.4 Questões iniciais
  - 7.2.5 Requisitos não funcionais
  - 7.2.6 Rastreabilidade
- 7.3 Levantamento de requisitos
  - 7.3.1 Coleta colaborativa de requisitos
  - 7.3.2 Cenários de uso
  - 7.3.3 Artefatos do levantamento de requisitos
- 7.4 Desenvolvimento de casos de uso

7.5 Construção do modelo de análise

7.5.1 Elementos do modelo de análise

7.5.2 Padrões de análise

7.6 Negociação de requisitos

7.7 Monitoramento de requisitos

7.8 Validação de requisitos

7.9 Resumo

Problemas e pontos a ponderar

## **8 MODELAGEM DE REQUISITOS: UMA ABORDAGEM RECOMENDADA**

---

8.1 Análise de requisitos

8.1.1 Filosofia e objetivos gerais

8.1.2 Regras práticas para a análise

8.1.3 Princípios da modelagem de requisitos

8.2 Modelagem baseada em cenários

8.2.1 Atores e perfis de usuário

8.2.2 Criação de casos de uso

8.2.3 Documentação de casos de uso

8.3 Modelagem baseada em classes

8.3.1 Identificação de classes de análise

8.3.2 Definição de atributos e operações

8.3.3 Modelos de classe da UML

8.3.4 Modelagem classe-responsabilidade-colaborador

8.4 Modelagem funcional

8.4.1 Uma visão procedural

8.4.2 Diagramas de sequência da UML

8.5 Modelagem comportamental

8.5.1 Identificação de eventos com o caso de uso

8.5.2 Diagramas de estados da UML

8.5.3 Diagramas de atividade da UML

8.6 Resumo

Problemas e pontos a ponderar

## **9 CONCEITOS DE PROJETO**

---

9.1 Projeto no contexto da engenharia de *software*

9.2 O processo de projeto

9.2.1 Diretrizes e atributos da qualidade de *software*

9.2.2 A evolução de um projeto de *software*

9.3 Conceitos de projeto

9.3.1 Abstração

9.3.2 Arquitetura

9.3.3 Padrões

9.3.4 Separação por interesses (por afinidades)

9.3.5 Modularidade

9.3.6 Encapsulamento[NT] de informações

9.3.7 Independência funcional

9.3.8 Refinamento gradual

9.3.9 Refatoração

9.3.10 Classes de projeto

9.4 O modelo de projeto

9.4.1 Princípios da modelagem de projetos

9.4.2 Elementos de projeto de dados

9.4.3 Elementos do projeto de arquitetura

9.4.4 Elementos do projeto de interface

9.4.5 Elementos do projeto de componentes

9.4.6 Elementos do projeto de implantação

9.5 Resumo

Problemas e pontos a ponderar

## **10 PROJETO DE ARQUITETURA: UMA ABORDAGEM RECOMENDADA**

---

10.1 Arquitetura de *software*

10.1.1 O que é arquitetura?

10.1.2 Por que a arquitetura é importante?

10.1.3 Descrições de arquitetura

10.1.4 Decisões de arquitetura

10.2 Agilidade e arquitetura

10.3 Estilos de arquitetura

10.3.1 Uma breve taxonomia dos estilos de arquitetura

10.3.2 Padrões de arquitetura

10.3.3 Organização e refinamento

10.4 Considerações sobre a arquitetura

10.5 Decisões de arquitetura

10.6 Projeto de arquitetura

10.6.1 Representação do sistema no contexto

- 10.6.2 Definição de arquétipos
- 10.6.3 Refinamento da arquitetura em componentes
- 10.6.4 Descrição das instâncias do sistema
- 10.7 Avaliação das alternativas de projeto de arquitetura
  - 10.7.1 Revisões da arquitetura
  - 10.7.2 Revisão de arquitetura baseada em padrões
  - 10.7.3 Verificação de conformidade da arquitetura
- 10.8 Resumo
- Problemas e pontos a ponderar

## **11 PROJETO DE COMPONENTES**

---

- 11.1 O que é componente?
  - 11.1.1 Uma visão orientada a objetos
  - 11.1.2 A visão tradicional
  - 11.1.3 Uma visão relacionada a processos
- 11.2 Projeto de componentes baseados em classes
  - 11.2.1 Princípios básicos de projeto
  - 11.2.2 Diretrizes para o projeto de componentes
  - 11.2.3 Coesão
  - 11.2.4 Acoplamento
- 11.3 Condução de projetos de componentes
- 11.4 Projetos de componentes especializados
  - 11.4.1 Projeto de componentes para WebApps

- 11.4.2 Projeto de componentes para aplicativos móveis
- 11.4.3 Projeto de componentes tradicionais
- 11.4.4 Desenvolvimento baseado em componentes
- 11.5 Refatoração de componentes
- 11.6 Resumo
- Problemas e pontos a ponderar

## **12 PROJETO DA EXPERIÊNCIA DO USUÁRIO**

---

- 12.1 Elementos do projeto da experiência do usuário
  - 12.1.1 Arquitetura da informação
  - 12.1.2 Projeto de interação do usuário
  - 12.1.3 Engenharia de usabilidade
  - 12.1.4 Projeto visual
- 12.2 As regras de ouro
  - 12.2.1 Deixar o usuário no comando
  - 12.2.2 Reduzir a carga de memória do usuário
  - 12.2.3 Tornar a interface consistente
- 12.3 Análise e projeto de interfaces
  - 12.3.1 Modelos de análise e projeto de interfaces
  - 12.3.2 O processo
- 12.4 Análise da experiência do usuário
  - 12.4.1 Pesquisa de usuário
  - 12.4.2 Modelagem de usuários
  - 12.4.3 Análise de tarefas

- 12.4.4 Análise do ambiente de trabalho
- 12.5 Projeto da experiência do usuário
- 12.6 Projeto de interface de usuário
  - 12.6.1 Aplicação das etapas para projeto de interfaces
  - 12.6.2 Padrões de projeto de interfaces do usuário
- 12.7 Avaliação de projeto
  - 12.7.1 Revisão do protótipo
  - 12.7.2 Testes de usuário
- 12.8 Usabilidade e acessibilidade
  - 12.8.1 Diretrizes de usabilidade
  - 12.8.2 Diretrizes de acessibilidade
- 12.9 UX e mobilidade de *software* convencional
- 12.10 Resumo
- Problemas e pontos a ponderar

## **13 PROJETO PARA MOBILIDADE**

---

- 13.1 Os desafios
  - 13.1.1 Considerações sobre o desenvolvimento
  - 13.1.2 Considerações técnicas
- 13.2 Ciclo de vida do desenvolvimento móvel
  - 13.2.1 Projeto de interface de usuário
  - 13.2.2 Lições aprendidas
- 13.3 Arquiteturas móveis
- 13.4 Aplicativos sensíveis ao contexto
- 13.5 Pirâmide de projeto para Web

- 13.5.1 Projeto de interface de WebApps
- 13.5.2 Projeto estético
- 13.5.3 Projeto de conteúdo
- 13.5.4 Projeto de arquitetura
- 13.5.5 Projeto de navegação
- 13.6 Projeto em nível de componentes
- 13.7 Mobilidade e qualidade do projeto
- 13.8 Melhores práticas do projeto de mobilidade
- 13.9 Resumo
- Problemas e pontos a ponderar

