

## PROCESSOS DE SOFTWARE

**N**esta seção você aprenderá sobre o processo que fornece uma metodologia para a prática da engenharia de software. As questões abaixo são tratadas nos capítulos seguintes:

- O que é um processo de software?
- Quais são as atividades metodológicas genéricas presentes em todos os processos de software?
- Como os processos são modelados e o que são padrões de processo?
- O que são modelos de processo prescritivo e quais são seus pontos fortes e fracos?
- Por que agilidade é um lema no trabalho da engenharia de software moderna?
- O que é desenvolvimento de software ágil e como ele difere dos modelos de processos mais tradicionais?

Respondidas tais questões, você estará mais bem preparado para compreender o contexto no qual a prática da engenharia de software é aplicada.

## 2

## MODELOS DE PROCESSO

### CONECTOS- -CHAVE

conjunto de tarefas .....	55
desenvolvimento baseado em componentes.....	69
modelo de métodos formais.....	69
modelo de processo gênero.....	53
modelos concorrentes .....	67
modelos de processo evolucionário .....	62
modelos de processo incremental.....	61
modelos de processo prescritivo .....	58
padrões de processo.....	55
processo de software em equipe.....	75
processo de software pessoal.....	74
processo unificado. .	71

**E**m um livro fascinante, que nos dá a visão de um economista acerca do software e da engenharia de software, Howard Baetjer, Jr. [Bae98], comenta o processo de software:

Pelo fato de software, como todo capital, ser conhecimento incorporado, e pelo fato de esse conhecimento ser, inicialmente, disperso, tácito, latente e em considerável medida, incompleto, o desenvolvimento de software é um processo de aprendizado social. Esse processo é um diálogo no qual o conhecimento, que deverá tornar-se o software, é coletado, reunido e incorporado ao software. Tal processo possibilita a interação entre usuários e projetistas, entre usuários e ferramentas em evolução e entre projetistas e ferramentas em evolução (tecnologia). Trata-se de um processo iterativo no qual a própria ferramenta em evolução serve como meio de comunicação, com cada nova rodada do diálogo extraíndo mais conhecimento útil das pessoas envolvidas.

De fato, construir software é um processo de aprendizado social iterativo e o resultado, algo que Baetjer denominaria “capital de software”, é a incorporação do conhecimento coletado, filtrado e organizado conforme se desenvolve o processo.

Mas o que é exatamente um processo de software do ponto de vista técnico? No contexto desse livro, *processo de software* é definido como uma metodologia para as atividades, ações e tarefas necessárias para desenvolver um software de alta qualidade. “Processo” é sinônimo de engenharia de software? A resposta é “sim e não”. Um processo de software define a abordagem adotada conforme um software é elaborado pela engenharia. Mas a engenharia de software também engloba tecnologias que fazem parte do processo — métodos técnicos e ferramentas automatizadas.

Mais importante, a engenharia de software é realizada por pessoas criativas e com amplos conhecimentos e que devem adaptar um processo de software maduro, de forma que fique apropriado aos produtos desenvolvidos e às demandas de seu mercado.

### PANORAMA

**O que é?** Quando se trabalha na elaboração de um produto ou sistema, é importante seguir uma série de passos previsíveis — um roteiro que ajude a criar um resultado de alta qualidade e dentro do prazo estabelecido. O roteiro é denominado “processo de software”.

**Quem realiza?** Os engenheiros de software e seus gerentes adaptam o processo às suas necessidades e então o seguem. Os solicitantes do software têm um papel a desempenhar no processo de definição, construção e teste do software.

**Por que ele é importante?** Porque propicia estabilidade, controle e organização para uma atividade que pode, sem controle, tornar-se bastante caótica. Entretanto, uma abordagem de engenharia de software moderna deve ser “ágil”. Deve demandar apenas atividades, controles e produtos de trabalho que sejam apropriados para a equipe do projeto e para o produto a ser produzido.

**Quais são as etapas envolvidas?** O processo adotado depende do software a ser desenvolvido. Um determinado processo pode ser apropriado para um software do sistema “aviônico” de uma aeronave, enquanto um processo totalmente diferente pode ser indicado para a criação de um site.

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

**Como garantir que o trabalho foi feito corretamente?** Há muitos mecanismos de avaliação dos processos de software que possibilitam às organizações determinarem o nível de “maturidade” de seu processo de software. Entretanto, a qualidade, o cumprimento de prazos e a viabilidade a longo prazo do produto que se desenvolve são os melhores indicadores da eficácia do processo utilizado.

## 2.1 UM MODELO DE PROCESSO GÊNERICO

No Capítulo 1, processo foi definido como um conjunto de atividades de trabalho, ações e tarefas realizadas quando algum artefato de software deve ser criado. Cada uma dessas atividades, ações e tarefas alocam-se dentro de uma metodologia ou modelo que determina seu relacionamento com o processo e seu relacionamento umas com as outras.

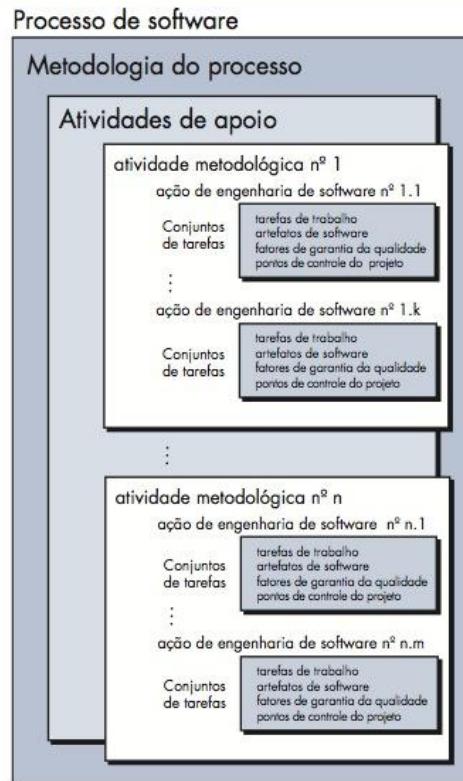
O processo de software é representado esquematicamente na Figura 2.1. De acordo com a figura, cada atividade metodológica é composta por um conjunto de ações de engenharia de software. Cada ação é definida por um *conjunto de tarefas*, o qual identifica as tarefas de trabalho a ser completadas, os artefatos de software que serão produzidos, os fatores de garantia da qualidade que serão exigidos e os marcos utilizados para indicar progresso.

Como discutido no Capítulo 1, uma metodologia de processo genérica para engenharia de software estabelece cinco atividades metodológicas: **comunicação, planejamento, modelagem, construção e entrega**. Além disso, um conjunto de atividades de apoio (*umbrella activities*) são aplicadas ao longo do processo, como o acompanhamento e controle do projeto, a administração de riscos, a garantia da qualidade, o gerenciamento das configurações, as revisões técnicas e outras.

### PONTO-CHAVE

A hierarquia de trabalho técnico, dentro do processo de software, consiste em: atividades, ações abrangentes, compostas por tarefas.

**FIGURA 2.1**  
Uma metodologia do processo de software



*"Achamos que desenvolvedores de software não percebem uma verdade essencial: a maioria das organizações não sabe o que faz. Elas acham que sabem, mas não sabem."*

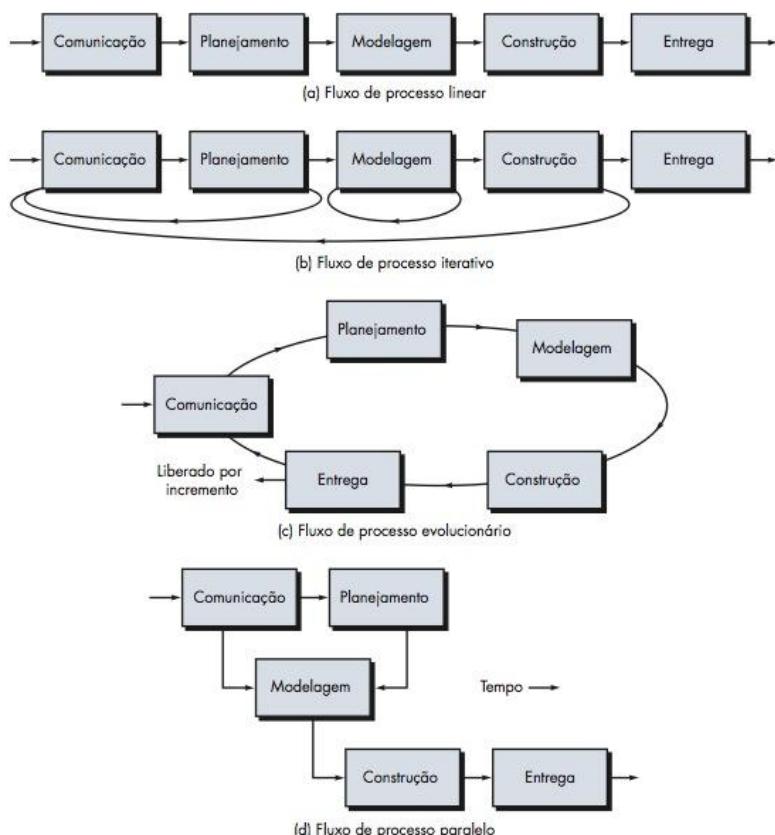
Tom DeMarco

Deve-se notar que um importante aspecto do processo de software ainda não foi discutido. Esse aspecto — chamado *fluxo de processo* — descreve como são organizadas as atividades metodológicas, bem como as ações e tarefas que ocorrem dentro de cada atividade em relação à sequência e ao tempo, como ilustrado na Figura 2.2.

Um *fluxo de processo linear* executa cada uma das cinco atividades metodológicas em sequência, começando com a de comunicação e culminando com a do emprego (Figura 2.2a). Um *fluxo de processo iterativo* repete uma ou mais das atividades antes de prosseguir para a seguinte (Figura 2.2b). Um *fluxo de processo evolucionário* executa as atividades de uma forma “circular”. Cada volta pelas cinco atividades conduz a uma versão mais completa do software (Figura 2.2c). Um *fluxo de processo paralelo* (Figura 2.2d) executa uma ou mais atividades em paralelo com outras atividades (por exemplo, a modelagem para um aspecto do software poderia ser executada em paralelo com a construção de um outro aspecto do software).

FIGURA 2.2

Fluxo de processo



### 2.1.1 Definindo atividade metodológica

Embora tenham sido descritas cinco atividades metodológicas e se tenha fornecido uma definição básica de cada uma delas no Capítulo 1, uma equipe de software precisa de muito mais informações antes de poder executar apropriadamente qualquer uma dessas atividades como parte do processo de software. Consequentemente, depara-se com uma questão-chave: *Que ações são apropriadas para uma atividade metodológica, uma vez fornecidos a natureza do problema a ser solucionado, as características das pessoas que estarão executando o trabalho e os interessados que estarão propondo o projeto?*

Como uma atividade metodológica é modificada de acordo com as alterações da natureza do projeto?

Para um pequeno projeto de software solicitado por uma única pessoa (numa localidade longínqua) com necessidades simples e objetivas, a atividade de comunicação pode resumir-se a pouco mais de um telefonema para o devido solicitante. Portanto, a única ação necessária é uma conversação telefônica, e as tarefas de trabalho (o conjunto de tarefas) que essa ação envolve são:

1. Contactar o interessado via telefone.
2. Discutir as necessidades e tomar notas.
3. Organizar anotações em uma breve relação de requisitos, por escrito.
4. Enviar um e-mail para o interessado para revisão e aprovação.

Se o projeto fosse consideravelmente mais complexo, com muitos interessados, cada qual com um conjunto de necessidades diferentes (por vezes conflitantes), a atividade de comunicação poderia ter seis ações distintas (descritas no Capítulo 5): *inserção, elucidação, elaboração, negociação, especificação e validação*. Cada uma dessas ações de engenharia de software contempla muitas tarefas de trabalho e uma série de diferentes artefatos.

#### PONTO-CHAVE

Projetos diferentes demandam conjuntos de tarefas diferentes. A equipe de software escolhe o conjunto de tarefas fundamental no problema e nas características do projeto.

### 2.1.2 Identificação de um conjunto de tarefas

Referindo-se novamente à Figura 2.1, cada ação de engenharia de software (por exemplo, elucidação, uma ação associada à atividade de comunicação) pode ser representada por vários e diferentes conjuntos de tarefas — cada um constituído por uma gama de tarefas de trabalho de engenharia de software, artefatos relativos, fatores de garantia da qualidade e pontos de controle do projeto. Deve-se escolher um conjunto de tarefas mais adequado às necessidades do projeto e às características da equipe. Isso significa que uma ação de engenharia de software pode ser adaptada às necessidades específicas do projeto de software e às características da equipe.

### 2.1.3 Padrões de processos

Toda equipe de desenvolvimento encontra problemas à medida que avança no processo de software. Seria útil se soluções comprovadas estivessem prontamente à disposição da equipe, de modo que os problemas pudessem ser localizados e resolvidos rapidamente. Um *padrão de processo*<sup>1</sup> descreve um problema de processo encontrado durante o trabalho de engenharia de software, identificando o ambiente onde foi encontrado e sugerindo uma ou mais soluções comprovadas para o problema. Em termos mais genéricos, um padrão de processo fornece um modelo (template) [Amb98] — um método consistente para descrever soluções de problemas no contexto do processo de software. Combinando padrões, uma equipe conseguirá solucionar problemas e elaborar um processo que melhor atenda às necessidades de um projeto.