## **14 PROJETO BASEADO EM PADRÕES**

---

- 14.1 Padrões de projeto
  - 14.1.1 Tipos de padrões
  - 14.1.2 *Frameworks*
  - 14.1.3 Descrição de padrões
  - 14.1.4 Aprendizado de máquina e descoberta de padrões
- 14.2 Projeto de *software* baseado em padrões
  - 14.2.1 Contexto do projeto baseado em padrões
  - 14.2.2 Pense em termos de padrões
  - 14.2.3 Tarefas de projeto
  - 14.2.4 Construção de uma tabela para organização de padrões
  - 14.2.5 Erros comuns de projeto
- 14.3 Padrões de arquitetura
- 14.4 Padrões de projeto de componentes

- 14.5 Antipadrões
- 14.6 Padrões de projeto de interfaces do usuário
- 14.7 Padrões de projeto de mobilidade
- 14.8 Resumo
- Problemas e pontos a ponderar

## **PARTE III    QUALIDADE E SEGURANÇA**

---

### **15 CONCEITOS DE QUALIDADE**

---

- 15.1 O que é qualidade?
- 15.2 Qualidade de *software*
  - 15.2.1 Fatores de qualidade
  - 15.2.2 Avaliação quantitativa da qualidade
  - 15.2.3 Avaliação quantitativa da qualidade
- 15.3 O dilema da qualidade do *software*
  - 15.3.1 *Software* “bom o suficiente”
  - 15.3.2 Custo da qualidade
  - 15.3.3 Riscos
  - 15.3.4 Negligência e responsabilidade civil
  - 15.3.5 Qualidade e segurança
  - 15.3.6 O impacto das ações administrativas
- 15.4 Alcançando a qualidade de *software*
  - 15.4.1 Métodos de engenharia de *software*
  - 15.4.2 Técnicas de gerenciamento de projetos

15.4.3 Aprendizado de máquina e previsão de defeitos

15.4.4 Controle de qualidade

15.4.5 Garantia da qualidade

15.5 Resumo

Problemas e pontos a ponderar

## **16 REVISÕES: UMA ABORDAGEM RECOMENDADA**

---

16.1 Impacto de defeitos de *software* nos custos

16.2 Amplificação e eliminação de defeitos

16.3 Métricas de revisão e seu emprego

16.4 Critérios para tipos de revisão

16.5 Revisões informais

16.6 Revisões técnicas formais

16.6.1 A reunião de revisão

16.6.2 Relatório de revisão e manutenção de registros

16.6.3 Diretrizes de revisão

16.7 Avaliações *post-mortem*

16.8 Revisões ágeis

16.9 Resumo

Problemas e pontos a ponderar

## **17 GARANTIA DA QUALIDADE DE SOFTWARE**

---

17.1 Plano de fundo

- 17.2 Elementos de garantia de qualidade de *software*
- 17.3 Processos da SQA e características do produto
- 17.4 Tarefas, metas e métricas da SQA
  - 17.4.1 Tarefas da SQA
  - 17.4.2 Metas, atributos e métricas
- 17.5 Abordagens formais da SQA
- 17.6 Estatística da garantia da qualidade de *software*
  - 17.6.1 Um exemplo genérico
  - 17.6.2 Seis Sigma para engenharia de *software*
- 17.7 Confiabilidade de *software*
  - 17.7.1 Medidas de confiabilidade e disponibilidade
  - 17.7.2 Uso da inteligência artificial para modelagem da confiabilidade
  - 17.7.3 Segurança do *software*
- 17.8 Os padrões de qualidade ISO 90007
- 17.9 O plano de SQA
- 17.10 Resumo
- Problemas e pontos a ponderar

## **18 ENGENHARIA DE SEGURANÇA DE *SOFTWARE***

---

Contribuição de Nancy Mead Carnegie Mellon University *Software Engineering Institute*

- 18.1 Por que a engenharia de segurança de *software* é importante

- 18.2 Modelos do ciclo de vida da segurança
- 18.3 Atividades do ciclo de vida do desenvolvimento seguro
- 18.4 Engenharia de requisitos de segurança
  - 18.4.1 SQUARE
  - 18.4.2 O processo SQUARE
- 18.5 Casos de mau uso e abuso e padrões de ataque
- 18.6 Análise de risco de segurança
- 18.7 Modelagem de ameaças, priorização e mitigação
- 18.8 Superfície de ataque
- 18.9 Codificação segura
- 18.10 Medição
- 18.11 Modelos de maturidade e melhoria do processo de segurança
- 18.12 Resumo
- Problemas e pontos a ponderar

## **19 TESTE DE SOFTWARE – NÍVEL DE COMPONENTES**

---

- 19.1 Uma abordagem estratégica do teste de *software*
  - 19.1.1 Verificação e validação
  - 19.1.2 Organizando o teste de *software*
  - 19.1.3 Visão global
  - 19.1.4 Critérios de “Pronto”
- 19.2 Planejamento e manutenção de registros

- 19.2.1 O papel do scaffolding
- 19.2.2 Eficácia dos custos dos testes
- 19.3 Projeto de caso de teste
  - 19.3.1 Requisitos e casos de uso
  - 19.3.2 Rastreabilidade
- 19.4 Teste caixa-branca
  - 19.4.1 Teste de caminho básico
  - 19.4.2 Teste de estrutura de controle
- 19.5 Teste caixa-preta
  - 19.5.1 Teste de interface
  - 19.5.2 Particionamento de equivalência
  - 19.5.3 Análise de valor limite
- 19.6 Teste orientado a objetos
  - 19.6.1 Teste de conjunto
  - 19.6.2 Teste comportamental
- 19.7 Resumo
- Problemas e pontos a ponderar

## **20 TESTE DE SOFTWARE – NÍVEL DE INTEGRAÇÃO**

---

- 20.1 Fundamentos do teste de *software*
  - 20.1.1 Teste caixa-preta
  - 20.1.2 Teste caixa-branca
- 20.2 Teste de integração
  - 20.2.1 Integração descendente
  - 20.2.2 Integração ascendente
  - 20.2.3 Integração contínua
  - 20.2.4 Artefatos do teste de integração

- 20.3 Inteligência artificial e testes de regressão
- 20.4 Teste de integração em contexto orientado a objetos
  - 20.4.1 Projeto de caso de teste baseado em falhas3
  - 20.4.2 Projeto de caso de teste baseado em cenários
- 20.5 Teste de validação
- 20.6 Padrões de teste
- 20.7 Resumo
- Problemas e pontos a ponderar

## **21 TESTE DE SOFTWARE – TESTES ESPECIALIZADOS PARA MOBILIDADE**

---

- 21.1 Diretrizes para testes móveis
- 21.2 As estratégias de teste
- 21.3 Questões de teste da experiência do usuário
  - 21.3.1 Teste de gestos
  - 21.3.2 Entrada por teclado virtual
  - 21.3.3 Entrada e reconhecimento de voz
  - 21.3.4 Alertas e condições extraordinárias
- 21.4 Teste de aplicações para Web
- 21.5 As estratégias de teste para a Web
  - 21.5.1 Teste de conteúdo
  - 21.5.2 Teste de interface
  - 21.5.3 Testes de navegação
- 21.6 Internacionalização

- 21.7 Teste de segurança
- 21.8 Teste de desempenho
- 21.9 Teste em tempo real
- 21.10 Testes para sistemas de inteligência artificial (IA)
  - 21.10.1 Testes estáticos e dinâmicos
  - 21.10.2 Teste baseado em modelo
- 21.11 Teste de ambientes virtuais
  - 21.11.1 Teste de usabilidade
  - 21.11.2 Teste de acessibilidade
  - 21.11.3 Teste de jogabilidade
- 21.12 Teste da documentação e dos recursos de ajuda
- 21.13 Resumo
- Problemas e pontos a ponderar

## **22 GESTÃO DE CONFIGURAÇÃO DE *SOFTWARE***

---

- 22.1 Gerenciamento de configuração de *software*
  - 22.1.1 Um cenário SCM
  - 22.1.2 Elementos de um sistema de gestão de configuração
  - 22.1.3 Referenciais
  - 22.1.4 Itens de configuração de *software*
  - 22.1.5 Gestão de dependências e alterações
- 22.2 O repositório de SCM
  - 22.2.1 Características gerais e conteúdo
  - 22.2.2 Características do SCM
- 22.3 Sistemas de controle de versão

- 22.4 Integração contínua
- 22.5 O processo de gestão de alterações
  - 22.5.1 Controle de alterações
  - 22.5.2 Gestão de impacto
  - 22.5.3 Auditoria de configuração
  - 22.5.4 Relatório de status
- 22.6 Mobilidade e gestão de alterações ágil
  - 22.6.1 Controle eletrônico de alterações
  - 22.6.2 Gestão de conteúdo
  - 22.6.3 Integração e publicação
  - 22.6.4 Controle de versão
  - 22.6.5 Auditoria e relatório
- 22.7 Resumo
- Problemas e pontos a ponderar

## **23 MÉTRICAS E ANÁLISE DE SOFTWARE**

---

- 23.1 Medição de *software*
  - 23.1.1 Medidas, métricas e indicadores
  - 23.1.2 Atributos de métricas de *software* eficazes
- 23.2 Análise de dados de *software*
- 23.3 Métricas de produto
  - 23.3.1 Métricas para o modelo de requisitos
  - 23.3.2 Métricas de projeto para *software* convencional
  - 23.3.3 Métricas de projeto para *software* orientado a objetos
  - 23.3.4 Métricas de projeto de interface de usuário

- 23.3.5 Métricas para código-fonte
- 23.4 Métricas para teste
- 23.5 Métricas para manutenção
- 23.6 Métricas de processo e de projeto
- 23.7 Medição de *software*
- 23.8 Métricas para qualidade de *software*
- 23.9 Estabelecimento de um programa de métricas de *software*
- 23.10 Resumo
- Problemas e pontos a ponderar

---

## **PARTE IV    GERENCIAMENTO DE PROJETOS DE SOFTWARE**

---

### **24 CONCEITOS DE GERENCIAMENTO DE PROJETO**

---

- 24.1 O espectro de gerenciamento
  - 24.1.1 As pessoas
  - 24.1.2 O produto
  - 24.1.3 O processo
  - 24.1.4 O projeto
- 24.2 As pessoas
  - 24.2.1 Os envolvidos
  - 24.2.2 Líderes de equipe
  - 24.2.3 A equipe de *software*

- 24.2.4 Questões de comunicação e coordenação
  - 24.3 Produto
    - 24.3.1 Escopo do *software*
    - 24.3.2 Decomposição do problema
  - 24.4 Processo
    - 24.4.1 Combinando o produto e o processo
    - 24.4.2 Decomposição do processo
  - 24.5 Projeto
  - 24.6 O princípio W 5 HH
  - 24.7 Práticas vitais
  - 24.8 Resumo
- Problemas e pontos a ponderar

## **25 CRIANDO UM PLANO DE SOFTWARE VIÁVEL**

---

- 25.1 Comentários sobre as estimativas
- 25.2 O processo de planejamento do projeto
- 25.3 Escopo e viabilidade do *software*
- 25.4 Recursos
  - 25.4.1 Recursos humanos
  - 25.4.2 Recursos de *software* reutilizáveis
  - 25.4.3 Recursos ambientais
- 25.5 Análise de dados e estimativa do projeto de *software*
- 25.6 Técnicas de estimativa e decomposição
  - 25.6.1 Dimensionamento do *software*
  - 25.6.2 Estimativa baseada em problema

- 25.6.3 Um exemplo de estimativa baseada em LOC
- 25.6.4 Um exemplo de estimativa baseada em FP
- 25.6.5 Um exemplo de estimativa baseada em processo
- 25.6.6 Um exemplo de estimativa usando pontos de caso de uso
- 25.6.7 Harmonizando estimativas
- 25.6.8 Estimativa para desenvolvimento ágil
- 25.7 Cronograma de projeto
- 25.7.1 Princípios básicos
- 25.7.2 Relação entre pessoas e esforço
- 25.8 Definição do conjunto de tarefas do projeto
  - 25.8.1 Um exemplo de conjunto de tarefas
  - 25.8.2 Refinamento das tarefas principais
- 25.9 Definição de uma rede de tarefas
- 25.10 Cronograma
  - 25.10.1 Gráfico de Gantt
  - 25.10.2 Acompanhamento do cronograma
- 25.11 Resumo
- Problemas e pontos a ponderar

## **26 GESTÃO DE RISCOS**

---

- 26.1 Estratégias de risco reativas *versus* proativas
- 26.2 Riscos de *software*

- 26.3 Identificação do risco
  - 26.3.1 Avaliação do risco geral do projeto
  - 26.3.2 Componentes e fatores de risco
- 26.4 Previsão de risco
  - 26.4.1 Desenvolvimento de uma tabela de riscos
  - 26.4.2 Avaliação do impacto do risco
- 26.5 Refinamento do risco
- 26.6 Mitigação, monitoramento e gestão de riscos (RMMM)
- 26.7 O plano RMMM
- 26.8 Resumo
- Problemas e pontos a ponderar

## **27 UMA ESTRATÉGIA PARA SUPORTE DE SOFTWARE**

---

- 27.1 Suporte de *software*
- 27.2 Manutenção de *software*
  - 27.2.1 Tipos de manutenção
  - 27.2.2 Tarefas de manutenção
  - 27.2.3 Engenharia reversa
- 27.3 Suporte proativo de *software*
  - 27.3.1 Uso de análise de *software*
  - 27.3.2 O papel das mídias sociais
  - 27.3.3 Custo do suporte
- 27.4 Refatoração
  - 27.4.1 Refatoração de dados
  - 27.4.2 Refatoração de código

- 27.4.3 Refatoração da arquitetura
- 27.5 Evolução de *software*
- 27.5.1 Análise de inventário
- 27.5.2 Reestruturação dos documentos
- 27.5.3 Engenharia reversa
- 27.5.4 Refatoração de código
- 27.5.5 Refatoração de dados
- 27.5.6 Engenharia direta
- 27.6 Resumo
- Problemas e pontos a ponderar