Padrões podem ser definidos em qualquer nível de abstração.<sup>2</sup> Em alguns casos, um padrão poderia ser utilizado para descrever um problema (e sua solução) associado ao modelo de processo completo (por exemplo, prototipação). Em outras situações, os padrões podem ser usados para descrever um problema (e sua solução) associado a uma atividade metodológica (por

1 Uma discussão detalhada sobre padrões é apresentada no Capítulo 12.

2 Os padrões são aplicáveis a várias atividades de engenharia de software. Padrões de análise, de projeto e de testes são discutidos nos Capítulos 7, 9, 10, 12 e 14. Padrões e “antipadrões” para atividades de gerenciamento de projetos são discutidos na Parte 4 deste livro.

O que é padrão de processo?

“A repetição de padrões é algo bem diferente da repetição de partes. De fato, as partes diferentes serão únicas, pois os padrões são os mesmos.”

Christopher Alexander

**Conjunto de tarefas**



Um conjunto de tarefas define o verdadeiro trabalho a ser feito para se atingir os objetivos de uma ação de engenharia de software. Por exemplo, elucidação (mais comumente denominada "levantamento de requisitos") é uma importante ação de engenharia de software que ocorre durante a atividade de comunicação. A meta do levantamento de requisitos é compreender o que os vários interessados esperam do software a ser desenvolvido.

Para um projeto pequeno, relativamente simples, o conjunto de tarefas para levantamento das necessidades seria semelhante ao seguinte:

1. Fazer uma lista dos envolvidos no projeto.
2. Fazer uma reunião informal com todos os interessados.
3. Solicitar para cada interessado uma lista com as características e funções necessárias.
4. Discutir sobre os requisitos e construir uma lista final.
5. Organizar os requisitos por grau de prioridade.
6. Destacar pontos de incertezas.

Para um projeto de software maior e mais complexo, necessita-se de um conjunto diferente de tarefas. Tal conjunto pode incluir as seguintes tarefas de trabalho:

1. Fazer uma lista dos envolvidos no projeto.
2. Entrevistar separadamente cada um dos envolvidos para levantamento geral de suas expectativas e necessidades.

**INFORMAÇÕES**

3. Fazer uma lista preliminar das funções e características, com base nas informações fornecidas pelos interessados.
4. Agendar uma série de reuniões facilitadoras para especificação de aplicações.
5. Promover reuniões.
6. Incluir cenários informais de usuários como parte de cada reunião.
7. Aprimorar os cenários de usuários, com base no feedback dos interessados.
8. Fazer uma lista revisada das necessidades dos interessados.
9. Empregar técnicas de aplicação de funções de qualidade para estabelecer graus de prioridade dos requisitos.
10. Agrupar os requisitos de modo que eles possam ser entregues incrementalmente.
11. Fazer um levantamento das limitações e restrições que serão aplicadas ao sistema.
12. Discutir sobre os métodos para validação do sistema.

Esses dois conjuntos de tarefas atingem o objetivo de "levantamento de necessidades", porém, são bem diferentes em relação aos graus de profundidade e formalidade. A equipe de software deve escolher o conjunto de tarefas que lhe possibilitará atingir o objetivo de cada ação, mantendo, inclusive, a qualidade e a agilidade.

exemplo, **planejamento**) ou uma ação dentro de uma atividade metodológica (por exemplo, estimativa de custos do projeto).

Ambler [Amb98] propôs um modelo para descrever um padrão de processo:

#### PONTO-CHAVE

Um modelo de padrões propicia um meio consistente para descrever um padrão.

**Nome do Padrão.** O padrão deve receber um nome significativo que o descreva no contexto do processo de software (por exemplo, **RevisõesTécnicas**).

**Forças.** Ambiente onde se encontram o padrão e as questões que tornam visível o problema e que poderiam afetar sua solução.

**Tipo.** É especificado o tipo de padrão. Ambler sugere três tipos:

1. **Padrão de estágio** — define um problema associado a uma atividade metodológica para o processo. Como uma atividade metodológica envolve múltiplas ações e tarefas de trabalho, um padrão de estágio engloba múltiplos padrões de tarefas (veja o próximo padrão) que são relevantes ao estágio (atividade metodológica). Podemos citar como um exemplo de padrão de estágio **EstabelecendoComunicação**. Esse padrão incorpora o padrão de tarefas **LevantamentodeNecessidades** e outros.
2. **Padrão de tarefas** — define um problema associado a uma ação de engenharia de software ou tarefa de trabalho relevante para a prática de engenharia de software bem-sucedida (por exemplo, **LevantamentodeNecessidades** é um padrão de tarefas).
3. **Padrão de fases** — define a sequência das atividades metodológicas que ocorrem dentro do processo, mesmo quando o fluxo geral de atividades é iterativo por natureza. Um exemplo de padrão de fases seria **ModeloEspiral** ou **Prototipação**.<sup>3</sup>

<sup>3</sup> Esses padrões de fases são discutidos na Seção 2.3.3.

**Contexto Inicial.** Descreve as condições sob as quais o padrão se aplica. Antes do início do padrão: (1) Que atividades organizacionais ou relacionadas à equipe já ocorreram? (2) Qual o estado inicial para o processo? (3) Que informação de engenharia de software ou de projeto já existe?

Por exemplo, o padrão **Planejamento** (um padrão de estágio) requer que: (1) clientes e engenheiros de software tenham estabelecido uma comunicação colaborativa; (2) Tenha ocorrido a finalização bem-sucedida de uma série de padrões de tarefas [especificados] para o padrão **Comunicação**; e (3) Sejam conhecidos o escopo do projeto, as necessidades básicas do negócio, bem como as restrições do projeto.

**Problema.** O problema específico a ser resolvido pelo padrão.

**Solução.** Descreve como implementar o padrão de forma bem-sucedida. Esta seção descreve como o estado inicial do processo (que existe antes de o padrão ser implementado) é modificado como consequência do início do padrão. Descreve também como as informações de engenharia de software ou de projeto que se encontram à disposição antes do início do padrão são transformadas como consequência da execução do padrão de forma bem-sucedida.

**Contexto Resultante.** Descreve as condições que resultarão assim que o padrão tiver sido implementado com êxito. Após a finalização do padrão:

- (1) Quais atividades organizacionais ou relacionadas à equipe devem ter ocorrido?
- (2) Qual é o estado de saída para o processo?
- (3) Quais informações de engenharia de software ou de projeto foram desenvolvidas?

**Padrões Relativos.** Fornece uma lista de todos os padrões de processo que estão diretamente relacionados ao processo em questão. Essa lista pode ser representada de forma hierárquica ou em alguma outra forma com diagramas. Por exemplo, o padrão de estágio **Comunicação** envolve os padrões de tarefas: **EquipeDeProjeto**, **DiretrizesColaborativas**, **IsolamentoDoEscopo**, **LevantamentoDeNecessidades**, **DescriçãoDasRestrições** e **CriaçãoDeCenários**.



### Um exemplo de padrão de processo

### INFORMAÇÕES

O padrão de processo sintetizado a seguir descreve uma abordagem que pode ser aplicada quando os interessados têm uma ideia geral do que precisa ser feito, mas estão incertos quanto aos requisitos específicos do software.

#### Nome do padrão. RequisitosImprecisos

**Intento.** Este padrão descreve uma abordagem voltada para a construção de um modelo (um protótipo) passível de ser avaliado iterativamente pelos interessados, num esforço para identificar ou solidificar os requisitos de software.

#### Tipo. Padrão de fase.

**Contexto inicial.** As condições seguintes devem ser atendidas antes de iniciar esse padrão: (1) interessados identificados; (2) forma de comunicação entre interessados e equipe de software já determinada; (3) principal problema de software a ser resolvido já identificado pelos interessados; (4) compreensão inicial do escopo do projeto, dos requisitos de negócios básicos e das restrições do projeto já atingida.

**Problema.** Os requisitos são vagos ou inexistentes, ainda assim há um reconhecimento claro de que existe um problema a

ser solucionado e este deve ser identificado utilizando-se uma solução de software. Os interessados não sabem o que querem, ou seja, eles não conseguem descrever os requisitos de software em detalhe.

**Solução.** Uma descrição do processo de prototipação poderia ser apresentada nesta etapa, mas é descrita posteriormente na Seção 2.3.3.

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

**Padrões associados.** Os seguintes padrões estão relacionados a esse padrão: **ComunicaçãoComCliente**, **ProjetoIterativo**, **DesenvolvimentoIterativo**, **AvaliaçãoDoCliente**, **ExtraçãoDeRequisitos**.

**Usos conhecidos e um exemplo.** A prototipação é recomendada quando as necessidades são incertas.

**Usos Conhecidos e Exemplos.** Indicam as instâncias específicas onde o padrão é aplicável. Por exemplo, **Comunicação** é obrigatória no início de todo projeto de software, é recomendável ao longo de todo o projeto de software e é obrigatória assim que a atividade de emprego estiver em andamento.

**WebRef**

Recursos completos para padrões de processo podem ser encontrados em [www.ambyssoft.com/processPatternsPage.html](http://www.ambyssoft.com/processPatternsPage.html).

Padrões de processo propiciam um mecanismo efetivo para localização de problemas associados a qualquer processo de software. Os padrões permitem que se desenvolva uma descrição de processo de forma hierárquica que se inicia com nível alto de abstração (um padrão de fases). A descrição é então refinada em um conjunto de padrões de estágio que descreve atividades metodológicas e são ainda mais refinadas de uma forma hierárquica, em padrões de tarefa mais detalhados para cada padrão de estágio. Uma vez que os padrões de processos tenham sido desenvolvidos, eles poderão ser reutilizados para a definição de variantes de processo — isto é, um modelo de processo personalizado pode ser definido por uma equipe de software usando os padrões como blocos de construção para o modelo de processo.

## 2.2 AVALIAÇÃO E APERFEIÇOAMENTO DE PROCESSOS

**PONTO-CHAVE**

Tentativas de avaliação para compreender o atual estado do processo de software com o intuito de aperfeiçoá-lo.

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

"As organizações de software apresentaram folhas significativas quanto à habilidade em capitalizar as experiências adquiridas nos projetos finalizados."

Nasa

A existência de um processo de software não garante que o software será entregue dentro do prazo, que estará de acordo com as necessidades do cliente ou que apresentará características técnicas que conduzirão a características de qualidade de longo prazo (Capítulos 14 e 16). Os padrões de processo devem ser combinados com uma prática de engenharia de software consistente (Parte 2 deste livro). Além disso, o próprio processo pode ser avaliado para assegurar que está de acordo com um conjunto de critérios de processo básicos comprovados como essenciais para uma engenharia de software de sucesso.<sup>4</sup>

Ao longo das últimas décadas foi proposta uma série de abordagens diferentes em relação à avaliação e ao aperfeiçoamento dos processos de software:

**SCAMPI (Standard CMMI Assessment Method for Process Improvement) — (Método Padrão CMMI de Avaliação para Aperfeiçoamento de Processo da CMMI):** fornece um modelo de avaliação do processo de cinco etapas, contendo cinco fases: início, diagnóstico, estabelecimento, atuação e aprendizado. O método SCAMPI usa o CMMI da SEI como base para avaliação [SEI00].

**CBA IPI (CMM — Based Appraisal for Internal Process Improvement) — (Avaliação para Aperfeiçoamento do Processo Interno baseada na CMM):** fornece uma técnica de diagnóstico para avaliar a maturidade relativa de uma organização de software; usa a CMM da SEI como base para a avaliação [Dun01].

**SPICE (ISO/IEC15504)** — padrão que define um conjunto de requisitos para avaliação do processo de software. A finalidade do padrão é auxiliar as organizações no desenvolvimento de uma avaliação objetiva da eficácia de um processo qualquer de software [ISO08].

**ISO 9001:2000 para Software** — padrão genérico aplicável a qualquer organização que queira aperfeiçoar a qualidade global de produtos, sistemas ou serviços fornecidos. Portanto, o padrão é aplicável diretamente a organizações e empresas de software [Ant06].

Uma discussão mais detalhada sobre métodos de avaliação de software e aperfeiçoamento de processo é apresentada no Capítulo 30.

## 2.3 MODELOS DE PROCESSO PRESCRITIVO

Originalmente, modelos de processo prescritivo foram propostos para trazer ordem ao caos existente na área de desenvolvimento de software. A história tem demonstrado que esses

<sup>4</sup> A CMMI [CMM07] da SEI descreve, de forma extremamente detalhada, as características de um processo de software e os critérios para o êxito de um processo.

modelos tradicionais proporcionaram uma considerável contribuição quanto à estrutura utilizable no trabalho de engenharia de software e forneceram um roteiro razoavelmente eficaz para as equipes de software. Entretanto, o trabalho de engenharia de software e o seu produto permanecem “à beira do caos”.

Em um intrigante artigo sobre o estranho relacionamento entre ordem e caos no mundo do software, Nogueira e seus colegas [Nog00] afirmam que

O limiar do caos é definido como “um estado natural entre ordem e caos, um grande compromisso entre estrutura e surpresa” [Kau95]. O limiar do caos pode ser visualizado como um estado instável, parcialmente estruturado... Instável porque é constantemente atraído para o caos ou para a ordem absoluta.

Temos uma tendência de pensar que ordem é o estado ideal da natureza. Isso pode ser um erro. Pesquisas... Defendem a teoria de que a operação longe do equilíbrio gera criatividade, processos auto-organizados e lucros crescentes [Roo96]. Ordem absoluta implica ausência de variabilidade, o que poderia ser uma vantagem em ambientes imprevisíveis. A mudança ocorre quando existe uma estrutura que permita que a mudança possa ser organizada, mas tal estrutura não deve ser tão rígida a ponto de impedir que a mudança ocorra. Por outro lado, caos em demasia pode tornar impossível a coordenação e a coerência. A falta de estrutura nem sempre implica desordem.