## **PARTE V TÓPICOS AVANÇADOS**

---

### **28 MELHORIA DO PROCESSO DE *SOFTWARE***

---

- 28.1 O que é SPI?
- 28.1.1 Abordagens para SPI
- 28.1.2 Modelos de maturidade
- 28.1.3 A SPI é para todos?
- 28.2 O processo de SPI
- 28.2.1 Avaliação e análise de lacunas
- 28.2.2 Educação e treinamento
- 28.2.3 Seleção e justificação
- 28.2.4 Instalação/migração
- 28.2.5 Avaliação
- 28.2.6 Gestão de riscos para SPI
- 28.3 O CMMI

28.4 Outros *frameworks* SPI

28.4.1 SPICE

28.4.2 TickIT Plus

28.5 Retorno sobre investimento em SPI

28.6 Tendências da SPI

28.7 Resumo

Problemas e pontos a ponderar

## **29 TENDÊNCIAS EMERGENTES NA ENGENHARIA DE *SOFTWARE***

---

29.1 Evolução da tecnologia

29.2 A engenharia de *software* como disciplina

29.3 Observação de tendências na engenharia de *software*

29.4 Identificação das “tendências leves”

29.4.1 Gestão da complexidade

29.4.2 *Software* aberto

29.4.3 Requisitos emergentes

29.4.4 O mix de talentos

29.4.5 Blocos básicos de *software*

29.4.6 Mudança na percepção de “valor”

29.4.7 Código aberto

29.5 Rumos da tecnologia

29.5.1 Tendências de processo

29.5.2 O grande desafio

29.5.3 Desenvolvimento colaborativo

29.5.4 Engenharia de requisitos

- 29.5.5 Desenvolvimento de *software* dirigido por modelo
- 29.5.6 Engenharia de *software* baseada em busca
- 29.5.7 Desenvolvimento guiado por teste
- 29.6 Tendências relacionadas a ferramentas
- 29.7 Resumo
- Problemas e pontos a ponderar

## **30 COMENTÁRIOS FINAIS**

---

- 30.1 A importância do *software* – revisitada
- 30.2 Pessoas e a maneira como constroem sistemas
- 30.3 Descoberta de conhecimento
- 30.4 A visão em longo prazo
- 30.5 A responsabilidade do engenheiro de *software*
- 30.6 Comentário final de RSP

## **Apêndice 1 INTRODUÇÃO À UML1**

## **Apêndice 2 CIÊNCIA DE DADOS PARA ENGENHEIROS DE SOFTWARE**

## **Referências**

# 1

---

## **Software e engenharia de software**

### **Conceitos-chave**

domínios de aplicação

curvas de defeitos

metodologia

princípios gerais

*software* legado

princípios

solução de problemas

*CasaSegura*

*software*,

definição de

natureza do

processo

perguntas sobre

engenharia de *software*,

definição de

camadas

prática

atividades de apoio

deterioração

Depois de me mostrar a construção mais recente de um dos *games* de tiro em primeira pessoa mais populares do mundo, o jovem desenvolvedor riu.

“Você não joga, né?”, ele perguntou.

Eu sorri. “Como adivinhou?”

O jovem estava de bermuda e camiseta. Sua perna balançava para cima e para baixo como um pistão, queimando a tensa energia que parecia ser comum entre seus colegas.



## Panorama

**O que é?** Software de computador é o produto que profissionais de software desenvolvem e ao qual dão suporte por muitos anos. Esses artefatos incluem programas executáveis em computador de qualquer porte ou arquitetura. A engenharia de software abrange um processo, um conjunto de métodos (práticas) e uma série 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 que têm contato com o mundo industrializado o utilizam. Os engenheiros de software aplicam o processo de engenharia de software.

**Por que é importante?** A engenharia de software é importante porque nos capacita para o desenvolvimento de sistemas complexos dentro do prazo e com alta qualidade. Ela impõe disciplina a um

trabalho que pode se tornar caótico, mas também permite que as pessoas produzam *software* de computador adaptado à sua abordagem, da maneira mais conveniente às suas necessidades.

**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.

**Qual é o artefato?** Do ponto de vista de um engenheiro de software, artefato é um conjunto de programas, conteúdo (dados) e outros artefatos que apoiam o *software* de computador. Porém, do ponto de vista do usuário, o artefato é uma ferramenta ou um produto que, de alguma forma, torna a vida dele melhor.

### **Como garantir que o trabalho foi realizado corretamente?**

Leia o restante deste livro, escolha as ideias aplicáveis ao *software* que você desenvolver e use-as em seu trabalho.

“Porque, se jogasse”, ele disse, “estaria muito mais empolgado. Você acabou de ver nosso mais novo produto, algo que nossos clientes matariam para ver... sem trocadilhos”.

Estávamos na área de desenvolvimento de uma das empresas de *games* mais bem-sucedidas do planeta. Ao longo dos anos, as gerações anteriores do *game* que ele demonstrou venderam mais de 50 milhões de cópias e geraram uma receita bilionária.

“Então, quando essa versão estará no mercado?”, perguntei.

Ele encolheu os ombros. “Em cerca de cinco meses. Ainda temos muito trabalho a fazer.”

Ele era responsável pela jogabilidade e pela funcionalidade de inteligência artificial de um aplicativo que abrangia mais de três milhões de linhas de código.

“Vocês usam técnicas de engenharia de *software*?”, perguntei, meio que esperando sua risada e sua resposta negativa.

Ele fez uma pausa e pensou por uns instantes. Então, lentamente, fez que sim com a cabeça. “Adaptamos às nossas necessidades, mas, claro, usamos.”

“Onde?”, perguntei, sondando.

“Geralmente, nosso problema é traduzir os requisitos que os criativos nos dão.”

“Os criativos?”, interrompi.

“Você sabe, os caras que projetam a história, os personagens, todas as coisas que tornam o jogo um sucesso. Temos de pegar o que eles nos dão e produzir um conjunto de requisitos técnicos que nos permita construir o jogo.”

“E depois os requisitos são fixados?”

Ele encolheu os ombros. “Precisamos ampliar e adaptar a arquitetura da versão anterior do jogo e criar um novo produto. Temos de criar código a partir dos requisitos, testá-lo com construções diárias e fazer muitas coisas que seu livro recomenda.”

“Conhece meu livro?” Eu estava sinceramente surpreso.

“Claro, usei na faculdade. Há muita coisa lá.”

“Falei com alguns de seus colegas aqui, e eles são mais céticos a respeito do material de meu livro.”

Ele franziu as sobrancelhas. “Olha, não somos um departamento de TI nem uma empresa aeroespacial, então, temos de adaptar o que você defende. Mas o resultado é o mesmo – precisamos criar um produto de alta qualidade, e o único jeito de conseguirmos isso sempre é adaptar nosso próprio subconjunto de técnicas de engenharia de *software*.”

“E como seu subconjunto mudará com o passar dos anos?”

Ele fez uma pausa como se estivesse pensando no futuro. “Os *games* vão se tornar maiores e mais complexos, com certeza. E nossos cronogramas de desenvolvimento vão ser mais apertados, à medida que a concorrência surgir. Lentamente, os próprios jogos nos obrigarão a aplicar um pouco mais de disciplina de desenvolvimento. Se não fizermos isso, estaremos mortos.”

\*\*\*\*\*

*Software* de computador continua a ser a tecnologia mais importante no cenário mundial. E é, também, um ótimo exemplo da lei das consequências não intencionais. Há 60 anos, ninguém poderia prever que o *software* se tornaria uma tecnologia indispensável para negócios, ciência e engenharia; que o *software* viabilizaria a criação de novas tecnologias (p. ex., engenharia genética e nanotecnologia), a extensão de tecnologias existentes (p. ex., telecomunicações) e a mudança radical nas tecnologias mais antigas (p. ex., a mídia); que o *software* se tornaria a força motriz por trás da revolução do computador pessoal; que aplicativos de *software* seriam comprados pelos consumidores por meio de seus dispositivos móveis; que o *software* evoluiria lentamente de produto para serviço, à medida que empresas de *software* “sob demanda” oferecessem funcionalidade imediata (*just-in-time*) via um navegador Web; que uma empresa de *software* se tornaria maior e mais influente do que todas as empresas da era industrial; ou que uma vasta rede comandada por *software* evoluiria e modificaria tudo: de pesquisa em bibliotecas a compras feitas pelos consumidores, de discursos políticos a comportamentos de namoro entre jovens e (não tão jovens) adultos.

Conforme aumenta a importância do *software*, a comunidade da área tenta criar 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 domínio de aplicação específico (p. ex., projeto e implementação de *sites*); outras são focadas em um campo de tecnologia (p. ex.,

sistemas orientados a objetos ou programação orientada a aspectos); outras, ainda, são de bases amplas (p. ex., sistemas operacionais, como o Linux). Entretanto, nós ainda temos de desenvolver uma tecnologia de *software* que faça tudo isso – e 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*.

Para desenvolver um *software* que esteja preparado para enfrentar os desafios do século XXI, devemos admitir alguns fatos:

- *Software* está profundamente incorporado em quase todos os aspectos de nossas vidas. O número de pessoas interessadas nos recursos e nas funções oferecidas por determinada aplicação<sup>1</sup> tem crescido significativamente. É preciso fazer um esforço conjunto para compreender o problema antes de desenvolver uma solução de *software*.
- Os requisitos de tecnologia da informação demandados por pessoas, empresas e órgãos governamentais estão mais complexos a cada ano. Hoje, equipes grandes desenvolvem programas de computador. *Software* sofisticado, outrora implementado em um ambiente computacional independente e previsível, hoje está incorporado em tudo, de produtos eletrônicos de consumo a equipamentos médicos e veículos autônomos. *Projetar se tornou uma atividade essencial*.
- Pessoas, negócios e governos dependem, cada vez mais, de *software* para a tomada de decisões estratégicas e táticas, assim como para controle e para operações cotidianas. Se o *software* falhar, as pessoas e as principais empresas poderão ter desde

pequenos inconvenientes até falhas catastróficas. *O software deve apresentar qualidade elevada.*

- À medida que o valor de uma aplicação específica aumenta, a probabilidade é de que sua base de usuários e sua longevidade também cresçam. Conforme sua base de usuários e seu tempo em uso forem aumentando, a demanda por adaptação e aperfeiçoamento também vai aumentar. *O software deve ser passível de manutenção.*

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

## 1.1 A natureza do *software*

---

Hoje, o *software* tem um duplo papel. Ele é um produto e 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. Seja localizado em um dispositivo móvel, em um computador de mesa, na nuvem ou em um *mainframe* ou máquina autônoma, o *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 representação de realidade aumentada derivada de dados obtidos de dezenas de fontes independentes e, então, sobreposta ao mundo real. 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 (p. ex., transações financeiras de um indivíduo) de modo que possam ser mais úteis

em determinado contexto; gerencia informações comerciais para aumentar a competitividade; fornece um portal para redes mundiais de informação (p. ex., Internet); e proporciona os meios para obter informações sob todas as suas formas. Ele também propicia um veículo que pode ameaçar a privacidade pessoal e um portal que permite a pessoas mal-intencionadas cometer crimes.

O papel do *software* passou por uma mudança significativa no decorrer dos últimos 60 anos. Aperfeiçoamentos consideráveis no desempenho do *hardware*, mudanças profundas nas arquiteturas computacionais, um vasto aumento na capacidade de memória e armazenamento e uma ampla variedade de opções exóticas 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 e projetar sistemas robustos.

Atualmente, uma enorme indústria de *software* tornou-se fator dominante nas economias do mundo industrializado. Equipes de especialistas em *software*, cada equipe 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 hoje, quando os modernos sistemas computacionais são desenvolvidos:<sup>2</sup>

- Por que a conclusão de 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 realizando a manutenção de programas existentes?
- Por que ainda temos dificuldades de medir o progresso de desenvolvimento e a manutenção de um *software*?

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

### 1.1.1 Definição de *software*

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

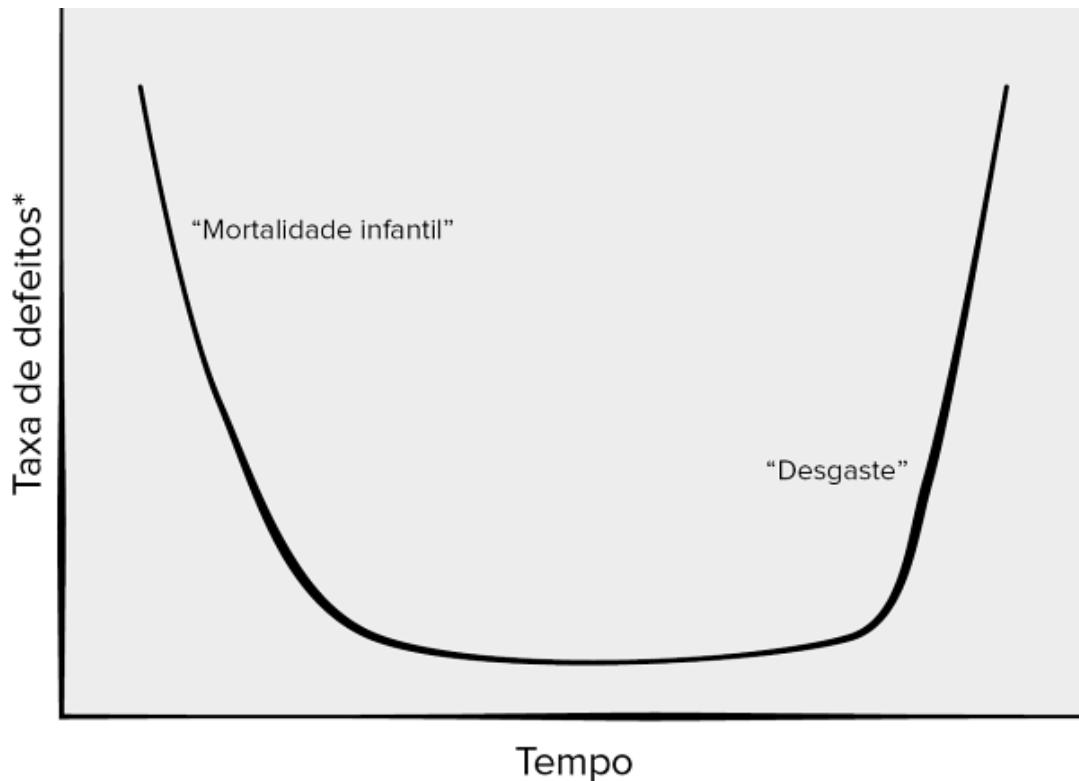
Uma descrição de *software* em um livro-texto poderia ser a seguinte:

*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 quanto 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 a compreensão do que é o *software*. Para conseguir isso, é importante examinar as características do *software* que o tornam diferente de outras coisas que os seres humanos constroem. *Software* é mais um elemento de sistema lógico do que físico. Portanto, o *software* tem uma característica fundamental que o torna consideravelmente diferente do *hardware*: o *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 (espera-se que seja 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 fatores maléficos do ambiente. Resumindo, o *hardware* começa a se *desgastar*.