As implicações filosóficas desse argumento são significativas para a engenharia de software. Se os modelos de processos prescritivos<sup>5</sup> buscam ao máximo a estrutura e a ordem, seriam eles inapropriados para um mundo de software que prospera com as mudanças? E mais, se rejeitarmos os modelos de processo tradicionais (e a ordem implícita) e substituí-los por algo menos estruturado, tornaríamos impossível atingir a coordenação e a coerência no trabalho de software?

Não há respostas fáceis para essas questões, mas existem alternativas disponíveis para os engenheiros de software. Nas seções seguintes, examina-se a abordagem de processos prescritivos no qual a ordem e a consistência do projeto são questões dominantes. Denominam-se “prescritivos” porque prescrevem um conjunto de elementos de processo — atividades metodológicas, ações de engenharia de software, tarefas, produtos de trabalho, garantia da qualidade e mecanismos de controle de mudanças para cada projeto. Cada modelo de processo também prescreve um fluxo de processo (também denominado *fluxo de trabalho*) — ou seja, a forma pela qual os elementos do processo estão inter-relacionados.

Todos os modelos de processo de software podem acomodar as atividades metodológicas genéricas descritas no Capítulo 1, porém, cada um deles dá uma ênfase diferente a essas atividades e define um fluxo de processo que invoca cada atividade metodológica (bem como tarefas e ações de engenharia de software) de forma diversa.

### 2.3.1 O modelo cascata

há casos em que os requisitos de um problema são bem compreendidos — quando o trabalho flui da **comunicação** ao **emprego** de forma relativamente linear. Essa situação ocorre algumas vezes quando adaptações ou aperfeiçoamentos bem definidos precisam ser feitos em um sistema existente (por exemplo, uma adaptação em software contábil exigida devido a mudanças nas normas governamentais). Pode ocorrer também em um número limitado de novos esforços de desenvolvimento, mas apenas quando os requisitos estão bem definidos e são razoavelmente estáveis.

O *modelo cascata*, algumas vezes chamado *ciclo de vida clássico*, sugere uma abordagem sequencial e sistemática<sup>6</sup> para o desenvolvimento de software, começando com o levantamento de necessidades por parte do cliente, avançando pelas fases de planejamento, modelagem, construção, emprego e culminando no suporte contínuo do software concluído (Figura 2.3).

<sup>5</sup> Os modelos de processos prescritivos são, algumas vezes, conhecidos como modelos de processos “tradicionais”.

<sup>6</sup> Embora o modelo cascata proposto por Winston Royce [Roy70] previsse os “feedback loops”, a vasta maioria das organizações que aplica esse modelo de processo os trata como se fossem estritamente lineares.

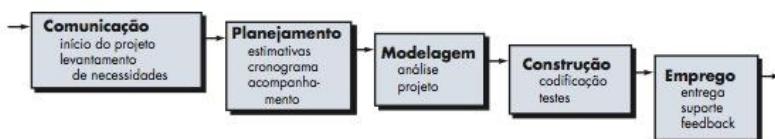
“Se o processo estiver correto, os resultados falarão por si mesmos.”  
Takashi Osada

### PONTO-CHAVE

Os modelos de processo preceptivo definem um conjunto prescrito de elementos de processo e um fluxo de trabalho de processo previsível.

**FIGURA 2.3**

**O modelo cascata**



### PONTO-CHAVE

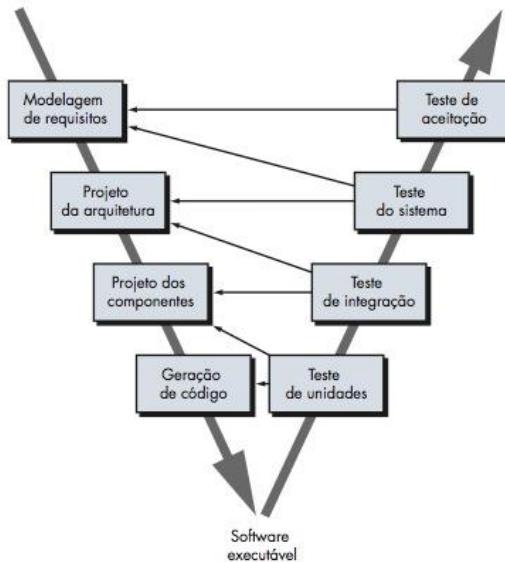
O modelo V ilustra como as ações de verificação e validação estão associadas a ações de engenharia anteriores.

Uma variação na representação do modelo cascata é denominada *modelo V*. Representado na Figura 2.4, o modelo V [Buc99] descreve a relação entre ações de garantia da qualidade e as ações associadas à comunicação, modelagem e atividades de construção iniciais. À medida que a equipe de software desce em direção ao lado esquerdo do V, os requisitos básicos do problema são refinados em representações progressivamente cada vez mais detalhadas e técnicas do problema e de sua solução. Uma vez que o código tenha sido gerado, a equipe se desloca para cima, no lado direito do V, realizando basicamente uma série de testes (ações de garantia da qualidade) que validem cada um dos modelos criados à medida que a equipe se desloca para baixo, no lado esquerdo do V.<sup>7</sup> Na realidade, não existe uma diferença fundamental entre o ciclo de vida clássico e o modelo V. O modelo V fornece uma forma para visualizar como a verificação e as ações de validação são aplicadas ao trabalho de engenharia anterior.

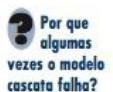
O modelo cascata é o paradigma mais antigo da engenharia de software. Entretanto, ao longo das últimas três décadas, as críticas a este modelo de processo fez com que até mesmo seus mais ardentes defensores questionassem sua eficácia [Han95]. Entre os problemas às vezes encontrados quando se aplica o modelo cascata, temos:

**FIGURA 2.4**

**Modelo V**



<sup>7</sup> Na Parte 3 deste livro é apresentada uma discussão detalhada sobre ações de garantia da qualidade.



**Autor desconhecido**

"Muito frequentemente, o trabalho de software segue o princípio da lei do ciclismo: Não importa para onde se esteja indo, é sempre ladeira acima e contra o vento."

### PONTO-CHAVE

O modelo incremental libera uma série de versões, denominadas incrementais, que oferecem, progressivamente, maior funcionalidade para o cliente à medida que cada incremento é entregue.

1. Projetos reais raramente seguem o fluxo sequencial que o modelo propõe. Embora o modelo linear possa conter iterações, ele o faz indiretamente. Como consequência, mudanças podem provocar confusão à medida que a equipe de projeto prossegue.
2. Frequentemente, é difícil para o cliente estabelecer explicitamente todas as necessidades. O modelo cascata requer isso e tem dificuldade para adequar a incerteza natural que existe no início de muitos projetos.
3. O cliente deve ter paciência. Uma versão operacional do(s) programa(s) não estará disponível antes de estarmos próximos do final do projeto. Um erro grave, se não detectado até o programa operacional ser revisto, pode ser desastroso.

Em uma interessante análise de projetos reais, Bradac [Bra94] descobriu que a natureza linear do ciclo de vida clássico conduz a "estados de bloqueio", nos quais alguns membros da equipe do projeto têm de aguardar outros completarem tarefas dependentes. De fato, o tempo gasto na espera pode exceder o tempo gasto em trabalho produtivo! Os estados de bloqueio tendem a prevalecer no início e no final de um processo sequencial linear.

Hoje em dia, o trabalho de software tem um ritmo acelerado e está sujeito a uma cadeia de mudanças intermináveis (em características, funções e conteúdo de informações). O modelo cascata é frequentemente inapropriado para tal trabalho. Entretanto, ele pode servir como um modelo de processo útil em situações nas quais os requisitos são fixos e o trabalho deve ser realizado até sua finalização de forma linear.

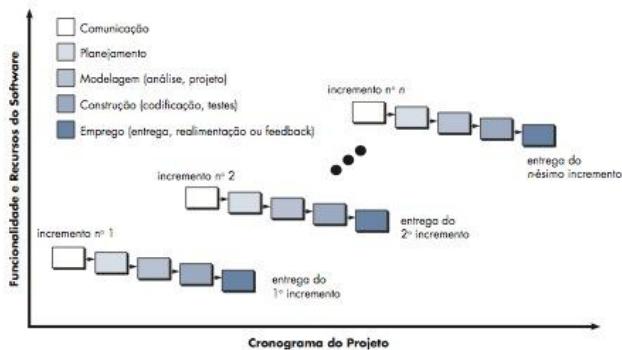
### 2.3.2 Modelos de processo incremental

Em várias situações, os requisitos iniciais do software são razoavelmente bem definidos, entretanto, devido ao escopo geral do trabalho de desenvolvimento, o uso de um processo puramente linear não é utilizado. Pode ser necessário o rápido fornecimento de um determinado conjunto funcional aos usuários, para somente após esse fornecimento, refinar e expandir sua funcionalidade em versões de software posteriores. Em tais casos, pode-se optar por um modelo de processo projetado para desenvolver o software de forma incremental.

O modelo *incremental* combina elementos dos fluxos de processos lineares e paralelos, discutidos na Seção 2.1. Na Figura 2.5, o modelo incremental aplica sequências lineares, de forma escalonada, à medida que o tempo vai avançando. Cada sequência linear gera "incrementais" (entregáveis/aprovados/liberados) do software [McD93] de maneira similar aos incrementais gerados por um fluxo de processos evolucionários (Seção 2.3.3).

**FIGURA 2.5**

#### O modelo incremental





**Se o cliente precisa**  
da entrega até uma  
determinada data  
impossível de ser  
atendida, sugira a  
entrega de um ou mais  
incrementos até a data  
e o restante do software  
(incrementos adicionais)  
posteriormente.

#### PONTO- -CHAVE

Os modelos de  
processo evolucionário  
produzem uma  
versão cada vez mais  
completa do software  
a cada iteração.

"Planeje para jogar  
algo fora. Você irá  
fazê-lo de qualquer  
maneira. Sua  
escolha consistirá  
em decidir se deve  
tentar ou não  
vender o que foi  
descartado aos  
clientes."

**Frederick P.  
Brooks**

Por exemplo, um software de processamento de texto desenvolvido com o emprego do paradigma incremental, poderia liberar funções básicas de gerenciamento de arquivos, edição e produção de documentos no primeiro incremento; recursos mais sofisticados de edição e produção de documentos no segundo; revisão ortográfica e gramatical no terceiro; e, finalmente, recursos avançados de formatação (layout) de página no quarto incremento. Deve-se notar que o fluxo de processo para qualquer incremento pode incorporar o paradigma da prototipação.

Quando se utiliza um modelo incremental, frequentemente, o primeiro incremento é um *produto essencial*. Isto é, as os requisitos básicos são atendidos, porém, muitos recursos complementares (alguns conhecidos, outros não) ainda não são entregues. Esse produto essencial é utilizado pelo cliente (ou passa por uma avaliação detalhada). Como resultado do uso e/ou avaliação, é desenvolvido um planejamento para o incremento seguinte. O planejamento já considera a modificação do produto essencial para melhor se adequar às necessidades do cliente e à entrega de recursos e funcionalidades adicionais. Esse processo é repetido após a liberação de cada incremento, até que seja produzido o produto completo.

O modelo de processo incremental tem seu foco voltado para a entrega de um produto operacional com cada incremento. Os primeiros incrementos são versões seccionadas do produto final, mas eles realmente possuem capacidade para atender ao usuário e também oferecem uma plataforma para avaliação do usuário.<sup>8</sup>

O desenvolvimento incremental é particularmente útil nos casos em que não há pessoal disponível para uma completa implementação na época de vencimento do prazo estabelecido para o projeto. Os primeiros incrementos podem ser implementados com número mais reduzido de pessoal. Se o produto essencial for bem acolhido, então um pessoal adicional (se necessário) poderá ser acrescentado para implementar o incremento seguinte. Além disso, os incrementos podem ser planejados para administrar riscos técnicos. Por exemplo, um sistema importante pode exigir a disponibilidade de novo hardware que ainda está em desenvolvimento e cuja data de entrega é incerta. Poderia ser possível planejar incrementos iniciais de modo a evitar o uso desse hardware, possibilitando, portanto, a liberação de funcionalidade parcial aos usuários finais, sem um atraso excessivo.

#### 2.3.3 Modelos de processo evolucionário

Software, assim como todos sistemas complexos, evolui ao longo do tempo. Conforme o desenvolvimento do projeto avança, as necessidades de negócio e de produto mudam frequentemente, tornando inadequado seguir um planejamento em linha reta de um produto final. Prazos apertados, determinados pelo mercado, tornam impossível completar um produto de software abrangente, porém uma versão limitada tem de ser introduzida para aliviar e/ou atender às pressões comerciais ou da concorrência. Um conjunto do produto essencial ou das necessidades do sistema está bem compreendido, entretanto, detalhes de extensões do produto ou do sistema ainda devem ser definidos. Em situações como essa ou similares, faz-se necessário um modelo de processo que tenha sido projetado especificamente para desenvolver um produto que evolua ao longo do tempo.

Modelos evolucionários são iterativos. Apresentam características que possibilitam desenvolver versões cada vez mais completas do software. Nos parágrafos seguintes, são apresentados dois modelos comuns em processos evolucionários.