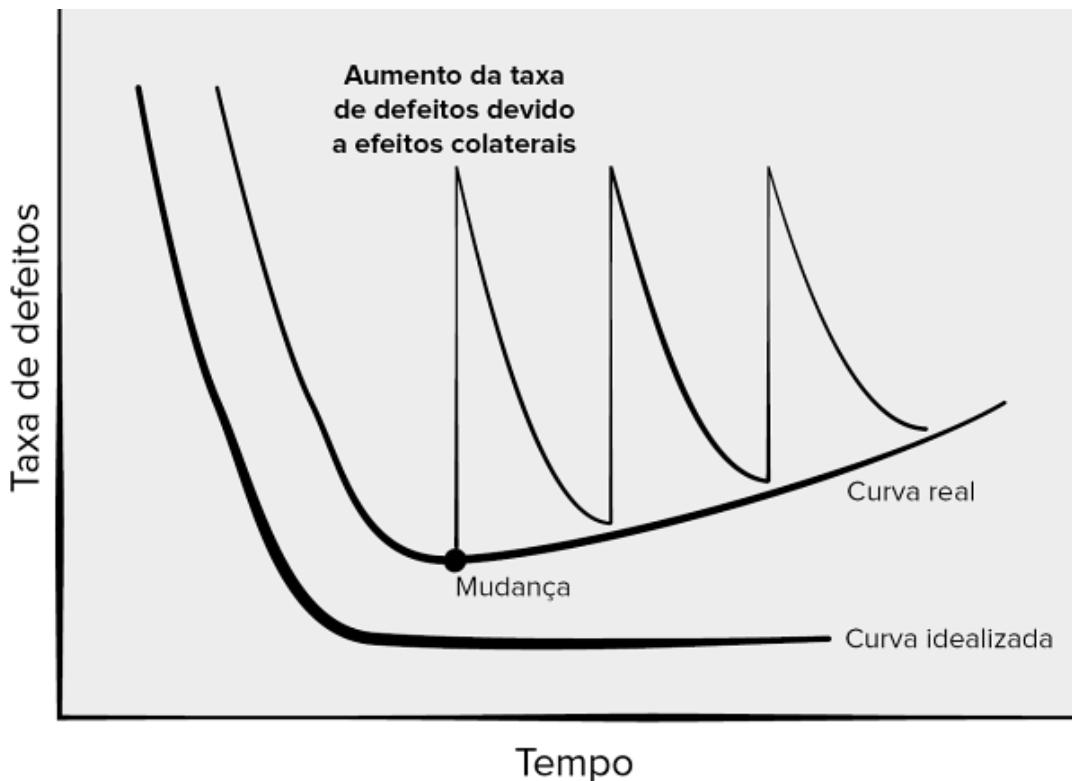


**Figura 1.1<sup>[NT]</sup>**

Curva de defeitos para *hardware*.

O *software* não é suscetível aos fatores maléficos do ambiente que causam o desgaste do *hardware*. 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 defeitos serão corrigidos, e a curva se achata, 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 deteriora!



**Figura 1.2**  
Curva de defeitos para *software*.

Essa aparente contradição pode ser elucidada pela curva real apresentada na Figura 1.2. Durante sua vida,<sup>3</sup> o *software* passará por alterações. À medida que elas ocorrem, é provável que sejam introduzidos erros, ocasionando o aumento da curva de taxa de defeitos, conforme mostrado na “curva real” (Figura 1.2). Antes que a curva possa retornar à taxa estável original, outra alteração é requisitada, aumentando a curva novamente. Lentamente, o nível mínimo da taxa começa a aumentar – o *software* está deteriorando devido à modificação.

Outro aspecto do 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 complexidade consideravelmente maior do que as de manutenção de *hardware*.

### **1.1.2 Domínios de aplicação de *software***

Atualmente, 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 (p. ex., compiladores, editores e utilitários para gerenciamento de arquivos) processam estruturas de informação complexas, mas determinadas.<sup>4</sup> Outras aplicações de sistema (p. ex., componentes de sistema operacional, *drivers*, *software* de rede, processadores de telecomunicações) processam dados amplamente indeterminados.

***Software de aplicação.*** Programas independentes 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.

***Software de engenharia/científico.*** Uma ampla variedade de programas de “cálculo em massa” que abrangem astronomia, vulcanologia, análise de estresse automotivo, dinâmica orbital, projeto auxiliado por computador, hábitos de consumo, análise genética e meteorologia, entre outros.

**Software embarcado.** Residente num produto ou sistema e utilizado para implementar e controlar características e funções para o usuário e para o próprio sistema. Executa funções limitadas e específicas (p. ex., controle do painel de um forno micro-ondas) ou fornece função significativa e capacidade de controle (p. ex., funções digitais de automóveis, como controle do nível de combustível, painéis de controle e sistemas de freio).

**Software para linha de produtos.** Composto por componentes reutilizáveis e projetado para prover capacidades específicas de utilização por muitos clientes diferentes. Software para linha de produtos pode se concentrar em um mercado hermético e limitado (p. ex., produtos de controle de inventário) ou lidar com consumidor de massa.

**Aplicações Web/aplicativos móveis.** Esta categoria de software voltada às redes abrange uma ampla variedade de aplicações, contemplando aplicativos voltados para navegadores, computação em nuvem, computação baseada em serviços e software residente em dispositivos móveis.

**Software de inteligência artificial.** Faz uso de heurísticas<sup>5</sup> para solucionar problemas complexos que não são passíveis de computação ou de análise direta. Aplicações nessa área incluem: robótica, sistemas de tomada de decisão, reconhecimento de padrões (de imagem e de voz), aprendizado de máquina, 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 um jovem engenheiro de software trabalhar em um programa mais velho do 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 esta geração facilite o trabalho dos futuros engenheiros de *software*.

### 1.1.3 **Software legado**

Centenas de milhares de programas de computador caem em um dos sete amplos domínios de aplicação discutidos na subseção anterior. Alguns deles são *software* de ponta. 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 às 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.

Essas mudanças podem criar um efeito colateral extra muito presente em *software legado* – a *baixa qualidade*.<sup>6</sup> Às vezes, os sistemas legados têm projetos inextensíveis, código de difícil entendimento, documentação deficiente ou inexistente, casos de teste e resultados que nunca foram documentados, um histórico de alterações mal gerenciado – a lista pode ser bastante longa. Ainda assim, esses sistemas dão suporte a “funções vitais de negócio e são indispensáveis para ele”. O que fazer?

A única resposta adequada 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 funciona de forma confiável, ele não está “quebrado” e não precisa ser “consertado”. Entretanto, com o passar do tempo, esses sistemas evoluem devido a uma ou mais das razões a seguir:

- 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 capaz de funcionar com outros bancos de dados ou com sistemas mais modernos.
- O *software* deve ter a sua arquitetura alterada para torná-lo viável dentro de um ambiente computacional em evolução.

Quando essas modalidades de evolução ocorrem, um sistema legado deve passar por reengenharia para que permaneça viável no futuro. O objetivo da engenharia de *software* moderna é “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 interagir e cooperar uns com os outros” [Day99].

## 1.2 Definição da disciplina

---

O Instituto de engenheiros eletricistas e eletrônicos (IEEE) [IEE17] elaborou a seguinte definição para engenharia de *software*:

**Engenharia de *software*:** 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*.

Entretanto, uma abordagem “sistemática, 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. Como ilustra a 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, ou Seis Sigma, e filosofias similares<sup>7</sup> promovem

uma cultura de aperfeiçoamento contínuo de processos. É essa cultura que, no final das contas, leva a abordagens cada vez mais eficazes na engenharia de *software*. A pedra fundamental que sustenta a engenharia de *software* é o foco na qualidade.



**Figura 1.3**  
Camadas da engenharia de *software*.

A base da 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 *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 artefatos (modelos, documentos, dados, relatórios, formulários, etc.), são estabelecidos marcos, a qualidade é garantida e as mudanças são geridas de forma apropriada.

Os *métodos* da engenharia de *software* fornecem as informações técnicas para desenvolver o *software*. Os métodos envolvem uma ampla variedade de tarefas, que incluem comunicação, análise de requisitos, modelagem de projeto, construção de programa, testes e suporte. Os métodos da engenharia de *software* se baseiam em um conjunto de princípios básicos que governam cada área da

tecnologia e incluem atividades de modelagem e outras técnicas descritivas.

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 utilizadas por outra, é estabelecido um sistema para o suporte ao desenvolvimento de *software*, denominado *engenharia de software com o auxílio do computador*.

## 1.3 O processo de *software*

---

Um *processo* é um conjunto de atividades, ações e tarefas realizadas na criação de algum artefato. Uma *atividade* se esforça para atingir um objetivo amplo (p. ex., comunicar-se com os envolvidos) e é utilizada independentemente do domínio de aplicação, do tamanho do projeto, da complexidade dos esforços ou do grau de rigor com que a engenharia de *software* será aplicada. Uma *ação* (p. ex., projeto de arquitetura) envolve um conjunto de tarefas que resultam em um artefato de *software* fundamental (p. ex., um modelo arquitetural). Uma *tarefa* se concentra em um objetivo pequeno, porém bem-definido (p. ex., 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 vão utilizá-lo.

### 1.3.1 A metodologia do processo

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 metodológicas* 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*) aplicáveis a todo o processo de *software*. Uma metodologia de processo genérica para engenharia de *software* compreende cinco atividades:

**Comunicação.** Antes que qualquer trabalho técnico possa começar, é de importância fundamental se comunicar e colaborar com o cliente (e outros envolvidos).<sup>8</sup> A intenção é entender os objetivos dos envolvidos para o projeto e reunir requisitos que ajudem a definir os recursos e as funções do *software*.

**Planejamento.** Qualquer jornada complicada pode ser simplificada com o auxílio de 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 serem conduzidas, os riscos prováveis, os recursos que serão necessários, os artefatos a serem produzidos e um cronograma de trabalho.

**Modelagem.** Seja 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” para 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, ele cria modelos para entender melhor as necessidades do *software* e o projeto que vai atender a essas necessidades.

**Construção.** O que se projeta deve ser construído. Essa atividade combina geração de código (manual ou automatizada) e testes necessários para revelar erros na codificação.

**Entrega.** O *software* (como uma entidade completa ou como um incremento parcialmente concluído) é entregue ao cliente, que avalia o produto 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 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 caso, 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 entrega são aplicados repetidamente, sejam quantas forem as iterações do projeto. Cada iteração produzirá um *incremento de software* que disponibilizará uma parte dos recursos e das funcionalidades do *software*. A cada incremento, o *software* se torna cada vez mais completo.

### **1.3.2 Atividades de apoio**

As atividades metodológicas do processo de engenharia de *software* são complementadas por diversas *atividades de apoio*. De modo geral, as atividades de apoio são aplicadas por todo um projeto de *software* e ajudam uma equipe de *software* a gerenciar e a controlar o andamento, a qualidade, as alterações e os riscos. 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 eles 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 dos envolvidos; 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 capacidade de reutilização.** Define critérios para a reutilização de artefatos (inclusive componentes de *software*) e estabelece mecanismos para a obtenção de componentes reutilizáveis.

**Preparo e produção de artefatos de software.** Engloba as atividades necessárias para criar artefatos, como modelos, documentos, logs, formulários e listas.

Cada uma dessas atividades de apoio será discutida em detalhes mais adiante.

### **1.3.3 Adaptação do processo**

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

- Fluxo geral de atividades, ações e tarefas e suas interdependências.
- Até que ponto as ações e tarefas são definidas dentro de cada atividade da metodologia.
- Até que ponto artefatos de *software* são identificados e exigidos.
- Modo de aplicar as atividades de garantia da 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 envolvidos).
- 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 um grau de detalhamento considerável.

## 1.4 A prática da engenharia de *software*

---

A Seção 1.3 introduziu 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 entrega** –, 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 vai adquirir um conhecimento básico dos princípios e conceitos genéricos que se aplicam às atividades de uma metodologia.<sup>9</sup>

### 1.4.1 A essência da prática

No livro clássico *How to Solve It* (*A Arte de Resolver Problemas*), escrito antes de os computadores modernos existirem, George Polya [Pol45] descreveu 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*).
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 um problema nos é apresentado. 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 perguntas simples:

- *Quem tem interesse na solução do problema?* Ou seja, quem são os envolvidos?
- *Quais são as incógnitas?* Quais 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?

**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 semelhantes anteriormente? Existem padrões que são reconhecíveis em uma possível solução? Existe algum software que implemente os dados, as funções e as características necessárias?
- Algum problema semelhante 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?

**Leve o plano adiante.** 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, mas o “planejamento” permitirá que continuemos sem nos perder.

- A solução é adequada ao plano? O código-fonte pode ser atribuído ao modelo de projeto?
- Todas as partes componentes da solução estão provavelmente corretas? O projeto e o código foram revistos ou, melhor ainda, as 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 da solução? Foi implementada uma estratégia de testes razoável?
- A solução produz resultados adequados aos dados, às funções e às características necessários? O software foi validado em relação a todas as solicitações dos envolvidos?

Não é surpresa que grande parte dessa metodologia consista em bom senso. De fato, é possível afirmar que uma abordagem de bom

senso à engenharia de *software* jamais o levará ao erro.

## 1.4.2 Princípios gerais

O dicionário define a palavra *princípio* como “uma importante afirmação ou lei básica 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 (p. ex., **comunicação**), e outros ainda destacam as ações de engenharia de *software* (p. ex., projeto de arquitetura) ou tarefas técnicas (p. ex., criar 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 por que 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:<sup>10</sup>

### Primeiro princípio: a razão de existir

Um sistema de *software* existe por um motivo: *agregar valor para seus usuários*. Todas as decisões devem ser tomadas com esse princípio em mente. Antes de especificar um requisito 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 agrega 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: *não complique!*)

Existem muitos fatores a considerar em qualquer trabalho de projeto. *Todo projeto deve ser o mais simples possível, mas não simplista*. Este 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, os projetos mais elegantes normalmente são os mais simples. Simples também não significa “gambiarra”. Na verdade, muitas vezes são necessárias muitas reflexões e trabalho em várias iterações para simplificar. A contrapartida é um *software* mais fácil de manter e menos propenso a erros.