**Prototipação.** Frequentemente, o cliente define uma série de objetivos gerais para o software, mas não identifica, detalhadamente, os requisitos para funções e recursos. Em outros casos, o desenvolvedor encontra-se inseguro quanto à eficiência de um algoritmo, quanto à adaptabilidade de um sistema operacional ou quanto à forma em que deva ocorrer a interação homem/máquina. Em situações como essas, e em muitas outras, o *paradigma de prototipação* pode ser a melhor escolha de abordagem.

<sup>8</sup> É importante notar que uma filosofia incremental também é utilizada em todos os modelos de processos "ágiles", discutidos no Capítulo 3.

Embora a prototipação possa ser utilizada como um modelo de processo isolado (stand-alone process) é mais comumente utilizada como uma técnica passível de ser implementada no contexto de qualquer um dos modelos de processo citados neste capítulo. Independentemente da forma como é aplicado, quando os requisitos estão obscuros, o paradigma da prototipação auxilia os interessados a compreender melhor o que está para ser construído.

### AVISO

Quando seu cliente tiver uma necessidade legítima, mas sem a mínima ideia em relação a detalhes, faça um protótipo para uma primeira etapa.

O paradigma da prototipação (Figura 2.6) começa com a comunicação. Faz-se uma reunião com os envolvidos para definir os objetivos gerais do software, identificar quais requisitos já são conhecidos e esquematizar quais áreas necessitam, obrigatoriamente, de uma definição mais ampla. Uma iteração de prototipação é planejada rapidamente e ocorre a modelagem (na forma de um "projeto rápido"). Um projeto rápido se concentra em uma representação daqueles aspectos do software que serão visíveis aos usuários finais (por exemplo, o layout da interface com o usuário ou os formatos de exibição na tela).

O projeto rápido leva à construção de um protótipo, que é empregado e avaliado pelos envolvidos, que fornecerão um retorno (feedback), que servirá para aprimorar os requisitos. A iteração ocorre conforme se ajusta o protótipo às necessidades de vários interessados e, ao mesmo tempo, possibilita a melhor compreensão das necessidades que devem ser atendidas.

Na sua forma ideal, o protótipo atua como um mecanismo para identificar os requisitos do software. Caso seja necessário desenvolver um protótipo operacional, pode-se utilizar partes de programas existentes ou aplicar ferramentas (por exemplo, geradores de relatórios e gerenciadores de janelas) que possibilitem gerar rapidamente tais programas operacionais.

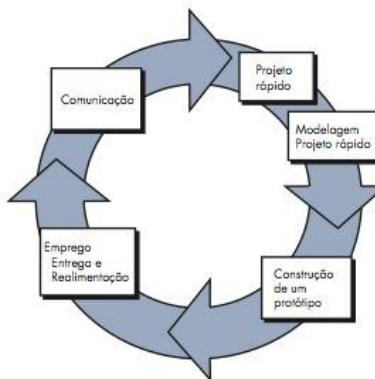
O que fazer com o protótipo quando este já serviu ao propósito descrito anteriormente? Brooks [Bro95] fornece uma resposta:

Na maioria dos projetos, o primeiro sistema dificilmente é utilizável. Pode estar muito lento, muito grande, estranho em sua utilização ou as três coisas juntas. Não há alternativa, a não ser começar de novo, ressentindo-se, porém, mais esperto (aprimorado), e desenvolver uma versão redesenhada na qual esses problemas são resolvidos.

O protótipo pode servir como "o primeiro sistema". Aquele que Brooks recomenda que se jogue fora. Porém, essa pode ser uma visão idealizada. Embora alguns protótipos sejam construídos como "descartáveis", outros são evolucionários, no sentido de que evoluem lentamente até se transformar no sistema real.

Tanto os interessados como os engenheiros de software gostam do paradigma da prototipação. Os usuários podem ter uma ideia prévia do sistema final, ao passo que os desenvolvedores

**FIGURA 2.6**  
O paradigma da prototipação



passam a desenvolver algo imediatamente. Entretanto, a prototipação pode ser problemática pelas seguintes razões:



*Resista à pressão de estender um protótipo grosseiro a um produto final. Quase sempre, como resultado, a qualidade fica comprometida.*

1. Os interessados enxergam o que parece ser uma versão operacional do software, ignorando que o protótipo é mantido de forma não organizada e que, na pressa de fazer com que ele se torne operacional, não se considera a qualidade global do software, nem sua manutenção a longo prazo. Quando informados que o produto deve ser reconstruído para que altos níveis de qualidade possam ser mantidos, os interessados protestam e solicitam que "umas poucas correções" sejam feitas para tornar o protótipo um produto operacional. Frequentemente, a gerência do desenvolvimento de software aceita.
2. O engenheiro de software, com frequência, assume compromissos de implementação para conseguir que o protótipo entre em operação rapidamente. Um sistema operacional ou linguagem de programação inapropriados podem ser utilizados simplesmente porque se encontram à disposição e são conhecidos; um algoritmo ineficiente pode ser implementado simplesmente para demonstrar capacidade. Após um tempo, pode-se acomodar com tais escolhas e esquecer todas as razões pelas quais eram inapropriadas. Uma escolha longe da ideal acaba se tornando parte integrante do sistema.

Embora possam ocorrer problemas, a prototipação pode ser um paradigma efetivo para a engenharia de software. O segredo é definir as regras do jogo logo no início; isso significa que todos os envolvidos devem concordar que o protótipo é construído para servir como um mecanismo para definição de requisitos. Portanto, será descartado (pelo menos em parte) e o software final é arquitetado visando qualidade.

## CASASEGURA



### Seleção de um Modelo de Processo, Parte 1

**Cena:** Sala de reuniões da equipe de engenharia de software na CPI Corporation, uma empresa (fictícia) que fabrica produtos de consumo para uso doméstico e comercial.

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

#### A conversa:

**Lee:** Então, vamos recapitular. Passei algum tempo discutindo sobre a linha de produtos CasaSegura, da forma como a visualizamos no momento. Sem dúvida, temos um monte de trabalho a ser feito para simplesmente definir as coisas, mas eu gostaria que vocês começassem a pensar sobre como irão abordar a parte do software deste projeto.

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

**Ed:** Eu não sei, Doug, nós sempre conseguimos entregar o produto.

**Doug:** É verdade, mas não sem grande sofrimento, e esse projeto parece ser maior e mais complexo do que qualquer outro que fizemos no passado.

**Jamie:** Não parece assim tão difícil, mas eu concordo... nossa abordagem "ad hoc" adotada para projetos anteriores não dará certo neste caso, principalmente se tivermos um cronograma muito apertado.

**Doug (sorrindo):** Quero ser um pouco mais profissional em nossa abordagem. Participei de um curso rápido na semana passada e aprendi bastante sobre engenharia de software... Bom conteúdo. Precisamos de um processo aqui.

**Jamie (franzindo a testa):** Minha função é desenvolver programas, não ficar mexendo em papéis.

**Doug:** Dê uma chance antes de me dizer não. Eis o que quero dizer. [Doug prossegue descrevendo a metodologia de processo descrita neste capítulo e os modelos de processo prescritivos apresentados até agora.]

**Doug:** Mas de qualquer forma, parece-me que um modelo linear não é adequado para nós... Ele assume que temos todos os requisitos antecipadamente e, conhecendo este lugar, isso é pouco provável.

**Vinod:** Isso mesmo, e parece orientado demais à tecnologia da informação... Provavelmente bom para construir um sistema de controle de estoque ou algo parecido, mas certamente não é adequado para o CasaSegura.

**Doug:** Eu concordo.

**Ed:** Essa abordagem de prototipação me parece OK. Bastante parecido com o que fazemos aqui.

**Vinod:** Isso é um problema. Estou preocupado que ela não nos dê estrutura suficiente.

**Doug:** Não é para se preocupar. Temos um monte de outras opções e quero que vocês escolham o que for melhor para a equipe e para o projeto.

**Modelo Espiral.** Originalmente proposto por Barry Boehm [Boe88], o *modelo espiral* é um modelo de processo de software evolucionário que acopla a natureza iterativa da prototipação com os aspectos sistemáticos e controlados do modelo cascata. Fornece potencial para o rápido desenvolvimento de versões cada vez mais completas do software. Boehm [Boe01a] descreve o modelo da seguinte maneira:

O modelo espiral de desenvolvimento é um gerador de *modelos de processos* dirigidos a riscos e é utilizado para guiar a engenharia de sistemas intensivos de software, que ocorre de forma concorrente e tem múltiplos envolvidos. Possui duas características principais que o distinguem. A primeira consiste em uma abordagem *cíclica voltada* para ampliar, incrementalmente, o grau de definição e a implementação de um sistema, enquanto diminui o grau de risco do mesmo. A segunda característica consiste em uma série de pontos âncora de controle para assegurar o comprometimento de interessados quanto à busca de soluções de sistema que sejam mutuamente satisfatórias e praticáveis.

Usando-se o modelo espiral, o software será desenvolvido em uma série de versões evolucionárias. Nas primeiras iterações, a versão pode consistir em um modelo ou em um protótipo. Já nas iterações posteriores, são produzidas versões cada vez mais completas do sistema que passa pelo processo de engenharia.

Um modelo espiral é dividido em um conjunto de atividades metodológicas definidas pela equipe de engenharia de software. Para fins ilustrativos, utilizam-se as atividades metodológicas genéricas discutidas anteriormente.<sup>9</sup> Cada uma dessas atividades representa um segmento do caminho espiral ilustrado na Figura 2.7. Assim, que esse processo evolucionário comece, a equipe de software realiza atividades indicadas por um circuito em torno da espiral no sentido horário, começando pelo seu centro. Os riscos (Capítulo 28) são considerados à medida que cada revolução é realizada. Pontos âncora de controle — uma combinação de produtos de trabalho e condições que são satisfeitas ao longo do trajeto da espiral — são indicados para cada passagem evolucionária.

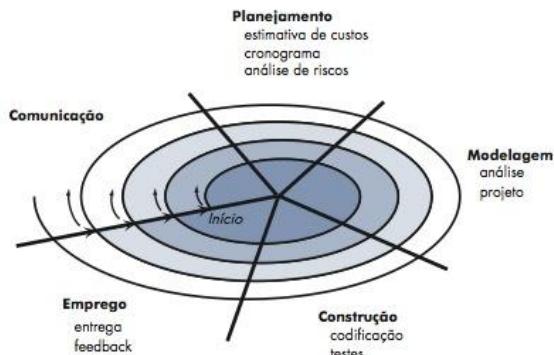
O primeiro circuito em volta da espiral pode resultar no desenvolvimento de uma especificação de produto; passagens subsequentes em torno da espiral podem ser usadas para desenvolver um protótipo e, então, progressivamente, versões cada vez mais sofisticadas do software. Cada passagem pela região de planejamento resulta em ajustes no planejamento do projeto.

### PONTO-CHAVE

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

**FIGURA 2.7**

**Modelo espiral típico**



<sup>9</sup> O modelo espiral discutido nesta seção é uma variação do modelo proposto por Boehm. Para mais informações sobre o modelo espiral original, consulte [Boe88]. Material mais recente sobre o modelo espiral de Boehm pode ser encontrado em [Boe98].

**WebRef**

Informações úteis sobre o modelo espiral podem ser obtidas em: [www.sei.cmu.edu/publications/documents/00.reports/00sr008.html](http://www.sei.cmu.edu/publications/documents/00.reports/00sr008.html).

**AVISO**

*Se a gerência quiser um desenvolvimento com orçamento fixo (geralmente uma péssima ideia), a espiral pode ser um problema. À medida que cada círculo for realizado, o custo do projeto será rapidamente revisado.*

*"Estou tão perto, mas apenas o amanhã guia o meu caminho."*

**Dave Matthews Band**

Custo e cronograma são ajustados de acordo com o feedback (a realimentação) obtido do cliente após entrega. Além disso, o gerente de projeto faz um ajuste no número de iterações planejadas para completar o software.

Diferente de outros modelos de processo, que terminam quando o software é entregue, o modelo espiral pode ser adaptado para ser aplicado ao longo da vida do software. Portanto, o primeiro circuito em torno da espiral pode representar um "projeto de desenvolvimento de conceitos" que começa no núcleo da espiral e continua por várias iterações<sup>10</sup>, até que o desenvolvimento de conceitos esteja completo. Se o conceito for desenvolvido para ser um produto final, o processo prossegue pela espiral pelas "bordas" e um "novo projeto de desenvolvimento de produto" se inicia. O novo produto evoluirá, passando por uma série de iterações, em torno da espiral. Posteriormente, uma volta em torno da espiral pode ser usada para representar um "projeto de aperfeiçoamento do produto". Em sua essência, a espiral, caracterizada dessa maneira, permanece em operação até que o software seja retirado. Há casos em que o processo fica inativo, porém, toda vez que uma mudança é iniciada, começa no ponto de partida apropriado (por exemplo, aperfeiçoamento do produto).

O modelo espiral é uma abordagem realista para o desenvolvimento de sistemas e de software em larga escala. Pelo fato de o software evoluir à medida que o processo avança, o desenvolvedor e o cliente compreendem e reagem melhor aos riscos em cada nível evolucionário. Esse modelo usa a prototipação como mecanismo de redução de riscos e, mais importante, torna possível a aplicação da prototipação em qualquer estágio do processo evolutivo do produto. Mantém a abordagem em etapas, de forma sistemática, sugerida pelo ciclo de vida clássico, mas a incorpora em uma metodologia iterativa que reflete mais realisticamente o mundo real. O modelo espiral requer consideração direta dos riscos técnicos em todos os estágios do projeto e, se aplicado apropriadamente, reduz os riscos antes de se tornarem problemáticos.

Mas como outros paradigmas, esse modelo não é uma panaceia. Pode ser difícil convencer os clientes (particularmente em situações contratuais) de que a abordagem evolucionária é controlável. Ela exige considerável especialização na avaliação de riscos e depende dessa especialização para seu sucesso. Se um risco muito importante não for descoberto e administrado, indubitavelmente ocorrerão problemas.

**CASASEGURA****Seleção de um modelo de processo, Parte 2**

**Cena:** Sala de reuniões do grupo de engenharia de software na CPI Corporation, empresa (fictícia) que fabrica produtos de consumo de uso doméstico e comercial.

**Participantes:** Lee Warren, gerente de engenharia; Doug Miller, gerente de engenharia de software; Vinod e Jamie, membros da equipe de engenharia de software.

**Conversa:** [Doug descreve as opções do processo evolutório.]

**Jamie:** Agora estou vendo algo de que gosto. Faz sentido uma abordagem incremental e eu realmente gosto do fluxo dessa coisa de modelo espiral. Isso tem a ver com a realidade.

**Vinod:** Concordo. Entregamos um incremento, aprendemos com o feedback do cliente, replanejamos e, então, entregamos

outro incremento. Também se encaixa na natureza do produto. Podemos colocar alguma coisa no mercado rapidamente, ter algo no mercado e, depois, acrescentar funcionalidade a cada versão, digo, incremento.

**Lee:** Espere um pouco, você disse que reformulamos o plano a cada volta na espiral, Doug? Isso não é tão legal; precisamos de um plano, um cronograma e temos de nos ater a ele.

**Doug:** Essa linha de pensamento é antiga, Lee. Como o pessoal disse, temos de mantê-lo real. Acho que é melhor ir ajustando o planejamento na medida em que formos aprendendo mais e as mudanças sejam solicitadas. É muito mais realista. Para que serve um plano se não refletir a realidade?

**Lee** [franzindo a testa]: Suponho que esteja certo, porém... A alta gerência não vai gostar disso... Querem um plano fixo.

**Doug** [sorrindo]: Então, você terá que reeducá-los, meu amigo.

<sup>10</sup> As setas que apontam para dentro, ao longo do eixo, separando a região de **emprego** da região de **comunicação**, indicam potencial para iteração local ao longo do mesmo trajeto da espiral.

### 2.3.4 Modelos concorrentes

O modelo de desenvolvimento concorrente, algumas vezes denominado *engenharia concorrente*, possibilita à equipe de software representar elementos concorrentes e iterativos de qualquer um dos modelos de processos descritos neste capítulo. Por exemplo, a atividade de modelagem definida para o modelo espiral é realizada invocando uma ou mais das seguintes ações de engenharia de software: prototipagem, análise e projeto.<sup>11</sup>



O modelo concorrente, com frequência, é mais adequado para projetos de engenharia de produto nos quais diferentes equipes de engenharia estão envolvidas.

A Figura 2.8 mostra um esquema de uma atividade da engenharia de software, dentro da atividade de modelagem, usando uma abordagem de modelagem concorrente. A atividade — **modelagem** — poderá estar em qualquer um dos estados<sup>12</sup> observados em qualquer instante determinado. Similarmente, outras atividades, ações ou tarefas (por exemplo, **comunicação** ou **construção**) podem ser representadas de maneira análoga. Todas as atividades de engenharia de software existem concorrentemente, porém estão em diferentes estados.

Por exemplo, no início de um projeto, a atividade de comunicação (não mostrada na figura) completou sua primeira iteração e se encontra no estado **aguardando modificações**. A atividade de modelagem (que se encontrava no estado **inativo** enquanto a comunicação inicial era completada, agora faz uma transição para o estado **em desenvolvimento**). Se, entretanto, o cliente indicar que mudanças nos requisitos devem ser feitas, a atividade de modelagem passa do estado **em desenvolvimento** para o estado **aguardando modificações**.

A modelagem concorrente define uma série de eventos que irão disparar transições de estado para estado para cada uma das atividades, ações ou tarefas da engenharia de software.

**FIGURA 2.8**  
Um elemento do  
modelo de processo  
concorrente



11 Deve-se notar que a análise e o projeto são tarefas complexas que requerem discussão substancial. A Parte 2 deste livro considera esses tópicos em detalhe.

12 Um estado é algum modo externamente observável do comportamento.

Por exemplo, durante estágios de projeto iniciais (uma ação de engenharia de software importante que ocorre durante a atividade de modelagem), uma inconsistência no modelo de requisitos não é descoberta. Isso gera o evento *correção do modelo de análise*, que irá disparar a ação de análise de requisitos, passando do estado **concluído** para o estado **aguardando modificações**.

A modelagem concorrente se aplica a todos os tipos de desenvolvimento de software e fornece uma imagem precisa do estado atual de um projeto. Em vez de limitar as atividades, ações e tarefas da engenharia de software a uma sequência de eventos, ela define uma rede de processos. Cada atividade, ação ou tarefa na rede existe simultaneamente com outras atividades, ações ou tarefas. Eventos gerados em um ponto da rede de processos disparam transições entre os estados.

### 2.3.5 Um comentário final sobre processos evolucionários

Como já foi mencionado anteriormente, os softwares modernos são caracterizados por contínuas modificações, prazos muito apertados e por uma ênfase na satisfação do cliente-usuário. Em muitos casos, o tempo de colocação de um produto no mercado é o requisito mais importante a ser gerenciado. Se o momento oportuno de entrada no mercado for perdido, o projeto de software pode ficar sem sentido.<sup>13</sup>

Os modelos de processo evolucionário foram concebidos para tratar dessas questões e, mesmo assim, como uma classe genérica de modelos de processo, apresentam seus pontos fracos. Esses pontos fracos foram resumidos por Nogueira e seus colegas [Nog00]:

Apesar dos inquestionáveis benefícios proporcionados pelos processos de software evolucionários, temos algumas preocupações. A primeira delas é que a prototipação [e outros processos evolucionários mais sofisticados] traz um problema para o planejamento do projeto devido ao número incerto de ciclos necessários para construir o produto. A maior parte das técnicas de gerenciamento e de estimativas do projeto baseia-se em layouts lineares das atividades, o que faz com que não se adequem completamente.

Em segundo lugar, os processos de software evolucionários não estabelecem a velocidade máxima da evolução. Se as evoluções ocorrerem numa velocidade excessivamente rápida, sem um período de acomodação, é certo que o processo cairá no caos. Por outro lado, se a velocidade for muito lenta, então a produtividade poderia ser afetada...

Em terceiro lugar, os processos de software devem manter seu foco mais na flexibilidade e na extensibilidade do que na alta qualidade. Essa afirmação soa assustadora. Entretanto, deve-se priorizar mais a velocidade de desenvolvimento do que a do desenvolvimento com zero defeito. Prolongar o desenvolvimento em busca da alta qualidade pode resultar em entrega tardia do produto, quando o nicho de oportunidade já desapareceu. Essa mudança de paradigma é imposta pela concorrência em situações em que se está à beira do caos.

Realmente, um processo de software priorizando flexibilidade, extensibilidade e velocidade de desenvolvimento, acima da alta qualidade, soa assustador. Ainda assim, essa ideia foi proposta por uma série de renomados especialistas em engenharia de software (como, por exemplo, [You95], [Bac97]).

O objetivo dos modelos evolucionários é desenvolver software de alta qualidade<sup>14</sup> de modo iterativo ou incremental. Entretanto, é possível usar um processo evolucionário para enfatizar a flexibilidade, a extensibilidade e a velocidade do desenvolvimento. O desafio para as equipes de software e seus gerentes será estabelecer um equilíbrio apropriado entre esses parâmetros críticos de projeto e produto e a satisfação dos clientes (o árbitro final da qualidade de um software).

<sup>13</sup> É importante notar, entretanto, que ser o primeiro a chegar ao mercado não é sinônimo de sucesso. De fato, muitos produtos de software bem-sucedidos foram o segundo ou até mesmo o terceiro a chegar ao mercado (aprendendo com os erros dos outros que o antecederam).

<sup>14</sup> Nesse contexto, a qualidade de software é definida de forma bastante abrangente para englobar não apenas a satisfação dos clientes, como também uma série de critérios técnicos, discutidos nos Capítulos 14 e 16.

"Em todo processo há um cliente, pois, sem cliente, um processo deixa de ter sentido."

V. Daniel Hunt

## 2.4 MODELOS DE PROCESSO ESPECIALIZADO

Os modelos de processo especializado levam em conta muitas das características de um ou mais dos modelos tradicionais apresentados nas seções anteriores. Tais modelos tendem a ser aplicados quando se opta por uma abordagem de engenharia de software especializada ou definida de forma restrita.<sup>15</sup>

### 2.4.1 Desenvolvimento baseado em componentes



Componentes de software comercial de prateleira ou COTS (sigla para Commercial Off-The-Shelf), desenvolvidos por vendedores que os oferecem como produtos, disponibilizam a funcionalidade almejada juntamente com as bem definidas interfaces, sendo que essas interfaces permitem que o componente seja integrado ao software a ser desenvolvido. O *modelo de desenvolvimento baseado em componentes* incorpora muitas das características do modelo espiral. É evolucionário em sua natureza [Nie92], demandando uma abordagem iterativa para a criação de software. O modelo de desenvolvimento baseado em componentes desenvolve aplicações a partir de componentes de software pré-empacotados.

As atividades de modelagem e construção começam com a identificação de possíveis candidatos a componentes. Esses componentes podem ser projetados como módulos de software convencionais, como classes orientadas a objeto ou pacotes<sup>16</sup> de classes. Independentemente da tecnologia usada para criar os componentes, o modelo de desenvolvimento baseado em componentes incorpora as seguintes etapas (implementadas usando-se uma abordagem evolucionária):

1. Produtos baseados em componentes disponíveis são pesquisados e avaliados para o campo de aplicação em questão.
2. Itens de integração de componentes são considerados.
3. Uma arquitetura de software é projetada para acomodar os componentes.
4. Os componentes são integrados na arquitetura.
5. Testes completos são realizados para assegurar funcionalidade adequada.

O modelo de desenvolvimento baseado em componentes conduz ao reúso do software e a reusabilidade proporciona uma série de benefícios mensuráveis aos engenheiros de software. A equipe de engenharia de software pode conseguir uma redução no tempo do ciclo de desenvolvimento, bem como uma redução no custo do projeto, caso a reutilização de componentes se torne parte de sua cultura. O desenvolvimento baseado em componentes é discutido de forma mais detalhada no Capítulo 10.

### 2.4.2 O modelo de métodos formais

O *modelo de métodos formais* engloba um conjunto de atividades que conduzem à especificação matemática formal do software. Os métodos formais possibilitam especificar, desenvolver e verificar um sistema baseado em computador através da aplicação de uma notação matemática rigorosa. Uma variação dessa abordagem, chamada *engenharia de software "cleanroom" (sala limpa/leve/limpida)* [Mil87, Dye92], é aplicada atualmente por algumas organizações de desenvolvimento de software.

Quando são utilizados métodos formais (Capítulo 21) durante o desenvolvimento, estes oferecem um mecanismo que elimina muitos dos problemas difíceis de ser superados com o uso

<sup>15</sup> Em alguns casos, esses modelos de processo especializado podem ser mais bem definidos como um conjunto de técnicas, ou uma "metodologia", para alcançar uma meta de desenvolvimento de software específica. Entretanto, elas realmente implicam um processo.

<sup>16</sup> Conceitos de orientação a objetos são discutidos no Apêndice 2 e são usados ao longo da Parte 2 deste livro. Nesse contexto, uma classe engloba um conjunto de dados e os procedimentos que processam os mesmos. Pacote de classes é um conjunto de classes relativas que operam juntas para alcançar algum resultado final.

de outros paradigmas de engenharia de software. Ambiguidade, incompletude e inconsistência podem ser descobertas e corrigidas mais facilmente — não por meio de uma revisão local, mas devido à aplicação de análise matemática. Quando são utilizados métodos formais durante o projeto, servem como base para verificar a programação e, portanto, possibilitam que se descubra e se corrijam erros que, de outra forma, poderiam passar despercebidos.

Embora não seja uma abordagem predominante, o modelo de métodos formais oferece a promessa de software sem defeitos. No entanto, foram mencionados motivos para preocupação a respeito de sua aplicabilidade em um ambiente de negócios:

 Se métodos formais são capazes de demonstrar correção de software, porque não são amplamente utilizados?

- Atualmente, o desenvolvimento de modelos formais consome muito tempo e dinheiro.
- Pelo fato de poucos desenvolvedores de software possuírem formação e experiência necessárias (background) para aplicação dos métodos formais, é necessário treinamento extensivo.
- É difícil usar os modelos como um meio de comunicação com clientes tecnicamente des-preparados (não sofisticados tecnicamente).

Apesar de tais preocupações, a abordagem de métodos formais tem conquistado adeptos entre os desenvolvedores de software que precisam desenvolver software com fator crítico de segurança (como, por exemplo, os desenvolvedores de sistemas aviônicos para aeronaves e equipamentos médicos), bem como entre desenvolvedores que sofreriam pesadas sanções econômicas se ocorressem erros no software.

#### 2.4.3 Desenvolvimento de software orientado a aspectos

##### WebRef

Uma ampla gama de recursos e informações sobre AOP pode ser encontrada em: [aosd.net](http://aosd.net).

##### PONTO-CHAVE

A AOSD define “aspectos” representando restrições do cliente que cruzam várias funções, recursos e informações do sistema.