### **Terceiro princípio:** *mantenha a visão*

*Uma visão clara é essencial para o sucesso de um projeto de software.* Sem uma integridade conceitual, corre-se o risco de transformar o projeto em uma colcha de retalhos de projetos incompatíveis, unidos por parafusos inadequados. Comprometer a visão arquitetural 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*

*Sempre especifique, projete, documente e implemente ciente de que mais alguém terá de entender o que você está fazendo.* O público para qualquer produto de desenvolvimento de *software* é potencialmente grande. Especifique tendo como objetivo os usuários. Projete tendo em mente os implementadores. Codifique se preocupando com aqueles que deverão manter e ampliar o sistema. Alguém terá de depurar o código que você escreveu, e isso o torna um usuário de seu código. Facilitando o trabalho de todas essas pessoas, você agrega maior valor ao sistema.

### **Quinto princípio:** *esteja aberto para o futuro*

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 em vez de em anos. Contudo, os verdadeiros

sistemas de *software* com “qualidade industrial” devem durar muito mais. Para serem bem-sucedidos nisso, esses sistemas precisam estar prontos para se adaptar a essas e outras 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 respostas possíveis, criando sistemas que resolvam o problema geral, não apenas o específico.<sup>11</sup>

### **Sexto princípio:** *planeje com antecedência, visando a reutilização*

A reutilização economiza tempo e esforço.<sup>12</sup> Alcançar um alto grau de reutilização 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 uma grande vantagem do uso de tecnologias orientadas a objetos. Contudo, o retorno desse investimento não é automático. *Planejar com antecedência para a reutilização 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, o mais menosprezado. *Pensar bem e de forma clara antes de agir quase sempre produz melhores resultados.* Quando se analisa alguma coisa, provavelmente ela sairá correta. Ganha-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 faz parte de um sistema, seu valor aflora. Aplicar os seis primeiros princípios exige intensa reflexão, para a qual as recompensas em potencial 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 sistemas complexos baseados em computador seriam eliminadas.

## 1.5 Como tudo começa

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 ampliar 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.

### Casa segura<sup>13</sup>



#### 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 trabalhando em 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, se conecta a quase tudo. Usa o protocolo sem fio 802.11n. Permite que accessemos saídas de dispositivos sem o emprego de fios. Acreditamos que nos levará a uma geração inteiramente nova de produtos.

**Joe:** Você concorda, Mal?

**Mal:** Sim. Na verdade, com as vendas tão baixas 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 de CasaSegura. Eles usam uma nova interface sem fio e oferecem a pequenos empresários e proprietários de residências 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 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 qual é o ponto principal.

**Mal:** PCs e tablets estão em mais de 70% dos lares nos EUA. Se pudermos acertar no preço, pode ser um aplicativo incrível. Ninguém mais tem nosso dispositivo sem fio... ele é exclusivo! Estaremos dois anos à frente de nossos concorrentes. E as receitas? Algo em torno de US\$ 30 a 40 milhões no segundo ano.

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

Exceto por uma rápida referência, o *software* mal foi mencionado como parte da conversa. Ainda assim, o *software* vai decretar o sucesso ou o fracasso da linha de produtos *CasaSegura*. O trabalho de engenharia só terá êxito se o *software* do *CasaSegura* tiver êxito. O mercado só vai aceitar o produto 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 deste livro.

## 1.6 Resumo

---

*Software* é o elemento-chave na evolução de produtos e sistemas baseados em computador e é uma das mais importantes tecnologias no cenário mundial. Ao longo dos últimos 60 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 do orçamento estabelecidos.

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

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 entrega, e elas se aplicam a todos os projetos de *software*. A prática da engenharia de *software* é uma atividade de resolução de problemas que segue um conjunto de princípios básicos. À medida que você aprende mais sobre engenharia de *software*, você começará a entender por que esses princípios devem ser considerados quando iniciamos qualquer projeto de *software*.

## Problemas e pontos a ponderar

---

- 1.1 Dê, no mínimo, mais cinco exemplos de como a lei das consequências não intencionais se aplica ao *software* de computador.
- 1.2 Forneça uma série de exemplos (positivos e negativos) que indiquem o impacto do *software* em nossa sociedade.
- 1.3 Dê suas próprias respostas para as cinco perguntas feitas no início da Seção 1.1. Discuta-as com seus colegas.
- 1.4 Muitas aplicações modernas mudam frequentemente – antes de serem apresentadas ao usuário e depois da primeira versão ser colocada em uso. Sugira algumas maneiras de construir um *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 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, em que a falha de um programa de computador poderia causar um grande dano em termos econômicos ou humanos.
- 1.7 Descreva uma metodologia de processo 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? Explique.
- 1.8 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 da metodologia?

---

**1** Neste livro, mais adiante, chamaremos tais pessoas de “envolvidos”.

**2** Em um excelente livro de ensaios sobre o setor de *software*, Tom DeMarco [DeM95] contesta. Segundo ele: “Em vez de perguntar por que o *software* custa tanto, precisamos começar perguntando: ‘O que fizemos para que o *software* atual custe tão pouco?’ A resposta a essa pergunta nos ajudará a continuar com o extraordinário nível de realização que tem distinguido a indústria de *software*”.

**[defeitos]** N. de R.T.: Os defeitos do *software* nem sempre se manifestam como falha, geralmente devido a tratamentos dos erros decorrentes desses defeitos pelo *software*. Esses conceitos serão 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.

**3** De fato, desde o momento em que o desenvolvimento começa, e muito antes de a primeira versão ser entregue, podem ser solicitadas mudanças por uma variedade de diferentes envolvidos.

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

**5** O uso de heurísticas é uma abordagem à solução de problemas que emprega um método prático, ou “regrinhas”, para o qual não há nenhuma garantia de perfeição, mas que é suficiente para a tarefa do momento.

**6** Nesse caso, a qualidade é julgada em termos da 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.

**7** A gestão da qualidade e as metodologias relacionadas são discutidas ao longo da Parte Três deste livro.

**8** Um *envolvido* é qualquer pessoa que tenha interesse no êxito de um projeto – executivos, usuários, engenheiros de *software*, pessoal de suporte, etc. Rob Thomsett ironiza: “Envolvido (*stakeholder*) é uma pessoa que segura (*hold*) uma estaca (*stake*) grande e pontiaguda... Se você não cuidar de seus envolvidos, sabe bem onde essa estaca vai parar”.

**9** Você deve rever seções relevantes contidas neste capítulo à medida que discutirmos os métodos de engenharia de *software* e as atividades de apoio específicas mais adiante neste livro.

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

**11** Esse conselho pode ser perigoso se levado ao extremo. Projetar para o “problema geral” algumas vezes exige comprometer o desempenho e pode tornar ineficientes as soluções específicas.

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

**13** O projeto *Casa Segura* 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 fictícios, porém as situações e os problemas são reais.