Independentemente do processo de software escolhido, os desenvolvedores de software complexos, invariavelmente, implementam um conjunto de recursos, funções e conteúdo localizados. Essas características de software localizadas são modeladas como componentes (por exemplo, classes orientadas a objetos) e, em seguida, construídas dentro do contexto da arquitetura do sistema. À medida que os modernos sistemas baseados em computadores se tornam mais sofisticados (e complexos), certas restrições — propriedades exigidas pelo cliente ou áreas de interesse técnico — se estendem por toda a arquitetura. Algumas restrições são propriedades de alto nível de um sistema (por exemplo, segurança, tolerância a falhas). Outras afetam funções (por exemplo, a aplicação de regras de negócio), sendo que outras são sistêmicas (por exemplo, sincronização de tarefas ou gerenciamento de memória).

Quando restrições cruzam múltiplas funções, recursos e informações do sistema, elas são, frequentemente, denominadas *restrições cruzadas*. Os requisitos de aspectos definem as restrições cruzadas que têm um impacto por toda a arquitetura de software. O desenvolvimento de software orientado a aspectos AOSD, *Aspect-Oriented Software Development*, com frequência conhecido como programação orientada a aspectos (AOP, *Aspect-Oriented Programming*), é um paradigma de engenharia de software relativamente novo que oferece uma abordagem metodológica e de processos para definir, especificar, projetar e construir *aspectos* — “mecanismos além das sub-rotinas e herança para localizar a expressão de uma restrição cruzada” [Elr01].

Grundy [Gru02] discute ainda mais os aspectos no contexto do que ele denomina engenharia de componentes orientada a aspectos (AOCE, *aspect-oriented component engineering*):

A AOCE usa um conceito de fatias horizontais através de componentes de software decompostos verticalmente, chamados “aspectos”, para caracterizar propriedades funcionais ou não funcionais cruzadas dos componentes. Aspectos sistêmicos, comuns, incluem interfaces com o usuário, trabalho colaborativo, distribuição, persistência, gerenciamento de memória, processamento de transações, segurança, integridade e assim por diante. Os componentes podem fornecer ou requerer um ou mais “detalhes de aspecto” relativo a um determinado aspecto, tais como um mecanismo de visualização, exequibilidade extensiva e gênero de interface (aspectos da interface com o usuário); geração de eventos, transporte e recebimento (aspectos de distribuição); armazenamento/recuperação e indexação de dados (aspectos de persistência); autenticação, codificação e direitos de acesso (aspectos de segurança); atomicidade

de transações, controle de concorrência e estratégia de entrada no sistema (aspectos transacionais) e assim por diante. Cada detalhe de aspecto possui uma série de propriedades relativas a características funcionais e não funcionais do detalhe do aspecto.

Um processo distinto orientado a aspectos ainda não atingiu sua maturação. Entretanto, é provável que um processo desses irá adotar características tanto dos modelos de processo evolucionário quanto de concorrente. O modelo evolucionário é apropriado quando os aspectos são identificados e então construídos. A natureza paralela do desenvolvimento concorrente é essencial, porque os aspectos são criados independentemente de componentes de software localizado e, apesar disso, os aspectos têm um impacto direto sobre esses componentes. Portanto, é essencial instanciar comunicação assíncrona entre as atividades de processos de software aplicadas na engenharia e construção de aspectos e componentes.

Uma discussão detalhada sobre desenvolvimento de software orientado a aspectos é mais bem colocada em livros dedicados ao assunto. Se tiver mais interesse, consulte [Saf08], [Cla05], [Jac04] e [Gra03].

#### FERRAMENTAS DO SOFTWARE



##### Gerenciamento de processos

**Objetivo:** ajudar na definição, execução e gerenciamento de modelos de processos prescritivos.

**Mecânicas:** as ferramentas de gerenciamento de processo possibilitam que uma organização ou equipe de software definam um modelo de processo de software completo (atividades metodológicas, ações, tarefas, pontos de verificação para garantia da qualidade, pontos de controle (marcos) e artefatos de software). Além disso, tais ferramentas acabam fornecendo um mapeamento (guias), conforme os engenheiros de software realizam o trabalho técnico, e também propiciam um modelo para os gerentes, os quais têm o dever de acompanhar e controlar o processo de software.

**Ferramentas Representativas:**<sup>17</sup> o GDPA, um conjunto de ferramentas para definição de processos de pesquisa, desen-

volido na Universidade de Bremen, na Alemanha ([www.informatik.uni-bremen.de/uniform/gdpa/home.htm](http://www.informatik.uni-bremen.de/uniform/gdpa/home.htm)), fornece uma grande quantidade de funções de gerenciamento e modelagem de processos.

O SpeeDev, desenvolvido pela SpeeDev Corporation ([www.speedev.com](http://www.speedev.com)) engloba um conjunto de ferramentas para definição de processo, gerenciamento de requisitos, resolução de itens, planejamento e acompanhamento de projetos.

O ProVision BPMx, desenvolvido pela Proforma ([www.proformacorp.com](http://www.proformacorp.com)), é representante de muitas ferramentas que auxiliam na definição de processo e automação de fluxo de trabalho.

Uma lista valiosa de diversas ferramentas diferentes associadas ao processo de software, pode ser encontrada no endereço [www.processwave.net/Links/tool\\_links.htm](http://www.processwave.net/Links/tool_links.htm).

## 2.5 O PROCESSO UNIFICADO

No livro que deu origem ao *Processo Unificado*, Ivar Jacobson, Grady Booch e James Rumbaugh [Jac99] discutem a necessidade de um processo de software "dirigido a casos de uso, centrado na arquitetura, iterativo e incremental" ao afirmarem:

Hoje em dia, a tendência do software é no sentido de sistemas maiores e mais complexos. Isso se deve, em parte, ao fato de que os computadores tornam-se mais potentes a cada ano, levando os usuários a ter uma expectativa maior em relação a eles. Essa tendência também foi influenciada pelo uso crescente da Internet para troca de todos os tipos de informação... Nossa apetite por software cada vez mais sofisticado aumenta à medida que tomamos conhecimento de uma versão do produto para a seguinte, como o produto pode ser aperfeiçoado. Queremos software que seja mais e mais adaptado a nossas necessidades, mas isso, por sua vez, simplesmente torna o software mais complexo. Em suma, queremos cada vez mais.

De certa forma, o Processo Unificado é uma tentativa de aproveitar os melhores recursos e características dos modelos tradicionais de processo de software, mas caracterizando-os de modo a implementar muitos dos melhores princípios do desenvolvimento ágil de software (Capítulo 3). O Processo Unificado reconhece a importância da comunicação com o cliente e de métodos racionalizados (sequencializados) para descrever a visão do cliente sobre um sistema (os

casos de uso<sup>17</sup>). Ele enfatiza o importante papel da arquitetura de software e “ajuda o arquiteto a manter o foco nas metas corretas, tais como compreensibilidade, confiança em mudanças futuras e reutilização” [Jac99]. Ele sugere um fluxo de processo iterativo e incremental, proporcionando a sensação evolucionária que é essencial no desenvolvimento de software moderno.

### 2.5.1 Breve histórico

Durante o início dos anos 1990, James Rumbaugh [Rum91], Grady Booch [Boo94] e Ivar Jacobson [Jac92] começaram a trabalhar em um “método unificado” que combinaria as melhores características de cada um de seus métodos individuais de análise e projeto orientados a objetos e adotaram características adicionais propostas por outros especialistas (por exemplo, [Wir90]) em modelagem orientada a objetos. O resultado foi a UML — uma *linguagem de modelagem unificada* que contém uma notação robusta para a modelagem e o desenvolvimento de sistemas orientados a objetos. Por volta de 1997, a UML tornou-se um padrão de fato da indústria em termos de desenvolvimento de software orientado a objetos.

A UML é usada ao longo da Parte 2 deste livro para representar tanto modelos de projeto quanto de requisitos. O Apêndice 1 apresenta um tutorial introdutório para aqueles que não estão familiarizados com as regras básicas de notações e de modelagem da UML. Uma apresentação completa da UML fica reservada a livros-texto dedicados ao assunto. Livros recomendados estão relacionados no Apêndice 1.

A UML forneceu a tecnologia necessária para dar suporte à prática de engenharia de software orientada a objetos, mas não ofereceu a metodologia de processo para orientar as equipes de projeto na aplicação da tecnologia. Ao longo de poucos anos que se seguiram, Jacobson, Rumbaugh e Booch desenvolveram o *Processo Unificado*, uma metodologia para engenharia de software orientada a objetos usando a UML. Hoje em dia, o Processo Unificado (PU ou UP, Unified Process) e a UML são amplamente utilizados em projetos orientados a objetos de todos os tipos. O modelo incremental e iterativo proposto pelo PU pode e deve ser adaptado para atender necessidades de projeto específicas.

### 2.5.2 Fases do processo unificado<sup>18</sup>

No início deste capítulo, foram apresentadas cinco atividades metodológicas genéricas e argumentos afirmando que elas poderiam ser usadas para descrever qualquer modelo de processo de software. O Processo Unificado não é nenhuma exceção. A Figura 2.9 descreve as “fases” do PU e as relaciona com as atividades genéricas que foram discutidas no Capítulo 1 e anteriormente neste capítulo.

#### PONTO-CHAVE

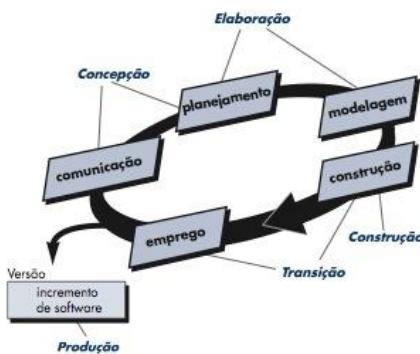
Em seu intento, as fases do Processo Unificado são similares às atividades metodológicas genéricas definidas neste livro.

A fase de concepção (*Inception*) do PU envolve tanto a atividade de comunicação com o cliente como a de planejamento. Colaborando com os interessados, identificam-se as necessidades de negócio para o software; propõe-se uma arquitetura rudimentar para o sistema e se desenvolve um planejamento para a natureza iterativa e incremental do projeto decorrente. Requisitos de negócio fundamentais são descritos por meio de um conjunto de casos práticos preliminares (Capítulo 5), descrevendo quais recursos e funções cada categoria principal de usuário deseja. Até esse ponto, a arquitetura nada mais é do que um esquema provisório dos principais subsistemas e da função e dos recursos que os compõem. Posteriormente, a arquitetura será refinada e expandida para um conjunto de modelos que representarão visões diferentes do sistema. O planejamento identifica recursos, avalia os principais riscos, define um cronograma e estabelece uma base para as fases que serão aplicadas à medida que o incremento de software é desenvolvido.

A fase de elaboração envolve atividades de comunicação e modelagem do modelo de processo genérico (Figura 2.9). A elaboração refina e expande os casos práticos preliminares, desenvol-

17 Um *caso de uso* (Capítulo 5) é uma narrativa textual ou modelo que descreve uma função ou recurso de um sistema do ponto de vista do usuário. Um caso de uso é escrito pelo usuário e serve como base para criação de um modelo de requisitos mais amplo.

18 O Processo Unificado é, algumas vezes, chamado de *Processo Unificado Racional RUP Rational Unified Process* em homenagem a Rational Corporation (posteriormente adquirida pela IBM), um dos primeiros contribuidores para o desenvolvimento e refinamento do PU e um desenvolvedor de ambientes completos (ferramentas e tecnologia) que dão suporte ao processo.

**FIGURA 2.9****O Processo Unificado**

vidos como parte da fase de concepção, e amplia a representação da arquitetura, incluindo cinco visões diferentes do software: modelo de caso prático, modelo de requisitos, modelo de projeto, modelo de implementação e modelo de emprego. Em alguns casos, a elaboração gera uma “base de arquitetura executável” [Arl02], consistindo num sistema executável “de degustação”.<sup>19</sup> Essa base demonstra a viabilidade da arquitetura, mas não oferece todos os recursos e funções necessárias para usar o sistema. Além disso, no auge da fase de elaboração, o plano é revisado cuidadosamente para assegurar que escopo, riscos e datas de entrega permaneçam razoáveis. Normalmente, as modificações no planejamento são feitas nesta oportunidade.

**WebRef**

Uma interessante abordagem sobre o PU, dentro do contexto de desenvolvimento ágil, poderá ser encontrada em [www.ambyssoft.com/unifiedprocess/agileUP.html](http://www.ambyssoft.com/unifiedprocess/agileUP.html).

A fase de construção do PU é idêntica à atividade de construção definida para o processo de software genérico. Tendo como entrada (input) o modelo de arquitetura, a fase de construção desenvolve ou adquire componentes de software; esses componentes farão com que cada caso prático (de uso) se torne operacional para os usuários finais. Para tanto, os modelos de requisitos e de projeto, iniciados durante a fase de elaboração, são completados para refletir a versão final do incremento de software. Então, implementa-se, no código-fonte, todos os recursos e funções necessárias e exigidas para o incremento de software (isto é, para a versão). À medida que os componentes estão sendo implementados, desenvolve-se e executam-se testes de unidades<sup>20</sup> para cada um deles. Além disso, realizam-se atividades de integração (montagem de componentes e testes de integração). Os casos práticos são usados para obter um pacote de testes de aceitação, executados antes do início da fase seguinte do PU.

A fase de transição do PU abrange os últimos estágios da atividade de construção genérica e a primeira parte da atividade de emprego genérico: entrega e realimentação (feedback). Entrega-se o software aos usuários finais para testes beta e o feedback dos usuários relata defeitos e mudanças necessárias. Além disso, a equipe de software elabora material com as informações de apoio (por exemplo, manuais para o usuário, guias para resolução de problemas, procedimentos de instalação) que são necessárias para lançamento da versão. Na conclusão da fase de transição, o incremento torna-se uma versão do software utilizável.

A fase de produção do PU coincide com a atividade de emprego do processo genérico. Durante essa fase, monitora-se o uso contínuo do software, disponibiliza-se suporte para o ambiente (infraestrutura) operacional, realiza-se e avalia-se relatórios de defeitos e solicitações de mudanças.

<sup>19</sup> É importante observar que a base da arquitetura não é um protótipo, já que ela não é descartada. Ao contrário, a base ganha corpo durante a fase seguinte do PU.

<sup>20</sup> Uma discussão extensiva de testes de software (inclusive testes de unidades) é apresentada nos Capítulos 17 a 20.

É provável que, ao mesmo tempo em que as fases de construção, transição e produção estejam sendo conduzidas, já se tenha iniciado o incremento de software seguinte. Isso significa que as cinco fases do PU não ocorrem em sequência, mas sim de forma concorrente e escalonada.

Um fluxo de trabalho de engenharia de software é distribuído ao longo de todas as fases do PU. Nesse contexto, um *fluxo de trabalho* é análogo a um conjunto de tarefas (descrito anteriormente neste capítulo), isto é, identifica as tarefas para realizar uma importante ação de engenharia de software e os artefatos produzidos como consequência da finalização de tarefas com êxito. Deve-se notar que nem toda tarefa identificada para um fluxo de trabalho do PU é conduzida em todos os projetos de software. A equipe adapta o processo (ações, tarefas, subtarefas e artefatos de software) para ficar de acordo com suas necessidades.

## 2.6 MODELOS DE PROCESSO PESSOAL E DE EQUIPE

"Pessoas bem-sucedidas desenvolveram, simplesmente, o hábito de realizar coisas que as fracassadas não irão fazer."

Dexter Yager

O melhor processo de software é aquele próximo às pessoas que realizarão o trabalho. Se um modelo de processo de software for desenvolvido em nível corporativo ou organizacional, ele apenas será efetivo se for aberto a significativas adaptações a fim de atender às necessidades da equipe de projeto (aquela que está efetivamente realizando o trabalho de engenharia de software). Num cenário ideal, você desenvolveria um processo que melhor se adequasse às suas necessidades e, simultaneamente, atendesse às necessidades mais amplas da equipe e da organização. De forma alternativa, a própria equipe pode criar seu próprio processo e, ao mesmo tempo, atender às necessidades mais específicas dos indivíduos e às necessidades mais amplas da organização. Watts Humphrey ([Hum97] e [Hum00]) afirmam que é possível criar um "processo de software pessoal" e/ou um "processo de software da equipe". Ambos requerem trabalho árduo, treinamento e coordenação, mas são alcançáveis.<sup>21</sup>

### 2.6.1 Processo de Software Pessoal (PSP)

Todo desenvolvedor utiliza algum processo para construir software. Esse processo pode ser nebuloso ou específico; pode mudar diariamente; não ser eficiente, efetivo ou bem-sucedido; porém, um "processo" realmente existe. Watts Humphrey [Hum97] sugere que a fim de modificar um processo pessoal não efetivo, um indivíduo deve passar por quatro fases, cada uma exigindo treinamento e orquestração cuidadosa. O *Processo de Software Pessoal* (*sigla PSP, Personal Software Process*) enfatiza a medição pessoal, tanto do artefato de software gerado quanto da qualidade resultante dele. Além disso, responsabiliza o profissional pelo planejamento de projetos (por exemplo, estimativa de custos e cronograma) e lhe dá poder para controlar a qualidade de todos os artefatos de software desenvolvidos. O modelo PSP define cinco atividades estruturais:

**Planejamento.** Essa atividade isola os requisitos e desenvolve as estimativas de porte e de recursos. Além disso, faz-se uma estimativa dos defeitos (o número de defeitos estimado para o trabalho). Registram-se todas as métricas em formulários ou planilhas. Finalmente, identificam-se as tarefas de desenvolvimento e faz-se um cronograma para o projeto.

**Projeto de alto nível.** Desenvolvem-se especificações externas para cada componente a ser construído e elabora-se um projeto de componentes. Quando há incerteza, constroem-se protótipos. Todos os problemas são registrados e localizados.

**Revisão de projeto de alto nível.** Aplicam-se métodos de verificação formais (Capítulo 21) para revelar erros no projeto. Métricas são mantidas para todos os resultados de trabalho e tarefas importantes.

**Desenvolvimento.** O projeto em nível de componentes é refinado e revisado. Código é gerado, revisado, compilado e testado. Métricas são mantidas para todos os resultados de trabalho e tarefas importantes.

<sup>21</sup> Vale notar que os defensores do desenvolvimento ágil de software (Capítulo 3) também afirmam que o processo deve ficar próximo à equipe. Eles propõem um método alternativo para conseguir isso.

#### WebRef

Uma grande quantidade de recursos para PSP é encontrada em [www.ipd.uka.de/PSP/](http://www.ipd.uka.de/PSP/).

Quais atividades metodológicas são utilizadas durante o PSP?

**Autópsia.** Usando as medidas e métricas coletadas (trata-se de um volume de dados substancial que deve ser analisado estatisticamente), é determinada a eficácia do processo. Medidas e métricas devem guiar as mudanças no processo de modo a melhorar sua eficiência.

### PONTO-CHAVE

O PSP enfatiza a necessidade de registrar e analisar tipos de erros cometidos, para que se possa elaborar estratégias para eliminá-los.

O PSP enfatiza a necessidade de identificar erros precocemente e, tão importante quanto, compreender os tipos de erros que provavelmente ocorrerão. Isso é obtido por meio de uma rigorosa atividade de avaliação em todos os artefatos de software gerados.

O PSP representa uma abordagem disciplinada e baseada em métricas para a engenharia de software que pode causar um choque cultural em muitos profissionais. Entretanto, quando apresentado de forma apropriada aos engenheiros de software [Hum96], a melhoria resultante na produtividade da engenharia e na qualidade de software é significativa [Fer97]. Apesar disso, não foi adotado largamente pelo setor. Os motivos, infelizmente, têm mais a ver com a natureza humana e com a inércia organizacional do que com os pontos fortes e fracos da abordagem PSP. Esse processo é intelectualmente desafiador e exige um nível de comprometimento (por parte dos profissionais e de seus gerentes) que nem sempre é possível alcançar. O período de treinamento é relativamente longo e os custos de treinamento são altos. O nível de medição exigido é culturalmente difícil para muitos profissionais da área de software.

O PSP pode ser utilizado como um processo de software eficaz no nível pessoal? A resposta é um inequívoco "sim". Porém, mesmo se não adotado em sua totalidade, muitos dos conceitos de aperfeiçoamento do processo pessoal que introduz são importantes e vale a pena aprendê-los.

### 2.6.2 Processo de Software em Equipe (TSP)

#### WebRef

Informações sobre a formação de equipes com alto desempenho empregando-se TSP e PSP podem ser obtidas em: [www.sei.cmu.edu/tsp/](http://www.sei.cmu.edu/tsp/).

#### AVISO

Para formar uma equipe autodirigida, deve haver boa colaboração internamente e boa comunicação externamente.

O fato de muitos projetos de software para nível industrial serem tratados por uma equipe de profissionais, Watts Humphrey estendeu as lições aprendidas com a introdução do PSP e propôs um *Processo de Software em Equipe (TSP, Team Software Process)*. O objetivo do TSP é criar uma equipe de projetos "autodirigida", que se organize por si mesma para produzir software de alta qualidade. Humphrey [Hum98] define os seguintes objetivos para o TSP:

- Criar equipes autodirigidas que planejam e acompanhem seu próprio trabalho, estabeleçam metas e sejam proprietárias de seus processos e planos. As equipes poderão ser puras ou equipes de produto integradas (IPTs, integrated product teams) com cerca de 3 a 20 engenheiros.
- Mostrar aos gerentes como treinar e motivar suas equipes e como ajudá-las a manter alto desempenho.
- **Acelerar o aperfeiçoamento dos processos de software, tornando o comportamento CMM<sup>22</sup> Nível 5 algo normal e esperado.**
- Fornecer orientação para melhorias a organizações com elevado grau de maturidade.
- Facilitar o ensino universitário de habilidades de trabalho em equipe de nível industrial.

Uma equipe autodirigida possui um entendimento consistente de suas metas e objetivos globais; define papéis e responsabilidades para cada um dos membros; monitora dados quantitativos de projeto (produtividade e qualidade); identifica um processo de equipe que seja apropriado para o projeto em questão e uma estratégia para implementação do processo; define padrões locais que sejam aplicáveis ao trabalho de engenharia da equipe; avalia continuamente os riscos e reage a eles e, finalmente, acompanha, gerencia e gera relatórios sobre a situação do projeto.

O TSP define as seguintes atividades metodológicas: **lançamento do projeto, projeto de alto nível, implementação, integração e testes** e **autópsia**. Assim como seus equivalentes no PSP (note que a terminologia é ligeiramente diferente), essas atividades capacitam a equipe a planejar, projetar e construir software de maneira disciplinada, ao mesmo tempo em

22 O Modelo de Maturidade de Capacidade (CMM, Capability Maturity Model), uma medida da eficiência de um processo de software, é discutido no Capítulo 30.

que mede quantitativamente o processo e o produto. A autópsia representa o estágio para melhorias dos processos.

Esse processo faz uso de uma grande variedade de roteiros (scripts), formulários e padrões que servem para orientar os membros da equipe em seu trabalho. Os roteiros definem atividades de processos específicas (isto é, lançamento do projeto, projeto, implementação, integração e testes do sistema, autópsia) e outras funções de trabalho mais detalhadas (por exemplo, planejamento do desenvolvimento, desenvolvimento de requisitos, gerenciamento das configurações de software, teste de unidade) que fazem parte do processo de equipe.

### PONTO-CHAVE

Os roteiros (scripts) do TSP definem os elementos e as atividades realizadas no transcorrer do processo.

O TSP reconhece que as melhores equipes de software são autodirigidas.<sup>23</sup> Seus membros estabelecem os objetivos do projeto, adaptam o processo para atender suas necessidades, controlam o cronograma e, através de medições e análise das métricas coletadas, trabalham continuamente para aperfeiçoar a abordagem em relação à engenharia de software.

Assim como o PSP, o TSP é uma rigorosa abordagem da engenharia de software que fornece benefícios distintos e quantificáveis para a produtividade e para a qualidade. A equipe deve se comprometer totalmente com o processo e deve passar por treinamento consciente para assegurar que a abordagem seja apropriadamente aplicada.

## 2.7 TECNOLOGIA DE PROCESSOS

Um ou mais dos modelos de processo discutidos nas seções anteriores devem ser adaptados para ser empregados por uma equipe de software. Para tanto, desenvolveram-se *ferramentas de tecnologia de processos*, com o objetivo de auxiliar organizações de software a analisar seus processos atuais, organizar tarefas de trabalho, controlar e monitorar o progresso, bem como administrar a qualidade técnica.<sup>24</sup>

As ferramentas de tecnologia de processos permitem a uma organização de software construir um modelo automatizado da metodologia de processos, conjuntos de tarefas e atividades de apoio (*umbrella activities*), discutidos na Seção 2.1. O modelo, normalmente representado como uma rede, pode, então, ser analisado para determinar o fluxo de trabalho típico e exa-



### Ferramentas de modelagem de processos

**Objetivo:** quando uma organização trabalha para aprimorar um processo de negócio (ou de software), ela precisa, primeiramente, compreendê-lo. As ferramentas de modelagem de processos (também chamadas ferramentas de tecnologia de processos ou ferramentas de gerenciamento de processos) são usadas para representar elementos-chave de um processo para que possa ser mais bem compreendido. Essas ferramentas podem também oferecer "links" para descrições de processos, ajudando os envolvidos no processo a compreender as ações e tarefas necessárias para realizá-lo. As ferramentas de modelagem de processos fornecem links para outras ferramentas que oferecem suporte para atividades de processos definidas.

**Mecânica:** as ferramentas nesta categoria permitem a uma equipe de desenvolvimento definir os elementos de um modelo

### FERRAMENTAS DO SOFTWARE

único de processo (ações, tarefas, artefato, pontos de garantia da qualidade de software), dar orientação detalhada sobre o conteúdo ou descrição de cada elemento de um processo e, então, gerenciar o processo conforme ele for conduzido. Em alguns casos, as ferramentas de tecnologia de processos incorporam tarefas padronizadas de gerenciamento de projeto como estimativa de custos, cronograma, acompanhamento e controle.

#### Ferramentas Representativas:

**Igrafx Process Tools** — ferramentas que capacitam uma equipe a mapear, medir e modelar o processo de software ([www.micrografix.com](http://www.micrografix.com))

**Adeptia BPM Server** — projetada para gerenciar, automatizar e otimizar processos de negócio ([www.adeptia.com](http://www.adeptia.com))

**SpeedDev Suite** — conjunto de seis ferramentas com forte ênfase no gerenciamento das atividades de comunicação e modelagem ([www.speeddev.com](http://www.speeddev.com))

<sup>23</sup> No Capítulo 3 discutiremos a importância das equipes "auto-organizadas" como um elemento-chave no desenvolvimento de software ágil.

<sup>24</sup> As ferramentas aqui citadas não representam um aval, mas sim uma amostragem de ferramentas nesta categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas de seus respectivos desenvolvedores.

minar estruturas de processos alternativas que possam levar à redução de custos e tempo de desenvolvimento.

Uma vez criado um processo aceitável, outras ferramentas de tecnologia de processo poderão ser usadas para alocar, monitorar e até mesmo controlar todas as atividades, ações e tarefas de engenharia de software definidas como parte do modelo de processo. Cada membro da equipe poderá usar tais ferramentas para desenvolver uma lista de controle das tarefas a ser realizadas, dos artefatos de software a ser gerados e das atividades de garantia da qualidade a ser realizadas. A ferramenta de tecnologia de processos também pode ser usada para coordenar o uso de outras ferramentas de engenharia de software que são apropriadas para uma determinada tarefa.

## 2.8 PROCESSO DO PRODUTO

Se o processo for fraco, certamente o produto final sofrerá consequências. Porém, uma confiança excessiva e obsessiva no processo é igualmente perigosa. Em um breve artigo, escrito muitos anos atrás, Margaret Davis [Dav95a] tece comentários atemporais sobre a dualidade produto e processo:

Aproximadamente a cada dez anos (acrescente ou elimine cinco), a comunidade de software redefine "o problema", mudando seu foco de itens do produto para itens de processo. Assim, adotamos linguagens de programação estruturada (produto), seguidas por métodos de análise estruturada (processo), seguidos pelo encapsulamento de dados (produto), seguido pela ênfase atual no Modelo de Maturação da Capacidade de Desenvolvimento de Software (processo), do Software Engineering Institute [seguido por métodos orientados a objeto, seguido pelo desenvolvimento de software ágil].

Enquanto a tendência natural de um pêndulo é a de vir repousar num ponto intermediário entre dois extremos, o foco da comunidade de software muda constantemente, pois nova força é aplicada quando a última oscilação falha. Essas oscilações causam danos para si mesmos e para o ambiente externo, confundindo o profissional típico de software, mudando radicalmente o que significava desempenhar bem seu trabalho. Essas oscilações também não resolvem "o problema", pois estão fadadas ao insucesso, enquanto produto e processo forem tratados como formando uma dicotomia (divisão de um conceito em dois elementos, em geral, contrários) em vez de uma dualidade (coexistência de dois princípios).

Na comunidade científica, há precedentes da tendência para adotar noções de dualidade quando, nas observações, as contradições não podem ser explicadas completamente nem por uma nem por outra teoria que competem entre si. A natureza dual da luz, parecendo ser simultaneamente partícula e onda, foi aceita desde os anos 1920, quando Louis de Broglie a propôs. Pelas observações feitas dos artefatos de software e de seu desenvolvimento, fica demonstrada a existência de uma dualidade fundamental entre produto e processo. Jamais poderemos desctrinchar ou compreender o artefato completo, seu contexto, uso, significado e valor se o enxergarmos apenas ou como um processo ou um produto...

Todas as atividades humanas podem ser um processo, mas todos sentem-se valorizados quando tais atividades se tornam uma representação ou um exemplo, sendo utilizadas ou apreciadas por mais de uma pessoa, repetidamente, ou então utilizadas num contexto não imaginado. Ou seja, extraímos sentimentos de satisfação na reutilização de nossos produtos, seja por nós mesmos, seja por outros.

Assim, enquanto a assimilação rápida das metas de reuso, no desenvolvimento de software, aumenta potencialmente a satisfação dos profissionais de software, ela também aumenta a urgência da aceitação da dualidade produto e processo. Enxergar um artefato reutilizável apenas como um produto ou apenas como um processo, obscurece o contexto e as maneiras de usá-lo, ou obscurece o fato de que cada uso resulta em produto que, por sua vez, será utilizado como entrada, em alguma outra atividade de desenvolvimento de software. Adotar

uma dessas visões em detrimento da outra reduz dramaticamente as oportunidades de reutilização e, portanto, perde-se a oportunidade de aumentar a satisfação no trabalho.

As pessoas obtêm satisfação tanto do processo criativo quanto do produto final. Um artista sente prazer tanto de suas pinceladas quanto do resultado geral de seu quadro. Um escritor sente prazer tanto da procura da metáfora apropriada quanto do livro finalizado. Como profissional de software criativo, você também deve extrair tanta satisfação do processo como do produto final. A dualidade produto e processo é um elemento importante para manter pessoas criativas engajadas à medida que a engenharia de software continua a evoluir.

## 2.9 RESUMO

Um modelo de processo genérico para engenharia de software consiste num conjunto de atividades metodológicas e de apoio (*umbrella activities*), ações e tarefas a realizar. Cada modelo de processo, dentre os vários existentes, pode ser descrito por um fluxo de processo diferente — descrição de como as atividades metodológicas, ações e tarefas são organizadas sequencial e cronologicamente. Padrões de processo são utilizados para resolver problemas comuns encontrados como parte do processo de software.

Os modelos de processo prescritivos são aplicados há anos, num esforço para organizar e estruturar o desenvolvimento de software. Cada um desses modelos sugere um fluxo de processos ligeiramente diferente, mas todos realizam o mesmo conjunto de atividades metodológicas genéricas: comunicação, planejamento, modelagem, construção e emprego.

Os modelos de processo sequenciais, tais como o de cascata e o modelo V, são os paradigmas da engenharia de software mais antigos. Eles sugerem um fluxo de processos linear que, frequentemente, é inadequado para considerar as características dos sistemas modernos (por exemplo, contínuas alterações, sistemas em evolução, prazos apertados). Entretanto, eles têm, realmente, aplicabilidade em situações em que os requisitos são bem definidos e estáveis.

Modelos de processo incremental são iterativos por natureza e produzem rapidamente versões operacionais do software. Modelos de processos evolucionários reconhecem a natureza iterativa e incremental da maioria dos projetos de engenharia de software e são projetados para adequar mudanças. Esses modelos, como prototipação e o modelo espiral, produzem rapidamente artefatos de software incrementais (ou versões operacionais do software). Podem ser adotados para ser aplicados por todas as atividades de engenharia de software — desde o desenvolvimento de conceitos até a manutenção do sistema a longo prazo.

Modelo de processo concorrente possibilita que uma equipe de software represente elementos iterativos e concorrentes de qualquer modelo de processo. Modelos especializados incluem o modelo baseado em componentes (que enfatiza a montagem e a reutilização de componentes); o modelo de métodos formais (que encoraja uma abordagem matemática para o desenvolvimento e a verificação de software); e o modelo orientado a aspectos (que considera interesses cruzados que se estendem por toda a arquitetura do sistema). O Processo Unificado é um processo de software "dirigido a casos práticos, centrado na arquitetura, iterativo e incremental", desenvolvido como uma metodologia para os métodos e ferramentas da UML.

Modelos pessoal e de equipe enfatizam a medição, o planejamento e autodirecionamento como ingredientes-chave para um processo de software bem-sucedido.

## PROBLEMAS E PONTOS A PONDERAR

- 2.1.** Na introdução deste capítulo, Baetjer observa: "O processo oferece interação entre usuários e projetistas, entre usuários e ferramentas em evolução e entre projetistas e ferramentas [de tecnologia] em evolução". Liste cinco perguntas que (a) os projetistas deveriam fazer aos usuários, (b) os usuários deveriam fazer aos projetistas, (c) os usuários deveriam fazer a si mesmos sobre o produto de software a ser desenvolvido, (d) os projetistas deveriam fazer

a si mesmos sobre o produto de software a ser construído e sobre o processo que será usado para construí-lo.

**2.2.** Tente desenvolver um conjunto de ações para a atividade de comunicação. Selecione uma ação e defina um conjunto de tarefas para ela.

**2.3.** Durante a **comunicação**, **um problema comum ocorre ao** encontrarmos dois interessados com ideias conflitantes sobre como o software deveria ser. Isto é, há requisitos mutuamente conflitantes. Desenvolva um padrão de processo (que seja um padrão de estágio) usando o modelo apresentado na Seção 2.1.3 que se refere a esse problema, e sugira uma abordagem efetiva para ele.

**2.4.** Pesquise sobre o PSP e faça uma breve apresentação descrevendo os tipos de medidas que um engenheiro de software individual deve fazer e como tais medidas podem ser usadas para aprimorar sua eficácia pessoal.

**2.5.** O uso de roteiros (scripts — um mecanismo exigido no TSP) não é universalmente apreciado na comunidade de software. Faça uma lista dos prós e contras referentes aos roteiros e sugira pelo menos duas situações nas quais seriam úteis e outras duas onde poderiam oferecer menos benefício.

**2.6.** Leia [Nog00] e redija um artigo de duas ou três páginas que discuta o impacto do "caos" na engenharia de software.

**2.7.** Forneça três exemplos de projetos de software que seriam suscetíveis ao modelo casca-ta. Seja específico.

**2.8.** Forneça três exemplos de projetos de software que seriam suscetíveis ao modelo de prototipação. Seja específico.

**2.9.** Quais adaptações de processo seriam necessárias caso o protótipo fosse se transformar em um sistema ou produto a ser entregue?

**2.10.** Forneça três exemplos de projetos de software que seriam suscetíveis ao modelo incremental. Seja específico.

**2.11.** À medida que se desloca para fora, ao longo do fluxo de processo em espiral, o que pode ser dito em relação ao software que está sendo desenvolvido ou sofrendo manutenção?

**2.12.** É possível combinar modelos de processo? Em caso positivo, dê um exemplo.

**2.13.** O modelo de processo concorrente define um conjunto de "estados". Descreva, com suas próprias palavras, o que esses estados representam e, em seguida, indique como entram em cena no modelo de processos concorrentes.

**2.14.** Quais são as vantagens e desvantagens em desenvolver software cuja qualidade é "boa o suficiente"? Ou seja, o que acontece quando enfatizamos a velocidade de desenvolvimento em detrimento da qualidade do produto?

**2.15.** Forneça três exemplos de projetos de software que seriam suscetíveis ao modelo baseado em componentes. Seja específico.

**2.16.** É possível provar que um componente de software e até mesmo um programa inteiro está correto. Então, por que todo mundo não faz isso?

**2.17.** Processo Unificado e UML são a mesma coisa? Justifique sua resposta.

#### LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

A maioria dos livros texto sobre engenharia de software considera os modelos de processo tradicionais com certo nível de detalhe. Livros como os de Sommerville (*Software Engineering*, 8. ed., Addison-Wesley, 2006), Pfeleger e Atlee (*Software Engineering*, 3. ed., Prentice-Hall, 2005) e Schach (*Engenharia de Software: Os Paradigmas Clássico e Orientado a Objetos*, 7. ed., McGraw-Hill, 2009) falam sobre os paradigmas tradicionais e discutem seus pontos fortes e fracos. Glass (*Facts and Fallacies of Software Engineering*, Prentice-Hall, 2002) fornece

uma visão nua e crua, bem como pragmática, do processo de engenharia de software. Embora não seja especificamente dedicado a processos, Brooks (*The Mythical Man-Month*, 2. ed., Addison-Wesley, 1995) apresenta conhecimentos sábios de projeto antigo que têm tudo a ver com processos.

Firesmith e Henderson-Sellers (*The OPEN Process Framework: An Introduction*, Addison-Wesley, 2001) apresenta um quadro geral para a criação de "processos de software flexíveis e que, ainda assim, não deixam de ser disciplinados" e discute atributos e objetivos dos processos. Madachy (*Software Process Dynamics*, Wiley-IEEE, 2008) fala sobre técnicas de modelagem que possibilitam a análise dos elementos técnicos e sociais inter-relacionados do processo de software. Sharpe e McDermott (*Workflow Modeling: Tools for Process Improvement and Application Development*, Artech House, 2001) apresenta ferramentas para modelagem de processos de software, bem como de processos de negócios.

Lim (*Managing Software Reuse*, Prentice Hall, 2004) discute a reutilização sob a perspectiva gerencial. Ezran, Morisio e Tully (*Practical Software Reuse*, 2002) e Jacobson, Griss e Jonsson (*Software Reuse*, Addison-Wesley, 1997) apresentam informações muito úteis sobre desenvolvimento baseado em componentes. Heineman e Council (*Component-Based Software Engineering*, Addison-Wesley, 2001) descrevem o processo exigido para implementar sistemas baseados em componentes. Kenett e Baker (*Software Process Quality: Management and Control*, Marcel Dekker, 1999) colocam como a gestão da qualidade e o projeto de processos estão intimamente ligados entre si.

Nygard (*Release It!: Design and Deploy Production-Ready Software*, Pragmatic Bookshelf, 2007) e Richardson e Gwaltney (*Ship it! A Practical Guide to Successful Software Projects*, Pragmatic Bookshelf, 2005) apresentam um amplo conjunto de diretrizes úteis aplicáveis à atividade de emprego.

Além do livro seminal de Jacobson, Rumbaugh e Booch sobre o Processo Unificado [Jac99], livros como os de Arlow e Neustadt (*UML 2 and the Unified Process*, Addison-Wesley, 2005), Kroll e Kruchten (*The Rational Unified Process Made Easy*, Addison-Wesley, 2003) e Farve (*UML and the Unified Process*, I RM Press, 2003) fornecem excelentes informações complementares. Gibbs (*Project Management with the IBM Rational Unified Process*, IBM Press, 2006) fala sobre o gerenciamento de projetos no contexto do PU.

Uma ampla variedade de fontes de informação sobre engenharia de software e o processo de software está disponível na Internet. Uma lista atualizada de referências relevantes para o processo de software pode ser encontrada no site [www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm](http://www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm).