A natureza do software 1

Depois de me mostrar a construção mais recente de um dos games de tiro em primeira pessoa mais populares do mundo, o jovem desenvolvedor riu. "Você não joga, né?", ele perguntou.

Eu sorri. "Como adivinhou?"

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

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

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

"Então, quando essa versão estará no mercado?", perguntei.

Ele encolheu os ombros. "Em cerca de cinco meses. Ainda temos muito trabalho a fazer". Ele era responsável pela jogabilidade e pela funcionalidade de inteligência artificial de um aplicativo que abrangia mais de três milhões de linhas de código.

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

Conceitos-chave

PANORAMA

O que é? Software de computador é o produto que profissionais de software desenvolvem e ao qual dão suporte no longo prazo. Abrange programas executáveis em um computador de qualquer porte ou arquitetura, conteúdos (apresentados à medida que os programas são executados), informações descritivas tanto na forma impressa (*hard copy*) quanto na virtual, abrangendo praticamente qualquer mídia eletrônica.

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

Por que é importante? Porque afeta quase todos os aspectos de nossa vida e se difundiu no comércio, na cultura e em nossas atividades cotidianas.

Quais são as etapas envolvidas? Os clientes e outros envolvidos expressam a necessidade pelo software de computador, os engenheiros constroem o produto de software e os usuários o utilizam para resolver um problema específico ou para tratar de uma necessidade específica.

Qual é o artefato? Um programa de computador que funciona em um ou mais ambientes específicos e atende às necessidades de um ou mais usuários.

Como garantir que o trabalho foi realizado corretamente? Se você é engenheiro de software, aplique as ideias contidas no restante deste livro. Se for usuário, conheça sua necessidade e seu ambiente e escolha uma aplicação que seja a mais adequada a ambos.

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

"Onde?", perguntei, sondando. "Geralmente, nosso problema é traduzir os requisitos que os criativos nos dão". "Os criativos?", interrompi. "Você sabe, os caras que projetam a história, os personagens, todas as coisas que tornam o jogo um sucesso. Temos de pegar o que eles nos dão e produzir um conjunto de requisitos técnicos que nos permita construir o jogo."

"E depois os requisitos são fixados?"

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

"Conhece meu livro?" Eu estava sinceramente surpreso. "Claro, usei na faculdade. Há muita coisa lá."

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

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

"E como seu subconjunto mudará com o passar dos anos?"

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

"Ideias e descobertas tecnológicas são os mecanismos que impulsionam o crescimento econômico."

Wall Street Journal

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

Ninguém poderia prever que o software seria incorporado a sistemas de todas as áreas: transportes, medicina, telecomunicações, militar, industrial, entretenimento, máquinas de escritório... a lista é quase infindável. E se você

acredita na lei das consequências não intencionais, há muitos efeitos que ainda não somos capazes de prever.

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

À medida que aumenta a importância do software, a comunidade da área tenta criar tecnologias que tornem mais fácil, mais rápido e mais barato desenvolver e manter programas de computador de alta qualidade. Algumas dessas tecnologias são direcionadas a um campo de aplicação específico (por exemplo, projeto e implementação de sites); outras são focadas em um campo de tecnologia (por exemplo, sistemas orientados a objetos ou programação orientada a aspectos); e outras ainda são de bases amplas (por exemplo, sistemas operacionais como o Linux). Entretanto, nós ainda temos de desenvolver uma tecnologia de software que faça tudo isso – e a probabilidade de surgir tal tecnologia no futuro é pequena. Ainda assim, as pessoas apostam seus empregos, seu conforto, sua segurança, seu entretenimento, suas decisões e suas próprias vidas em software. Tomara que estejam certas.

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

1.1 A natureza do software

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

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

O software distribui o produto mais importante de nossa era – a *informação*. Ele transforma dados pessoais (por exemplo, transações financeiras de um indivíduo) de modo que possam ser mais úteis em determinado contexto; gerencia informações comerciais para aumentar a competitividade; fornece um portal para redes mundiais de informação (Internet) e os meios para obter informações sob todas as suas formas. Também propicia um veículo que pode ameaçar a privacidade pessoal e é uma porta que permite a pessoas mal-intencionadas cometer crimes.

"Software é um lugar onde sonhos são plantados e pesadelos são colhidos, um pântano abstrato e místico onde demônios terríveis competem com mágicas panaceias, um mundo de lobisomens e balas de prata."

Brad J. Cox

O papel do software passou por uma mudança significativa no decorrer da metade final do século passado. Aperfeiçoamentos significativos no desempenho do hardware, mudanças profundas nas arquiteturas computacionais, um vasto aumento na capacidade de memória e armazenamento e uma ampla variedade exótica de opções de entrada e saída; tudo isso resultou em sistemas computacionais mais sofisticados e complexos. Sofisticação e complexidade podem produzir resultados impressionantes quando um sistema é bem-sucedido; porém, também podem trazer enormes problemas para aqueles que precisam desenvolver e projetar sistemas robustos.

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

- Por que a conclusão de um software leva tanto tempo?
- Por que os custos de desenvolvimento são tão altos?
- Por que não conseguimos encontrar todos os erros antes de entregarmos o software aos clientes?
- Por que gastamos tanto tempo e esforço realizando a manutenção de programas existentes?
- Por que ainda temos dificuldades de medir o progresso de desenvolvimento e a manutenção de um software?

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

1.1.1 Definição de software

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

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

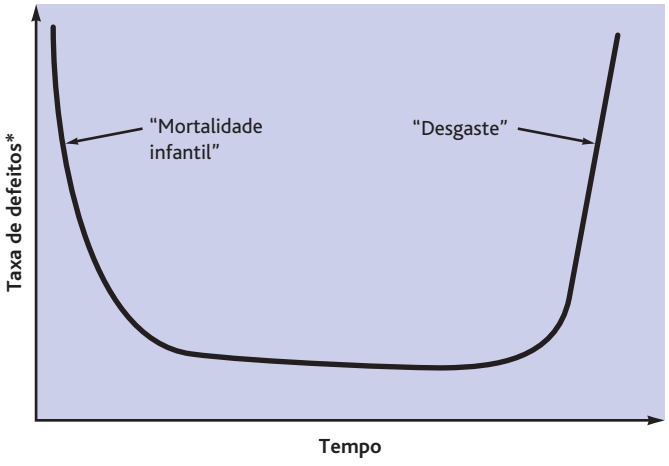
Como devemos definir *software*?

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

Sem dúvida, poderíamos dar outras definições mais completas, mas, provavelmente, uma definição mais formal não melhoraria, consideravelmente, a compreensão do que é software.

---

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



Caso queira reduzir a deterioração do software, terá de fazer um projeto de software melhor (Capítulos 12 a 18).

FIGURA 1.1 Curva de defeitos para hardware.

Para conseguir isso, é importante examinar as características do software que o tornam diferenciado de outras coisas que os seres humanos constroem. Software é mais um elemento de sistema lógico do que físico. Portanto, o software tem uma característica fundamental que o torna consideravelmente diferente do hardware: *software não "se desgasta"*.

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

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

Essa aparente contradição pode ser elucidada pela curva real apresentada na Figura 1.2. Durante sua vida[2], o software passará por alterações. À

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

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

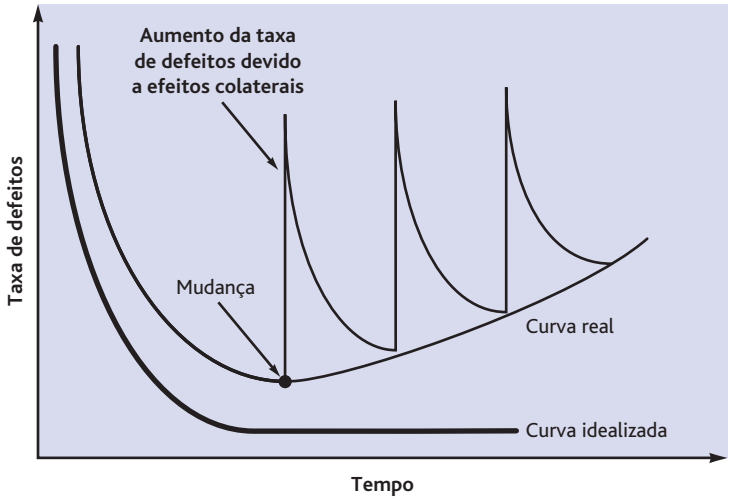


FIGURA 1.2 Curva de defeitos para software.

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

medida que elas ocorram, é provável que sejam introduzidos erros, fazendo com que a curva de taxa de defeitos se acentue, conforme mostrado na "curva real" (Figura 1.2). Antes que a curva possa retornar à taxa estável original, outra alteração é requisitada, fazendo com que a curva se acentue novamente. Lentamente, o nível mínimo da taxa começa a aumentar – o software está deteriorando devido à modificação.

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

1.1.2 Campos de aplicação de software

Atualmente, sete grandes categorias de software apresentam desafios contínuos para os engenheiros de software:

Software de sistema Conjunto de programas feito para atender a outros programas. Certos softwares de sistema (por exemplo, compiladores, editores e utilitários para gerenciamento de arquivos) processam estruturas de informação complexas; porém, determinadas.[3] Outras aplicações de sistema (por exemplo, componentes de sistema operacional, drivers, software de rede, processadores de telecomunicações) processam dados amplamente indeterminados.

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

Software de aplicação Programas independentes que solucionam uma necessidade específica de negócio. Aplicações nessa área processam dados comerciais ou técnicos de uma forma que facilite operações comerciais ou tomadas de decisão administrativas/técnicas.

Uma das mais abrangentes bibliotecas de shareware/freeware (software compartilhado/livre) pode ser encontrada em **shareware.cnet.com**.

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

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

Software para linha de produtos Projetado para prover capacidade específica de utilização por muitos clientes diferentes. Software para linha de produtos pode se concentrar em um mercado hermético e limitado (por exemplo, produtos de controle de inventário) ou lidar com consumidor de massa.

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

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

"Para mim, o computador é a ferramenta mais extraordinária que já inventamos. É o equivalente de uma bicicleta para nossas mentes."

Steve Jobs

Milhões de engenheiros de software em todo o mundo trabalham arduamente em projetos de software em uma ou mais dessas categorias. Em alguns casos, novos sistemas estão sendo construídos, mas, em muitos outros, aplicações já existentes estão sendo corrigidas, adaptadas e aperfeiçoadas. Não é incomum um jovem engenheiro de software trabalhar em um programa mais velho do que ele! Gerações passadas de pessoal de software deixaram um legado em cada uma das categorias discutidas. Espera-se que o legado a ser deixado por esta geração facilite o trabalho dos futuros engenheiros de software.

1.1.3 Software legado

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

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

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

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

O que faço caso encontre um sistema legado de baixa qualidade?

Infelizmente, de vez em quando uma característica adicional está presente em software legado – a *baixa qualidade*.[4] Às vezes, os sistemas legados têm projetos inextensíveis, código de difícil entendimento, documentação deficiente ou inexistente, casos de teste e resultados que nunca foram documentados, um histórico de alterações mal gerenciado – a lista pode ser bastante longa. Ainda assim, esses sistemas dão suporte a "funções vitais de negócio e são indispensáveis para ele". O que fazer?

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

Que tipos de mudanças são feitas em sistemas legados?

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

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

Quando essas modalidades de evolução ocorrem, um sistema legado deve passar por reengenharia (Capítulo 36) para que permaneça viável no futuro. O objetivo da engenharia de software moderna é "elaborar metodologias baseadas na noção de evolução"; isto é, na noção de que os sistemas de software modificam-se continuamente, novos sistemas são construídos a partir dos antigos e... todos devem agir em grupo e cooperar uns com os outros" [Day99].

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

1.2 A natureza mutante do software

A evolução de quatro categorias amplas de software domina o setor. Ainda assim, há pouco mais de uma década essas categorias estavam em sua infância.

1.2.1 WebApps

Nos primórdios da World Wide Web (por volta de 1990 a 1995), os *sites* eram formados por nada mais do que um conjunto de arquivos de hipertexto linkados e que apresentavam informações usando texto e gráficos limitados. Com o tempo, o crescimento da linguagem HTML, via ferramentas de desenvolvimento (por exemplo, XML, Java), tornou possível aos engenheiros da Internet oferecerem capacidade computacional juntamente com as informações. Nasciam, então, os *sistemas e aplicações baseados na Web*[5] (refiro-me a eles coletivamente como *WebApps*).

Atualmente, as WebApps evoluíram para sofisticadas ferramentas computacionais que não apenas oferecem funções especializadas (*stand-alone functions*) ao usuário, como também foram integradas aos bancos de dados corporativos e às aplicações de negócio.

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

Louis Monier

Há uma década, as WebApps "envolvem(iam) uma mistura de publicação impressa e desenvolvimento de software, de marketing e computação, de comunicações internas e relações externas e de arte e tecnologia" [Pow98]. Mas, hoje, fornecem potencial de computação total em muitas das categorias de aplicação registradas na seção 1.1.2.

No decorrer da década passada, tecnologias Semantic Web (muitas vezes denominadas Web 3.0) evoluíram para sofisticadas aplicações corporativas e para o consumidor, as quais abrangem "bancos de dados semânticos [que] oferecem novas funcionalidades que exigem links Web, representação [de dados] flexível e APIs de acesso externo" [Hen10]. Sofisticadas estruturas de dados relacionais levarão a WebApps inteiramente novas que permitirão acessar informações díspares de maneiras inéditas.

1.2.2 Aplicativos móveis

O termo *aplicativo* evoluiu para sugerir software projetado especificamente para residir em uma plataforma móvel (por exemplo, iOS, Android ou Windows Mobile). Na maioria dos casos, os aplicativos móveis contêm uma interface de usuário que tira proveito de mecanismos de interação exclusivos fornecidos pela plataforma móvel, da interoperabilidade com recursos baseados na Web que dão acesso a uma grande variedade de informações relevantes ao aplicativo e de capacidades de processamento local que coletam, analisam e formatam as informações de forma mais conveniente para a plataforma. Além disso, um aplicativo móvel fornece recursos de armazenamento persistente dentro da plataforma.

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

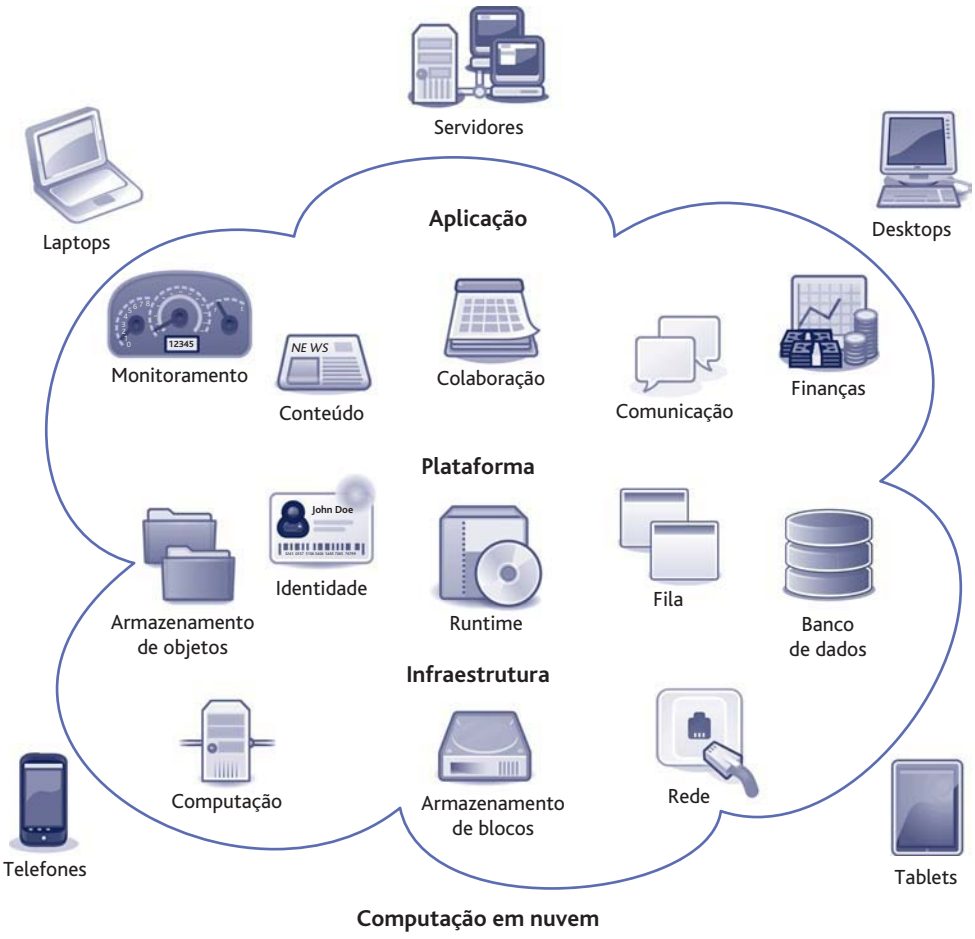


FIGURA 1.3 Arquitetura lógica da computação em nuvem [Wik13].

Qual a diferença entre uma WebApp e um aplicativo móvel?

É importante reconhecer que existe uma diferença sutil entre aplicações web móveis e aplicativos móveis. Uma *aplicação web móvel* (WebApp) permite que um dispositivo móvel tenha acesso a conteúdo baseado na web por meio de um navegador especificamente projetado para se adaptar aos pontos fortes e fracos da plataforma móvel. Um *aplicativo móvel* pode acessar diretamente as características do hardware do dispositivo (por exemplo, acelerômetro ou localização por GPS) e, então, fornecer os recursos de processamento e armazenamento local mencionados anteriormente. Com o passar do tempo, essa diferença entre WebApps móveis e aplicativos móveis se tornará indistinta, à medida que os navegadores móveis se tornarem mais sofisticados e ganharem acesso ao hardware e às informações em nível de dispositivo.

1.2.3 Computação em nuvem

A *computação em nuvem* abrange uma infraestrutura ou "ecossistema" que permite a qualquer usuário, em qualquer lugar, utilizar um dispositivo de computação para compartilhar recursos computacionais em grande escala. A arquitetura lógica global da computação em nuvem está representada na Figura 1.3.

De acordo com a figura, os dispositivos de computação residem fora da nuvem e têm acesso a uma variedade de recursos dentro dela. Esses recursos abrangem aplicações, plataformas e infraestrutura. Em sua forma mais simples, um dispositivo de computação externa acessa a nuvem por meio de um navegador Web ou software semelhante. A nuvem dá acesso a dados residentes nos bancos de dados e em outras estruturas de dados. Além disso, os dispositivos podem acessar aplicativos executáveis, que podem ser usados no lugar das aplicações residentes no dispositivo de computação.

A implementação da computação em nuvem exige o desenvolvimento de uma arquitetura que contenha serviços de front-end e de back-end. O *front-end* inclui o dispositivo cliente (usuário) e o software aplicativo (por exemplo, um navegador) que permite o acesso ao back-end. O *back-end* inclui servidores e recursos de computação relacionados, sistemas de armazenamento de dados (por exemplo, bancos de dados), aplicativos residentes no servidor e servidores administrativos que utilizam middleware para coordenar e monitorar o tráfego, estabelecendo um conjunto de protocolos de acesso à nuvem e aos seus recursos residentes [Str08].

A arquitetura da nuvem pode ser segmentada para dar acesso a uma variedade de diferentes níveis de acesso público total para arquiteturas de nuvem privadas, acessíveis somente a quem tenha autorização.

1.2.4 Software para linha de produtos (de software)

O Software Engineering Institute define uma *linha de produtos de software* como "um conjunto de sistemas de software que compartilham um conjunto comum de recursos gerenciados, satisfazendo as necessidades específicas de um segmento de mercado ou de uma missão em particular, desenvolvidos a partir de um conjunto comum de itens básicos, contemplando uma forma prescrita" [SEI13]. De certa maneira, a noção de linha de produtos de software relacionados não é nova, mas a ideia de uma linha de produtos de software – todos desenvolvidos com a mesma aplicação subjacente e com as mesmas arquiteturas de dados, sendo todos implementados com um conjunto de componentes de software reutilizáveis em toda a linha de produtos – proporciona um potencial significativo para a engenharia.

Uma linha de produtos de software compartilha um conjunto de itens que incluem requisitos (Capítulo 8), arquitetura (Capítulo 13), padrões de projeto (Capítulo 16), componentes reutilizáveis (Capítulo 14), casos de teste (Capítulos 22 e 23) e outros produtos de trabalho para a engenharia de software. Basicamente, uma linha de produtos de software resulta no desenvolvimento de muitos produtos projetados tirando proveito dos atributos comuns a tudo que é feito dentro da linha de produtos.

1.3 Resumo

Software é o elemento-chave na evolução de produtos e sistemas baseados em computador e é uma das mais importantes tecnologias no cenário mundial. Ao longo dos últimos 50 anos, o software evoluiu de uma ferramenta especializada em análise de informações e resolução de problemas para uma indústria

propriamente dita. Mesmo assim, ainda temos problemas para desenvolver software de boa qualidade dentro do prazo e orçamento estabelecidos.

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

A natureza do software é mutante. As aplicações e os sistemas baseados na Internet passaram de simples conjuntos de conteúdo informativo para sofisticados sistemas que apresentam funcionalidade complexa e conteúdo multimídia. Embora essas WebApps possuam características e requisitos exclusivos, elas não deixam de ser um tipo de software. Os aplicativos móveis apresentam novos desafios, à medida que migram para uma ampla variedade de plataformas. A computação em nuvem transformará o modo de distribuir software e o seu ambiente. O software para linha de produtos oferece eficiências em potencial na maneira de construir software.

Problemas e pontos a ponderar

1.1 Dê no mínimo mais cinco exemplos de como a lei das consequências não intencionais se aplica a software de computador.

1.2 Forneça uma série de exemplos (positivos e negativos) que indiquem o impacto do software em nossa sociedade.

1.3 Dê suas próprias respostas para as cinco perguntas feitas no início da Seção 1.1. Discuta-as com seus colegas.

1.4 Muitas aplicações modernas mudam frequentemente – antes de serem apresentadas ao usuário e depois da primeira versão ser colocada em uso. Sugira algumas maneiras de construir software para impedir a deterioração decorrente de mudanças.

1.5 Considere as sete categorias de software apresentadas na Seção 1.1.2. Você acha que a mesma abordagem em relação à engenharia de software pode ser aplicada a cada uma delas? Justifique sua resposta.

Leituras e fontes de informação complementares[6]

Literalmente milhares de livros são escritos sobre software. A grande maioria trata de linguagens de programação ou aplicações de software, porém poucos tratam do software em si. Pressman e Herron (*Software Shock*, Dorset House, 1991) apresentaram uma discussão preliminar (dirigida ao grande público) sobre software e a maneira como os profissionais o desenvolvem. O best-seller de Negroponte (*Being Digital*, Alfred A. Knopf, 1995) dá uma visão geral da computação e seu impacto global no século 21. DeMarco (*Why Does Software Cost So Much*? Dorset House, 1995) produziu um conjunto de divertidos e perspicazes ensaios sobre software e o processo pelo qual ele é desenvolvi-

[6] A seção *Leitura e fontes de informação complementares* ao final de cada capítulo apresenta uma visão geral de publicações que podem ajudar a expandir o seu entendimento dos principais tópicos apresentados no capítulo. Criamos um site completo (em inglês) para dar suporte a este livro, no endereço **www.mhhe.com/pressman**. Entre os muitos temas tratados no site estão recursos de engenharia de software, capítulo por capítulo, até informações baseadas na Web que podem complementar o material apresentado em cada capítulo. Dentro desses recursos existe um link para a Amazon.com para cada livro mencionado nesta seção.

do. Ray Kurzweil (*How to Create a Mind,* Viking, 2013) discute como em breve o software imitará o pensamento humano e levará a uma "singularidade" na evolução de humanos e máquinas.

Keeves (*Catching Digital,* Business Infomedia Online, 2012) discute como os líderes empresariais devem se adaptar, à medida que o software evolui em um ritmo cada vez maior. Minasi (*The Software Conspiracy: Why Software Companies Put out Faulty Products, How They Can Hurt You, and What You Can Do*, McGraw-Hill, 2000) argumenta que o "flagelo moderno" dos bugs de software pode ser eliminado e sugere maneiras para concretizar isso. Eubanks (*Digital Dead End: Fighting for Social Justice in the Information Age,* MIT Press, 2011) e Compaine (*Digital Divide: Facing a Crisis or Creating a Mith*, MIT Press, 2001) defendem que a "separação" entre aqueles que têm acesso a fontes de informação (por exemplo, a Web) e aqueles que não o têm está diminuindo, à medida que avançamos na primeira década deste século. Kuniavsky (*Smart Things: Ubiquitous Computing User Experience Design,* Morgan Kaufman, 2010), Greenfield (*Everyware: The Dawning Age of Ubiquitous Computing*, New Riders Publishing, 2006) e Loke (*Context-Aware Pervasive Systems*: *Architectures for a New Breed of Applications*, Auerbach, 2006) introduzem o conceito de software "aberto" e preveem um ambiente sem fio no qual o software deve se adaptar às exigências que surgem em tempo real.

Uma ampla variedade de fontes de informação que discutem a natureza do software está disponível na Internet. Uma lista atualizada de referências relevantes (em inglês) para o processo do software pode ser encontrada no site: **www.mhhe.com/pressman.**

2 Engenharia de software

Conceitos-chave

Para desenvolver software que esteja preparado para enfrentar os desafios do século 21, devemos admitir alguns fatos:

- Software está profundamente incorporado em praticamente todos os aspectos de nossas vidas e, consequentemente, o número de pessoas interessadas nos recursos e nas funções oferecidas por determinada aplicação[1] tem crescido significativamente. *Depreende-se, portanto, que é preciso fazer um esforço conjunto para compreender o problema antes de desenvolver uma solução de software.*
- Os requisitos de tecnologia da informação demandados por pessoas, empresas e órgãos governamentais estão mais complexos a cada ano. Atualmente, equipes grandes desenvolvem programas de computador que, antigamente, eram desenvolvidos por um só indivíduo. Software sofisticado, outrora implementado em um ambiente computacional independente e previsível, hoje está incorporado em tudo, de produtos eletrônicos de consumo a equipamentos médicos e sistemas de armamento. *Depreende-se, portanto, que projetar se tornou uma atividade essencial.*

PANORAMA

O que é? A engenharia de software abrange um processo, um conjunto de métodos (práticas) e um leque de ferramentas que possibilitam aos profissionais desenvolverem software de altíssima qualidade.

Quem realiza? Os engenheiros de software aplicam o processo de engenharia de software.

Por que é importante? A engenharia de software é importante porque nos capacita para o desenvolvimento de sistemas complexos dentro do prazo e com alta qualidade. Ela impõe disciplina a um trabalho que pode se tornar caótico, mas também permite que as pessoas produzam software de computador adaptado à sua abordagem, da maneira mais conveniente às suas necessidades.

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

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

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

[1] Neste livro, mais adiante, chamaremos tais pessoas de "envolvidos".

- Pessoas, negócios e governos dependem, cada vez mais, de software para a tomada de decisões estratégicas e táticas, assim como para controle e para operações cotidianas. Se o software falhar, as pessoas e as principais empresas poderão ter desde pequenos inconvenientes até falhas catastróficas. *Depreende-se, portanto, que um software deve apresentar qualidade elevada.*

Entenda o problema antes de elaborar uma solução.

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

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

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

2.1 Definição da disciplina

O IEEE [IEE93a] elaborou a seguinte definição para engenharia de software:

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

Como definimos engenharia de software?

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

A engenharia de software é uma tecnologia em camadas. Como ilustra a Figura 2.1, qualquer abordagem de engenharia (inclusive engenharia de software) deve estar fundamentada em um comprometimento organizacional com a qualidade. A gestão da qualidade total Seis Sigma e filosofias similares[2] promovem uma cultura de aperfeiçoamento contínuo de processos, e é essa cultura que, no final das contas, leva ao desenvolvimento de abordagens cada vez mais eficazes na engenharia de software. A pedra fundamental que sustenta a engenharia de software é o foco na qualidade.

A base da engenharia de software é a camada de *processos*. O processo de engenharia de software é a liga que mantém as camadas de tecnologia coesas e possibilita o desenvolvimento de software de forma racional e dentro do prazo. O processo define uma metodologia que deve ser estabelecida para a entrega efetiva de tecnologia de engenharia de software. O processo de software constitui a base para o controle do gerenciamento de projetos de software e estabelece o contexto no qual são aplicados métodos técnicos, são produzidos

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

[2] A gestão da qualidade e as metodologias relacionadas são discutidas ao longo da Parte III deste livro.

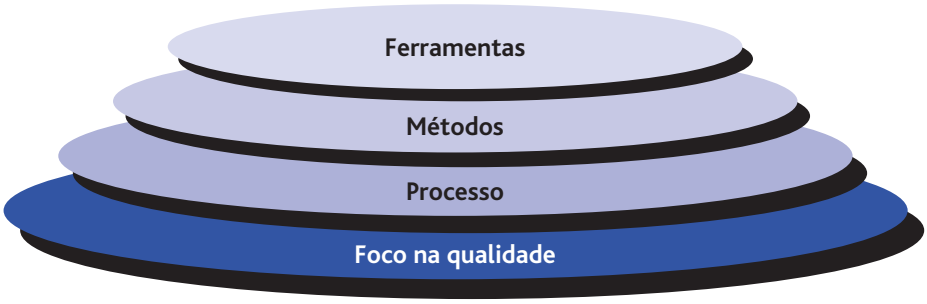


FIGURA 2.1 Camadas da engenharia de software.

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

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

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

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

2.2 O processo de software

Quais são os elementos de um processo de software?

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

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

Ivar Jacobson, Grady Booch e James Rumbaugh

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

2.2.1 A metodologia do processo

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

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

Comunicação. Antes que qualquer trabalho técnico possa começar, é de importância fundamental se comunicar e colaborar com o cliente (e outros envolvidos).[3] A intenção é entender os objetivos dos envolvidos para o projeto e reunir requisitos que ajudem a definir os recursos e as funções do software.

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

Fred Brooks

Planejamento. Qualquer jornada complicada pode ser simplificada com auxílio de um mapa. Um projeto de software é uma jornada complicada, e a atividade de planejamento cria um "mapa" que ajuda a guiar a equipe na sua jornada. O mapa – denominado plano de projeto de software – define o trabalho de engenharia de software, descrevendo as tarefas técnicas a serem conduzidas, os riscos prováveis, os recursos que serão necessários, os produtos resultantes a ser produzidos e um cronograma de trabalho.

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

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

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

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

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

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

2.2.2 Atividades de apoio

As atividades metodológicas do processo de engenharia de software são complementadas por diversas *atividades de apoio*. De modo geral, as atividades de apoio são aplicadas por todo um projeto de software e ajudam uma equipe de software a gerenciar e a controlar o andamento, a qualidade, as alterações e os riscos. As atividades de apoio típicas são:

Atividades de apoio ocorrem ao longo do processo de software e se concentram, principalmente, em gerenciamento, acompanhamento e controle do projeto.

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

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

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

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

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

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

Gerenciamento da capacidade de reutilização – define critérios para a reutilização de artefatos (inclusive componentes de software) e estabelece mecanismos para a obtenção de componentes reutilizáveis.

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

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

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

2.2.3 Adaptação do processo

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

- Fluxo geral de atividades, ações e tarefas e suas interdependências.
- Até que ponto as ações e tarefas são definidas dentro de cada atividade da metodologia.
- Até que ponto artefatos de software são identificados e exigidos.
- Modo de aplicar as atividades de garantia da qualidade.
- Modo de aplicar as atividades de acompanhamento e controle do projeto.
- Grau geral de detalhamento e rigor da descrição do processo.
- Grau de envolvimento com o projeto (por parte do cliente e de outros envolvidos).
- Nível de autonomia dada à equipe de software.
- Grau de prescrição da organização da equipe.

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

Madame Benoit

A Parte I deste livro examina o processo de software com um grau de detalhamento considerável.

2.3 A prática da engenharia de software

A Seção 2.2 apresentou uma introdução a um modelo de processo de software genérico, composto por um conjunto de atividades que estabelecem uma metodologia para a prática da engenharia de software. As atividades genéricas da metodologia – **comunicação, planejamento, modelagem, construção** e **entrega** –, bem como as atividades de apoio, estabelecem um esquema para o trabalho da engenharia de software. Mas como a prática da engenharia de software se encaixa nisso? Nas seções seguintes, você vai adquirir um conhecimento básico dos princípios e conceitos genéricos que se aplicam às atividades de uma metodologia.[4]

Diversas citações instigantes sobre a prática da engenharia de software podem ser encontradas em **www.literate-programming.com**.

2.3.1 A essência da prática

No livro clássico *How to Solve It*, escrito antes de os computadores modernos existirem, George Polya [Pol45] descreveu em linhas gerais a essência da solução de problemas e, consequentemente, a essência da prática da engenharia de software:

1. *Compreender o problema* (comunicação e análise).
2. *Planejar uma solução* (modelagem e projeto de software).
3. *Executar o plano* (geração de código).
4. *Examinar o resultado para ter precisão* (testes e garantia da qualidade).

Pode-se afirmar que a abordagem de Polya é simplesmente uma questão de bom senso. É verdade. Mas é espantoso como o bom senso é tão pouco usado no mundo do software.

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

Compreenda o problema. Algumas vezes é difícil de admitir; porém, a maioria de nós é arrogante quando um problema nos é apresentado. Ouvimos por

O elemento mais importante para se entender um problema é escutar.

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

alguns segundos e então pensamos: "*Ah, sim, estou entendendo, vamos começar a resolver este problema*". Infelizmente, compreender nem sempre é assim tão fácil. Vale a pena despender um pouco de tempo respondendo a algumas perguntas simples:

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

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

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

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

George Polya

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

- *A solução é adequada ao plano?* O código-fonte pode ser atribuído ao modelo de projeto?
- *Todas as partes componentes da solução estão provavelmente corretas?* O projeto e o código foram revistos ou, melhor ainda, provas da correção foram aplicadas ao algoritmo?

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

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

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

2.3.2 Princípios gerais

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

David Hooker [Hoo96] propôs sete princípios que se concentram na prática da engenharia de software como um todo. Eles são reproduzidos nos parágrafos a seguir:[5]

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

Primeiro princípio: ***a razão de existir***

Um sistema de software existe por um motivo: *agregar valor para seus usuários*. Todas as decisões devem ser tomadas com esse princípio em mente. Antes de especificar um requisito de um sistema, antes de indicar alguma parte da funcionalidade de um sistema, antes de determinar as plataformas de hardware ou os processos de desenvolvimento, pergunte a si mesmo: "Isso realmente agrega valor real ao sistema?". Se a resposta for "não", não o faça. Todos os demais princípios se apoiam neste primeiro.

"Há certa majestade na simplicidade, que está muito acima de toda a excentricidade do saber."

Alexandre Pope (1688-1744)

Segundo princípio: ***KISS (Keep It Simple, Stupid!, ou seja: não complique!)***

O projeto de software não é um processo casual. Existem muitos fatores a considerar em qualquer trabalho de projeto. *Todo projeto deve ser o mais simples possível, mas não simplista*. Esse princípio contribui para um sistema mais fácil de compreender e manter. Isso não significa que características, até mesmo as internas, devam ser descartadas em nome da simplicidade. De fato, os projetos mais elegantes normalmente são os mais simples. Simples também não significa "gambiarra". Na verdade, muitas vezes são necessárias muitas reflexões e trabalho em várias iterações para simplificar. A contrapartida é um software mais fácil de manter e menos propenso a erros.

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

Uma visão clara é essencial para o sucesso. Sem ela, um projeto se torna ambíguo. Sem uma integridade conceitual, corre-se o risco de transformar o projeto em uma colcha de retalhos de projetos incompatíveis, unidos por parafusos inadequados... Comprometer a visão arquitetural de um sistema de software debilita e até poderá destruir sistemas bem projetados. Ter um ar-

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

quiteto responsável e capaz de manter a visão clara e de reforçar a adequação ajuda a assegurar o êxito de um projeto.

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

Quarto princípio: *o que um produz outros consomem*

Raramente um sistema de software de qualidade industrial é construído e utilizado de forma isolada. De uma maneira ou de outra, alguém mais vai usar, manter, documentar ou, de alguma forma, depender da capacidade de entender seu sistema. Portanto, *sempre especifique, projete e implemente ciente de que mais alguém terá de entender o que você está fazendo*. O público para qualquer produto de desenvolvimento de software é potencialmente grande. Especifique tendo como objetivo os usuários. Projete tendo em mente os implementadores. Codifique se preocupando com aqueles que deverão manter e ampliar o sistema. Alguém terá de depurar o código que você escreveu, e isso o torna um usuário de seu código. Facilitando o trabalho de todas essas pessoas, você agrega maior valor ao sistema.

Quinto princípio: *esteja aberto para o futuro*

Um sistema com tempo de vida mais longo tem mais valor. Nos ambientes computacionais de hoje, em que as especificações mudam de um instante para outro, e as plataformas de hardware se tornam rapidamente obsoletas, a vida de um software, em geral, é medida em meses. Contudo, os verdadeiros sistemas de software com "qualidade industrial" devem durar muito mais. Para serem bem-sucedidos nisso, esses sistemas precisam estar prontos para se adaptar a essas e outras mudanças. Sistemas que obtêm sucesso são aqueles que foram projetados dessa forma desde seu princípio. *Jamais faça projetos limitados*. Sempre pergunte "e se" e prepare-se para todas as respostas possíveis, criando sistemas que resolvam o problema geral, não apenas o específico.[6] Isso muito provavelmente conduziria à reutilização de um sistema inteiro.

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

A reutilização economiza tempo e esforço.[7] Alcançar um alto grau de reutilização é indiscutivelmente a meta mais difícil de ser atingida ao se desenvolver um sistema de software. A reutilização de código e projetos tem sido proclamada como uma grande vantagem do uso de tecnologias orientadas a objetos. Contudo, o retorno desse investimento não é automático. Aproveitar as possibilidades de reutilização – oferecidas pela programação orientada a objetos (ou convencional) – exige planejamento e capacidade de fazer previsões. Existem várias técnicas para levar a cabo a reutilização em cada um dos níveis do processo de desenvolvimento do sistema... *Planejar com antecedên-*

[6] Esse conselho pode ser perigoso se levado ao extremo. Projetar para o "problema geral" algumas vezes exige comprometer o desempenho e pode tornar ineficientes as soluções específicas.

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

cia para a reutilização reduz o custo e aumenta o valor tanto dos componentes reutilizáveis quanto dos sistemas aos quais eles serão incorporados.

Sétimo princípio: *pense!*

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

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

2.4 Mitos do desenvolvimento de software

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

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

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

Software Project Managers Network, em www.spmn.com, pode ajudá-lo a refutar esses e outros mitos.

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

Realidade: O livro com padrões pode muito bem existir, mas ele é realmente utilizado? Os praticantes da área estão cientes de que ele existe? Esse livro reflete a prática moderna da engenharia de software? É completo? É adaptável? Está otimi-

zado para melhorar o tempo de entrega, mantendo ainda o foco na qualidade? Em muitos casos, a resposta para todas essas perguntas é "não".

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

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

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

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

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

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

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

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

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

Realidade: É verdade que os requisitos de software mudam, mas o impacto da mudança varia dependendo do momento em que foi introduzida. Quando as mudanças dos requisitos são solicitadas cedo (antes de o projeto ou de a codificação te-

rem começado), o impacto sobre os custos é relativamente pequeno.[8] Entretanto, conforme o tempo passa, ele aumenta rapidamente – recursos foram comprometidos, uma estrutura de projeto foi estabelecida e mudar pode causar uma revolução que exija recursos adicionais e modificações fundamentais no projeto.

Mitos dos profissionais da área. Mitos que ainda sobrevivem entre os profissionais da área têm resistido por mais de 60 anos de cultura da programação. Durante seus primórdios, a programação era vista como uma forma de arte. Hábitos e atitudes antigos são difíceis de perder.

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

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

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

Mito: *Até que o programa esteja "em execução", não há como avaliar sua qualidade.*

Realidade: Um dos mecanismos de garantia da qualidade de software mais eficientes pode ser aplicado a partir da concepção de um projeto – *a revisão técnica*. Os revisores de software (descritos no Capítulo 20) são "filtros de qualidade", considerados mais eficientes do que os testes feitos para encontrar certas classes de defeitos de software.

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

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

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

Realidade: O objetivo da engenharia de software não é criar documentos. É criar um produto de qualidade. Uma qualidade melhor leva à redução do retrabalho. E retrabalho reduzido resulta em tempos de entrega menores.

Atualmente, muitos profissionais de software reconhecem a falácia dos mitos que acabamos de descrever. Estar ciente das realidades do software é o primeiro passo para buscar soluções práticas na engenharia de software.

[8] Muitos engenheiros de software têm adotado uma abordagem "ágil" que favorece as alterações incrementais, controlando, com isso, seu impacto e seu custo. Os métodos ágeis são discutidos no Capítulo 5.

2.5 Como tudo começa

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

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

CASASEGURA[9]

Como começa um projeto

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

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

Conversa:

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

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

Joe: Você concorda, Mal?

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

Joe: Ampla em que sentido?

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

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

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

Joe: Interessante. Mas eu perguntei qual é o ponto principal.

Mal: PCs e tablets estão em mais de 70% dos lares nos EUA. Se pudermos acertar no preço, essa aplicação poderá ser excepcional. Ninguém mais tem nosso dispositivo sem fio... ele é exclusivo! Estaremos dois anos à frente de nossos concorrentes... e as receitas? Algo em torno de 30 a 40 milhões no segundo ano...

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

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

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

software incorporado atender adequadamente às necessidades (ainda não declaradas) do cliente. Acompanharemos a evolução da engenharia do software *CasaSegura* em vários dos capítulos que estão por vir.

2.6 Resumo

A engenharia de software engloba processos, métodos e ferramentas que possibilitam a construção de sistemas complexos baseados em computador dentro do prazo e com qualidade. O processo de software incorpora cinco atividades estruturais: comunicação, planejamento, modelagem, construção e entrega, e elas se aplicam a todos os projetos de software. A prática da engenharia de software é uma atividade de resolução de problemas que segue um conjunto de princípios básicos.

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

Problemas e pontos a ponderar

2.1. A Figura 2.1 coloca as três camadas de engenharia de software acima de uma camada intitulada "foco na qualidade". Isso implica um programa de qualidade organizacional como o de gestão da qualidade total. Pesquise um pouco a respeito e crie um sumário dos princípios básicos de um programa de gestão da qualidade total.

2.2. A engenharia de software é aplicável na construção de WebApps? Em caso positivo, como poderia ser modificada para atender às características únicas das WebApps?

2.3. À medida que o software invade todos os setores, riscos ao público (devido a programas com imperfeições) passam a ser uma preocupação cada vez maior. Crie um cenário o mais catastrófico possível, porém realista, em que a falha de um programa de computador poderia causar um grande dano em termos econômicos ou humanos.

2.4. Descreva uma metodologia de processo com suas próprias palavras. Ao afirmarmos que atividades de modelagem se aplicam a todos os projetos, isso significa que as mesmas tarefas são aplicadas a todos os projetos, independentemente de seu tamanho e complexidade? Explique.

2.5. As atividades de apoio ocorrem ao longo do processo de software. Você acredita que elas são aplicadas de forma homogênea ao longo do processo ou algumas delas são concentradas em uma ou mais atividades da metodologia?

2.6. Acrescente mais dois mitos à lista apresentada na Seção 2.4. Declare também a realidade que acompanha o mito.

Leituras e fontes de informação complementares

O estado atual da engenharia de software e do processo de software pode ser mais bem determinado a partir de publicações como *IEEE Software*, *IEEE Computer*, *CrossTalk* e *IEEE*

Transactions on Software Engineering. Periódicos do setor, como *Application Development Trends* e *Cutter IT Journal*, normalmente contêm artigos sobre tópicos da engenharia de software. A disciplina é "sintetizada" todos os anos no *Proceeding of the International Conference on Software Engineering*, patrocinado pelo *IEEE* e *ACM*, e é discutida de forma aprofundada em periódicos como *ACM Transactions on Software Engineering and Methodology*, *ACM Software Engineering Notes* e *Annals of Software Engineering*. Dezenas de milhares de páginas Web são dedicadas à engenharia de software e ao processo de software.

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

SWEBOK: Guide to the Software Engineering Body of Knowledge, [10] IEEE, 2013, consulte: http:// www.computer.org/portal/web/swebok

Andersson, E., *et al*., *Software Engineering for Internet Applications,* MIT Press, 2006.

Braude, E. e M. Bernstein, *Software Engineering: Modern Approaches,* 2ª ed., Wiley, 2010.

Christensen, M. e R. Thayer, *A Project Manager's Guide to Software Engineering Best Practices,* IEEE-CS Press (Wiley), 2002.

Glass, R., *Fact and Fallacies of Software Engineering,* Addison-Wesley, 2002.

Hussain, S., *Software Engineering,* I K International Publishing House, 2013.

Jacobson, I., *Object-Oriented Software Engineering: A Use Case Driven Approach,* 2ª ed., Addison-Wesley, 2008.

Jalote, P., *An Integrated Approach to Software Engineering,* 3ª ed., Springer, 2010.

Pfleeger, S., *Software Engineering: Theory and Practice,* 4ª ed., Prentice Hall, 2009.

Schach, S., *Object-Oriented and Classical Software Engineering,* 8ª ed., McGraw-Hill, 2010.

Sommerville, I., *Software Engineering,* 9ª ed., Addison-Wesley, 2010.

Stober, T. e U. Hansmann, *Agile Software Development: Best Practices for Large Development Projects,* Springer, 2009.

Tsui, F. e O. Karam, *Essentials of Software Engineering,* 2ª ed., Jones & Bartlett Publishers, 2009.

Nygard (*Release It!: Design and Deploy Production-Ready Software,* Pragmatic Bookshelf, 2007), Richardson e Gwaltney (*Ship it! A Practical Guide to Successful Software Projects,* Pragmatic Bookshelf, 2005) e Humble e Farley (*Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley, 2010) apresentam uma ampla coleção de diretrizes úteis, aplicáveis à atividade de entrega.

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

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 (em inglês) para o processo para o software pode ser encontrada no site: **www.mhhe.com/pressman.**

[10] Disponível gratuitamente em **<http://www.computer.org/portal/web/swebok/htmlformat>.**

PARTE

I

O processo de software

Nesta parte do livro, você vai aprender sobre o processo que fornece uma metodologia para a prática da engenharia de software. Estas questões são tratadas nos capítulos que seguem:

- 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 se diferencia dos modelos de processos mais tradicionais?

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

3 Estrutura do processo de software

Conceitos-chave

Em um livro fascinante que apresenta a visão de um economista sobre software e engenharia de software, Howard Baetjer Jr. [Bae98] comenta o processo de software:

> Porque o software, como todo capital, é conhecimento incorporado, e porque esse conhecimento é, 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 se tornar o software, é coletado, reunido e incorporado ao software. O 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 extraindo 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 à medida que se desenvolve o processo.

Mas, do ponto de vista técnico, o que é exatamente um processo de software? No contexto deste livro, definimos *processo de software* como uma metodologia para as atividades, ações e tarefas necessárias para desenvolver

PANORAMA

O que é? Quando se elabora 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? 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 é importante? Porque propicia estabilidade, controle e organização para uma atividade que pode se tornar bastante caótica sem controle. Entretanto, uma abordagem de engenharia de software moderna deve ser "ágil". Deve demandar apenas atividades, controles e artefatos 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. Determinado processo pode ser adequado para um software do sistema de voo 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 artefatos são os programas, os documentos e os dados produzidos pelas atividades e tarefas definidas pelo processo.

Como garantir que o trabalho foi realizado corretamente? Há muitos mecanismos de avaliação que permitem às empresas determinarem o nível de "maturidade" de seu processo de software. Entretanto, a qualidade, o cumprimento de prazos e a viabilidade em longo prazo do produto que se desenvolve são os melhores indicadores da eficácia do processo utilizado.

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. Entretanto, 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 amplo conhecimento, e que devem adaptar um processo de software maduro de modo que fique adequado aos produtos desenvolvidos e às demandas de seu mercado.

3.1 Um modelo de processo genérico

No Capítulo 2, 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 se alocam dentro de uma metodologia ou modelo que determina sua relação com o processo e umas com as outras.

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

O processo de software está representado esquematicamente na Figura 3.1. 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 2, 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 é aplicado 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, entre outras.

O que é fluxo de processo?

Um aspecto importante 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 3.2.

Um *fluxo de processo linear* executa cada uma das cinco atividades metodológicas em sequência, começando com a comunicação e culminando com a entrega (Figura 3.2a). Um *fluxo de processo iterativo* repete uma ou mais das atividades antes de prosseguir para a seguinte (Figura 3.2b). Um *fluxo de processo evolucionário* executa as atividades de forma "circular". Cada volta pelas cinco atividades conduz a uma versão mais completa do software (Figura 3.2c). Um *fluxo de processo paralelo* (Figura 3.2d) executa uma ou mais atividades em paralelo com outras (por exemplo, a modelagem para um aspecto do software poderia ser executada em paralelo com a construção de outro aspecto do software).

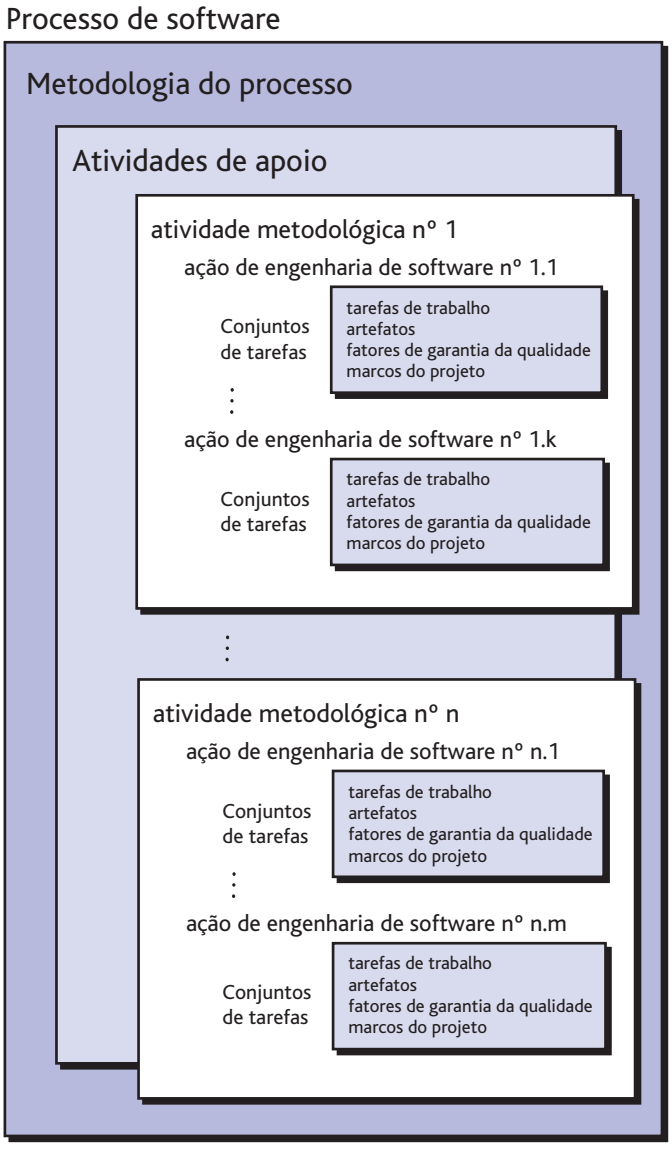


FIGURA 3.1 Uma metodologia do processo de software.

3.2 Definição de uma atividade metodológica

"Se o processo estiver correto, os resultados falarão por si mesmos."

Takashi Osada

Embora cinco atividades metodológicas tenham sido descritas e tenha-se fornecido uma definição básica de cada uma delas no Capítulo 2, uma equipe de software precisa de muito mais informações antes de poder executar qualquer uma das atividades como parte do processo de software. Assim, enfrenta-se uma questão-chave: *Quais ações são apropriadas para uma atividade metodológica, uma vez fornecidos a natureza do problema a ser solucionado, as características das pessoas que farão o trabalho e os envolvidos no 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 (em um local distante) com requisitos simples e objetivos, a atividade de **comunicação** pode se resumir a pouco mais de um telefonema ou email para o

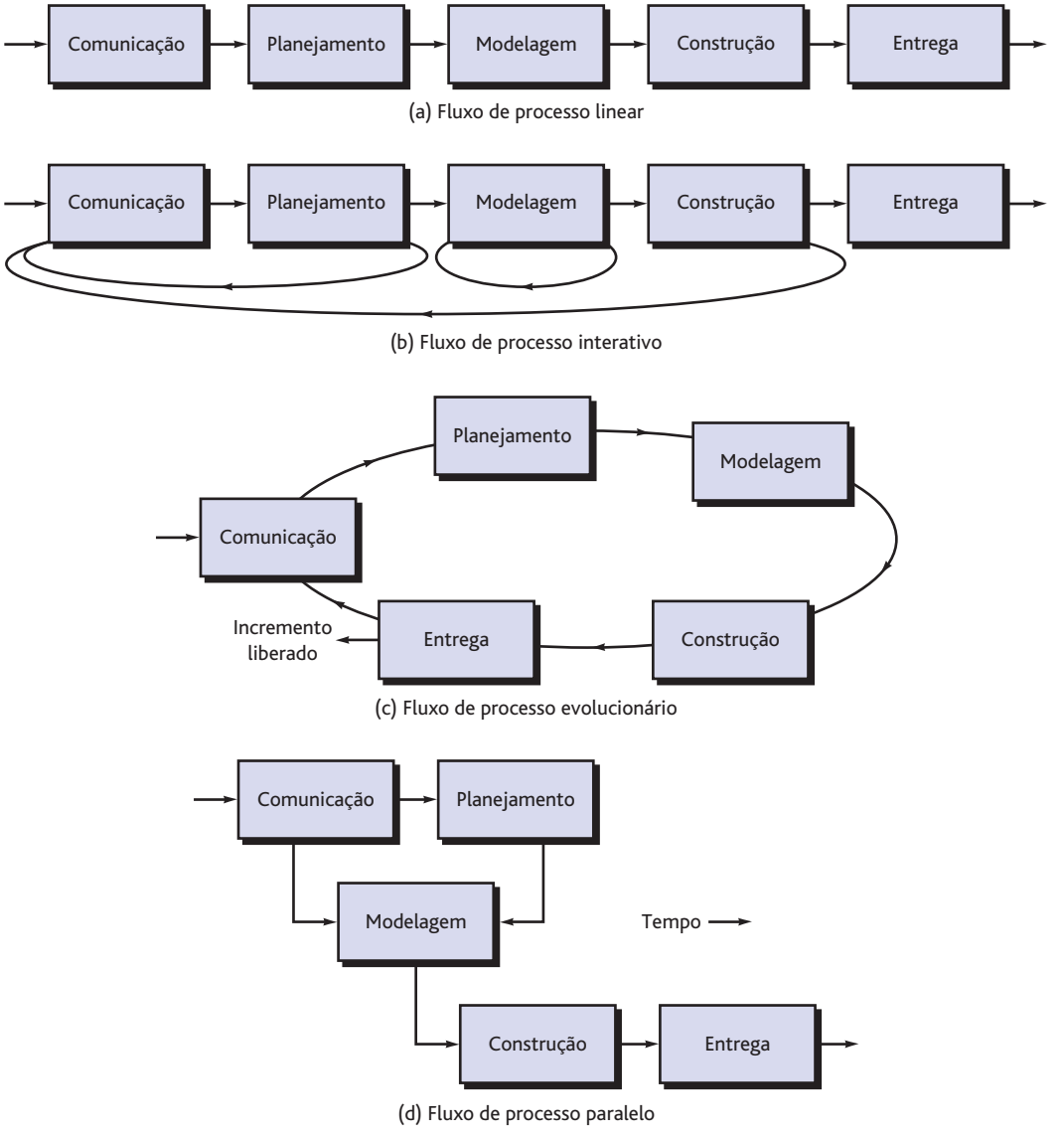


FIGURA 3.2 Fluxo de processo.

envolvido. Portanto, a única ação necessária é uma *conversa telefônica*, e as tarefas de trabalho (o *conjunto de tarefas*) que essa ação envolve são:

1. Contatar o envolvido via telefone.
2. Discutir os requisitos e gerar anotações.
3. Organizar as anotações em uma breve relação de requisitos, por escrito.
4. Enviar um email para o envolvido para revisão e aprovação.

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

Se o projeto fosse consideravelmente mais complexo, com muitos envolvidos, cada qual com um conjunto de requisitos diferentes (por vezes conflitantes), a atividade de comunicação poderia ter seis ações distintas (descritas no Capítulo 8): *concepção*, *levantamento*, *elaboração*, *negociação*, *especificação* e *validação*. Cada uma dessas ações de engenharia de software conteria muitas tarefas de trabalho e uma série de diferentes artefatos.

INFORMAÇÕES

Conjunto de tarefas

Um conjunto de tarefas define o trabalho a ser feito para atingir os objetivos de uma ação de engenharia de software. Por exemplo, *levantamento* (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 envolvidos esperam do software a ser desenvolvido.

Para um projeto pequeno e relativamente simples, o conjunto de tarefas para levantamento dos requisitos seria semelhante a este:

1. Fazer uma lista dos envolvidos no projeto.
2. Fazer uma reunião informal com todos os envolvidos.
3. Solicitar a cada envolvido uma lista com as características e funções necessárias.
4. Discutir sobre os requisitos e elaborar 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, é preciso usar um conjunto diferente de tarefas. Esse 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.
3. Fazer uma lista preliminar das funções e características, com base nas informações fornecidas pelos envolvidos.
4. Agendar uma série de reuniões facilitadoras para especificação de aplicações.
5. Realizar reuniões.
6. Incluir cenários informais de usuários como parte de cada reunião.
7. Refinar os cenários de usuários, com base no feedback dos envolvidos.
8. Fazer uma lista revisada dos requisitos dos envolvidos.
9. Empregar técnicas de implantação de funções de qualidade para estabelecer graus de prioridade dos requisitos.
10. Agrupar os requisitos de modo que possam ser entregues em incrementos.
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 do "levantamento de requisitos"; porém, são bem diferentes quanto ao seu grau de profundidade e formalidade. A equipe de software deve escolher o conjunto de tarefas que possibilite atingir o objetivo de cada ação, mantendo, inclusive, a qualidade e a agilidade.

3.3 Identificação de um conjunto de tarefas

Voltando novamente à Figura 3.1, cada ação de engenharia de software (por exemplo, *levantamento*, uma ação associada à atividade de **comunicação**) pode ser representada por vários e diferentes *conjuntos de tarefas* – constituídos por uma gama de tarefas de trabalho de engenharia de software, artefatos relacionados, fatores de garantia da qualidade e marcos 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.

3.4 Padrões de processo

O que é padrão de processo?

Toda equipe de desenvolvimento encontra problemas à medida que avança no processo de software. Seria útil se soluções comprovadas estivessem pronta-

mente à disposição da equipe, de modo que os problemas pudessem ser localizados e resolvidos rapidamente. Um *padrão de processo*[1] 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 [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.

"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

Padrões podem ser definidos em qualquer nível de abstração.[2] 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 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:

Nome do padrão. O padrão deve receber um nome 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.

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

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 várias ações e tarefas de trabalho, um padrão de estágio engloba vários padrões de tarefas (veja o próximo padrão) relevantes ao estágio (atividade metodológica). Um exemplo de padrão de estágio poderia ser **EstabelecimentoDeComunicação**. Esse padrão poderia incorporar o padrão de tarefas **LevantamentoDeRequisitos** 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, **LevantamentoDeRequisitos** 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**.[3]

[1] Uma discussão detalhada sobre padrões é apresentada no Capítulo 11.

[2] Os padrões são aplicáveis a várias atividades de engenharia de software. Padrões de análise, de projeto e de testes são discutidos nos Capítulos 11, 13, 15, 16 e 20. Padrões e "antipadrões" para atividades de gerenciamento de projetos são discutidos na Parte IV deste livro.

[3] Esses padrões de fases são discutidos no Capítulo 4.

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

Tom DeMarco

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 é o estado inicial do processo? (3) Que informação de engenharia de software ou de projeto já existe?

Por exemplo, o padrão **Planejamento** (um padrão de estágio) exige 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 e as restrições do projeto, bem como os requisitos básicos do negócio.

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 bem-sucedida do padrão.

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 do processo? (3) Quais informações de engenharia de software ou de projeto foram desenvolvidas?

Padrões relativos. Fornecem 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**, **LevantamentoDeRequisitos**, **DescriçãoDasRestrições** e **CriaçãoDeCenários**.

Muitos recursos sobre padrões de processo podem ser encontrados em **www.ambysoft.com/processPatternsPage.html**.

Usos conhecidos e exemplos. Indicam as instâncias específicas a que 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 **Entrega** estiver em andamento.

Padrões de processo propiciam um mecanismo eficaz para a localização de problemas associados a qualquer processo de software. Os padrões permitem que se desenvolva uma descrição do 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 na definição de variantes do 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 do processo.

INFORMAÇÕES

Um exemplo de padrão de processo

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

Nome do padrão. **RequisitosImprecisos**

Intuito. Esse padrão descreve uma abordagem voltada à construção de um modelo (um protótipo) passível de ser avaliado iterativamente pelos envolvidos, em um esforço para identificar ou solidificar requisitos do software.

Tipo. Padrão de fase.

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

Problema. Os requisitos são vagos ou inexistentes, ainda assim há o reconhecimento claro de que existe um problema a ser solucionado e ele deve ser identificado utilizando-se uma solução de software. Os envolvidos 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 – e está disponível posteriormente, na Seção 4.1.3.

Contexto resultante. Um protótipo de software que identifique os requisitos básicos (por exemplo, modos de interação, características computacionais, funções de processamento) é aprovado pelos envolvidos. 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 processos.

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

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

3.5 Avaliação e aperfeiçoamento de processos

A avaliação tenta compreender o atual estado do processo de software com o intuito de aperfeiçoá-lo.

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 resultarão em qualidade de longo prazo (Capítulo 19). Os padrões de processo devem ser combinados com uma prática de engenharia de software confiável (Parte II deste livro). Além disso, o próprio processo pode ser avaliado para que esteja de acordo com um conjunto de critérios de processo básicos, comprovados como essenciais para uma engenharia de software bem-sucedida.[4]

Ao longo das últimas décadas foi proposta uma série de diferentes abordagens de avaliação e 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].

[4] 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.

"As empresas de software têm mostrado grandes deficiências em tirar proveito das experiências adquiridas com projetos executados."

NASA

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 37.

3.6 Resumo

Um modelo de processo genérico para engenharia de software consiste em um conjunto de atividades metodológicas e de apoio, ações e tarefas a realizar. Cada modelo de processo, entre os vários existentes, pode ser descrito por um fluxo de processo diferente – uma descrição de como as atividades metodológicas, ações e tarefas são organizadas, sequencial e cronologicamente. Padrões de processo são utilizados para resolver problemas comuns encontrados no processo de software.

Problemas e pontos a ponderar

3.1. 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 (1) os projetistas deveriam fazer aos usuários, (2) os usuários deveriam fazer aos projetistas, (3) os usuários deveriam fazer a si mesmos sobre o produto de software a ser desenvolvido, (4) 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.

3.2. Discuta as diferenças entre os vários fluxos de processo descritos na Seção 3.1. Consegue identificar tipos de problemas que poderiam ser aplicáveis a cada um dos fluxos genéricos descritos?

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

3.4. Durante a **comunicação**, um problema comum ocorre ao encontrarmos dois envolvidos com ideias conflitantes sobre como o software deve ser. Isto é, há requisitos mutuamente conflitantes. Desenvolva um padrão de processo (que seja um padrão de estágio) usando o modelo apresentado na Seção 3.4 que trata desse problema e sugira uma abordagem eficaz para ele.

Leituras e fontes de informação complementares

A maioria dos livros-texto sobre engenharia de software considera os modelos de processo com certo nível de detalhe. Livros de Sommerville (*Software Engineering,* 9ª ed., Addison-Wesley, 2010), Schach (*Object-Oriented and Classical Software Engineering,* 8ª ed., McGraw-Hill, 2010) e Pfleeger e Atlee (*Software Engineering: Theory and Practice,* 4ª ed., Prentice Hall, 2009) consideram os paradigmas tradicionais e discutem suas vantagens e desvantagens. Munch e seus colegas (*Software Process Definition and Management,* Springer, 2012) apresentam uma visão do processo e do produto da engenharia de software e de sistemas. Glass (*Facts and Fallacies of Software Engineering*, Prentice Hall, 2002) fornece uma visão simples e pragmática do processo de engenharia de software. Embora não seja especificamente dedicado a processos, Brooks (*The Mythical Man-Month*, 2ª 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) apresentam um quadro geral para a criação de "processos de software flexíveis e que, ainda assim, não deixam de ser disciplinados" e discutem 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 relacionados do processo de software. Sharpe e McDermott (*Workflow Modeling: Tools for Process Improvement and Application Development*, 2ª ed., Artech House, 2008) apresentam ferramentas para modelagem de processos de software e de negócios.

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 (em inglês) para o processo para o software pode ser encontrada no site: **www.mhhe.com/pressman.**

4 Modelos de processo

Conceitos-chave

Originalmente, os modelos de processo foram propostos para trazer ordem ao caos existente na área de desenvolvimento de software. A história demonstra que esses modelos fazem uma considerável contribuição à estrutura utilizável no trabalho de engenharia de software e fornecem um roteiro razoavelmente eficaz para as equipes de software. Entretanto, o trabalho de engenharia de software e os produtos gerados 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 comprometimento 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.
>
> Tendemos a pensar que a 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 permite que a mudança seja organizada, mas tal estrutura não deve ser tão rígida a ponto de impedir que a mudança ocorra. Por outro lado, caos em demasia pode

PANORAMA

O que é? Um modelo de processo fornece um guia específico para o trabalho de engenharia de software. Ele define o fluxo de todas as atividades, ações e tarefas, o grau de iteração, os artefatos e a organização do trabalho a ser feito.

Quem realiza? Os engenheiros de software e seus gerentes adaptam um modelo de 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 é importante? Porque o processo 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 gerado.

Quais são as etapas envolvidas? O modelo de processo fornece os "passos" necessários para realizar um trabalho de engenharia de software disciplinado.

Qual é o artefato? Do ponto de vista de um engenheiro de software, o artefato é uma descrição personalizada das atividades e tarefas definidas pelo processo.

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

impossibilitar a coordenação e a coerência. A falta de estrutura nem sempre implica desordem.

A finalidade dos modelos de processo é tentar reduzir o caos presente no desenvolvimento de novos produtos de software.

As implicações filosóficas desse argumento são importantes para a engenharia de software. Cada modelo de processo descrito neste capítulo tenta encontrar um equilíbrio entre a necessidade de pôr ordem em um mundo caótico e a de ser adaptável quando as coisas mudam constantemente.

4.1 Modelos de processo prescritivo

Um *modelo de processo prescritivo*[1] concentra-se em estruturar e ordenar o desenvolvimento de software. As atividades e tarefas ocorrem sequencialmente, com diretrizes de progresso definidas. Mas os modelos prescritivos são adequados para o mundo do software que se alimenta de mudanças? Se rejeitarmos os modelos de processo tradicionais (e a ordem implícita) e os substituirmos por algo menos estruturado, tornaremos impossível atingir a coordenação e a coerência no trabalho de software?

Um premiado "jogo de simulação de processos", que inclui os mais importantes modelos de processo prescritivos, pode ser encontrado em: **http://www.ics.uci.edu/~emilyo/SimSE/downloads.html**.

Não há respostas fáceis para essas questões, mas existem alternativas disponíveis para os engenheiros de software. Nas próximas seções, examinamos a abordagem dos processos prescritivos, nos quais a ordem e a consistência do projeto são questões predominantes. Chamamos esses processos de "prescritivos" porque prescrevem um conjunto de elementos de processo – atividades metodológicas, ações de engenharia de software, tarefas, artefatos, 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 relacionados.

Todos os modelos de processo de software podem acomodar as atividades metodológicas genéricas descritas nos Capítulos 2 e 3; 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.

4.1.1 O modelo cascata

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

Há casos em que os requisitos de um problema são bem compreendidos – quando o trabalho flui da comunicação à disponibilização de modo 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.

[1] Os modelos de processo prescritivos são, algumas vezes, conhecidos como modelos de processo "tradicionais".

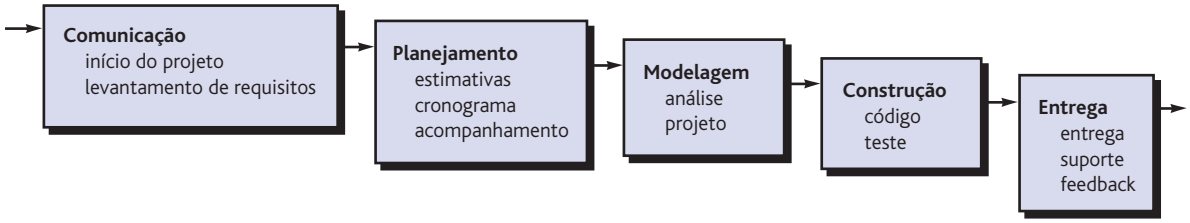


FIGURA 4.1 O modelo cascata.

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

O *modelo cascata*, algumas vezes chamado *ciclo de vida clássico*, sugere uma abordagem sequencial[2] e sistemática para o desenvolvimento de software, começando com a especificação dos requisitos do cliente, avançando pelas fases de planejamento, modelagem, construção e disponibilização, e culminando no suporte contínuo do software concluído (Figura 4.1).

Uma variação na representação do modelo cascata é denominada *modelo V*. Representado na Figura 4.2, o modelo V [Buc99] descreve a relação entre ações de garantia da qualidade e ações associadas a 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 cada vez mais detalhadas e técnicas do problema e de sua solução. Uma vez gerado o código, a equipe passa para o lado direito do V, basicamente realizando uma série de testes (ações de garantia da qualidade) que validam cada um dos modelos criados à medida que a equipe desce pelo lado esquerdo.[3] Na realidade, não há nenhuma diferença fundamental entre o ciclo de vida clássico e o modelo V. O modelo V oferece uma maneira de visualizar como as ações de verificação e validação são aplicadas a um trabalho de engenharia anterior.

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

Por que algumas vezes o modelo cascata falha?

1. Projetos reais raramente seguem o fluxo sequencial proposto pelo modelo. 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 exige isso e tem dificuldade para adequar a incerteza natural existente 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 ao final do projeto. Um

[2] 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.

[3] Na Parte III deste livro, é apresentada uma discussão detalhada sobre ações de garantia da qualidade.

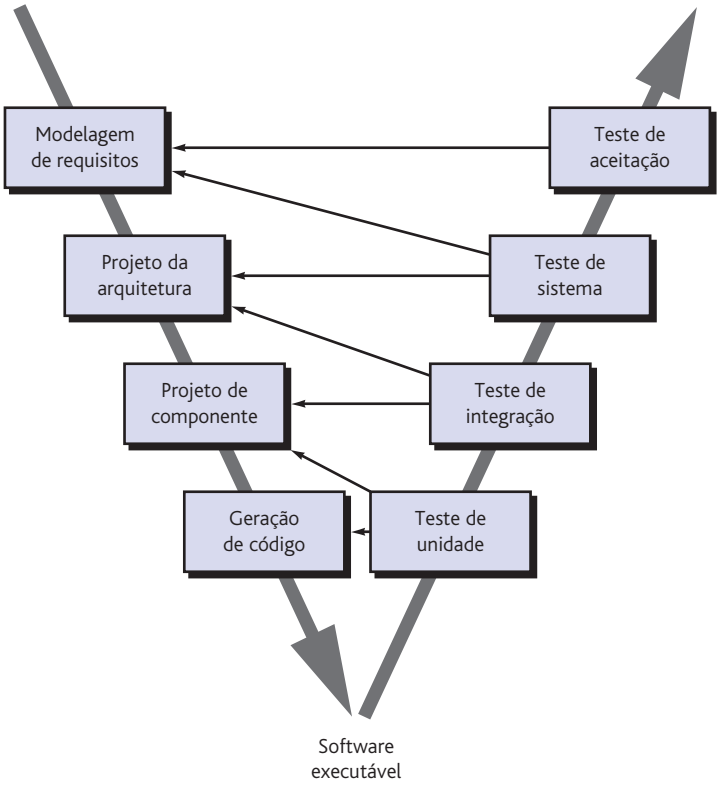


FIGURA 4.2 Modelo V.

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 de projeto têm de aguardar outros completarem tarefas dependentes. O tempo gasto na espera pode exceder o tempo gasto em trabalho produtivo! O estado de bloqueio tende a prevalecer no início e no final de um processo sequencial linear.

"Muito frequentemente, o trabalho de software segue a primeira lei do ciclismo: não importa aonde se esteja indo, é sempre ladeira acima e contra o vento."

Autor desconhecido

Hoje, o trabalho com 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 inadequado para esse 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.

4.1.2 Modelos de processo incremental

Há várias situações em que os requisitos iniciais do software são razoavelmente bem definidos; entretanto, o escopo geral do trabalho de desenvolvimento, impede o uso de um processo puramente linear. Pode ser necessário o rápido fornecimento de 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 libera uma série de versões, denominadas incrementos, que oferecem, progressivamente, maior funcionalidade ao cliente à medida que cada incremento é entregue.

O modelo incremental combina os fluxos de processo linear e paralelo dos elementos, discutidos no Capítulo 3. Na Figura 4.3, o modelo incremental aplica sequências lineares de forma escalonada, à medida que o tempo vai avançando. Cada sequência linear produz "incrementos" entregáveis do software [McD93].

Por exemplo, um software de processamento de textos 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, discutido na próxima subseção.

Seu cliente exige a entrega em uma data impossível de atender. Sugira entregar um ou mais incrementos nessa data e o restante do software (incrementos adicionais) posteriormente.

Quando se utiliza um modelo incremental, frequentemente o primeiro incremento é um *produto essencial*. Ou seja, 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 gerado o produto completo.

4.1.3 Modelos de processo evolucionário

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

Como todos os sistemas complexos, software evolui ao longo do tempo. Conforme o desenvolvimento do projeto avança, os requisitos do negócio e do produto frequentemente mudam, tornando inadequado seguir um planejamento em linha reta de um produto final. Prazos apertados, determinados pelo mer-

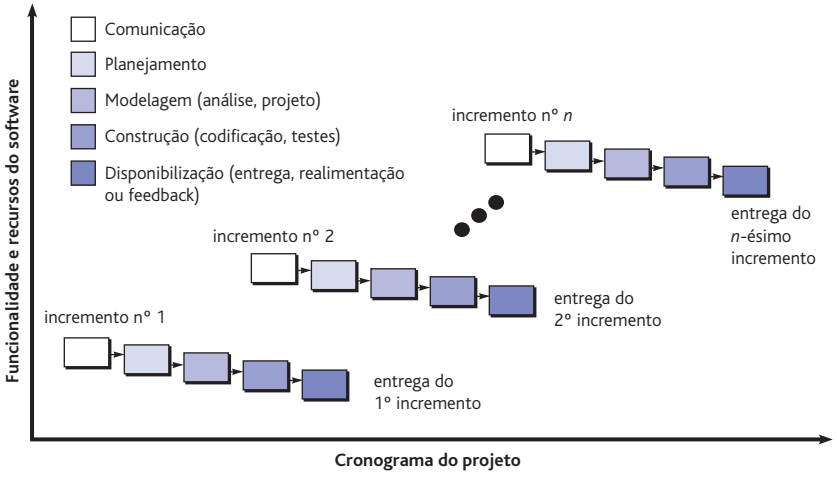


FIGURA 4.3 O modelo incremental.

cado, tornam impossível concluir 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 dos requisitos 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 cresce e muda.

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

"Planeje jogar algo fora. Você vai fazer isso de qualquer maneira. Sua escolha consistirá em decidir se deve tentar ou não vender aos clientes o que foi descartado."

Frederick P. Brooks

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 se encontra inseguro quanto à eficiência de um algoritmo, quanto à adaptabilidade de um sistema operacional ou quanto à forma em que deve ocorrer a interação homem-máquina. Em situações como essas, e em muitas outras, o *paradigma da prototipação* pode ser a melhor abordagem.

Embora a prototipação possa ser utilizada como um modelo de processo isolado (*stand-alone process*), ela é mais comumente utilizada como uma técnica a 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 envolvidos a compreender melhor o que está para ser construído.

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

O paradigma da prototipação (Figura 4.4) começa com a comunicação. Faz-se uma reunião com os envolvidos para definir os objetivos gerais do software, identificar os requisitos já 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

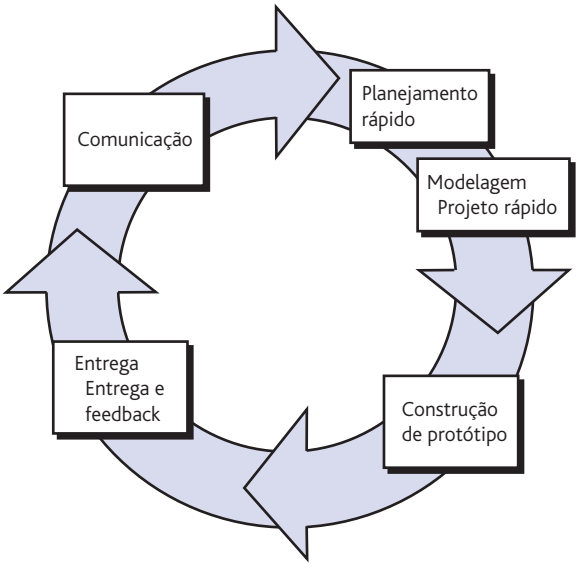


FIGURA 4.4 O paradigma da prototipação.

"projeto rápido"). Um projeto rápido se concentra em uma representação dos aspectos do software que serão visíveis para os usuários (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. O protótipo é entregue e avaliado pelos envolvidos, os quais fornecem feedback que é usado para refinar ainda mais os requisitos. A iteração ocorre conforme se ajusta o protótipo às necessidades de vários envolvidos 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 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 útil. Pode ser lento demais, grande demais, estranho em sua utilização ou as três coisas juntas. Não há alternativa, a não ser começar de novo – ressentido, porém mais esperto – e desenvolver uma versão reformulada 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 transformarem no sistema real.

Tanto os envolvidos quanto os engenheiros de software gostam do paradigma da prototipação. Os usuários podem ter uma ideia prévia do sistema final, ao passo que os desenvolvedores passam a desenvolver algo imediatamente. Entretanto, a prototipação pode ser problemática pelas seguintes razões:

Resista à pressão de transformar um protótipo grosseiro em um produto final. Quase sempre a qualidade fica comprometida.

1. Os envolvidos 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 em longo prazo. Quando informados de que o produto deve ser reconstruído para que altos níveis de qualidade possam ser mantidos, os envolvidos 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 uma linguagem de programação inadequada 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, é possível se acomodar com tais escolhas e esquecer todas as razões pelas quais eram inadequadas. Uma escolha longe da ideal acaba se tornando parte do sistema.

CASASEGURA

Seleção de um modelo de processo, parte 1

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

Atores: 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.

Conversa:

Lee: Recapitulando. Discuti bastante sobre a linha de produtos *CasaSegura*, da forma como a visualizamos no momento. Sem dúvida, temos muito trabalho a fazer para definir as coisas, mas eu gostaria que vocês começassem a pensar em como vão abordar a parte do software desse projeto.

Doug: Acho que fomos bastante desorganizados em nossa abordagem de software no passado.

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

Doug: É, mas não sem grande sofrimento, e esse projeto parece ser maior e mais complexo do que qualquer outro que já fizemos.

Jamie: Não parece assim tão difícil, mas eu concordo... a abordagem improvisada que adotamos em 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 dizer não. Eis o que quero dizer. (Doug prossegue descrevendo a metodologia de processo descrita no Capítulo 3 e os modelos de processo prescritivo apresentados até agora.)

Doug: De qualquer forma, parece-me que um modelo linear não é adequado para nós... ele presume 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: Concordo.

Ed: Essa abordagem de prototipação me parece boa. Bastante parecida com o que fazemos aqui.

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

Doug: Não se preocupe. Temos várias opções e quero que vocês escolham o que for melhor para a equipe e para o projeto.

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

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

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.

> O modelo espiral de desenvolvimento é um gerador de *modelos de processos* dirigidos *a riscos* e é utilizado para guiar a engenharia de sistemas com muito software, que ocorre de forma concorrente e tem vários envolvidos. Possui duas características principais que o distinguem. A primeira consiste em uma estratégia *cíclica* voltada para ampliar, de forma incremental, o grau de definição e a imple-

mentação de um sistema, enquanto diminui o grau de risco do mesmo. A segunda característica é que possui uma série de *marcos de pontos-âncora* para garantir o comprometimento dos envolvidos quanto à busca de soluções de sistema que sejam mutuamente satisfatórias e viáveis.

Com 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.

O modelo espiral é dividido em um conjunto de atividades metodológicas definidas pela equipe de engenharia de software. A título de ilustração, utilizam-se as atividades metodológicas genéricas discutidas anteriormente.[4] Cada uma dessas atividades representa um segmento do caminho espiral ilustrado na Figura 4.5. Assim que esse processo evolucionário começa, a equipe de software realiza atividades indicadas por um circuito em torno da espiral, no sentido horário, começando pelo seu centro. Os riscos (Capítulo 35) são levados em conta à medida que cada revolução é realizada. *Marcos de pontos-âncora* – uma combinação de artefatos e condições satisfeitas ao longo do trajeto da espiral – são indicados para cada passagem evolucionária.

Informações úteis sobre o modelo espiral podem ser obtidas em: **www.sei.cmu.edu/publications/documents/00.reports/00sr008.html**.

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

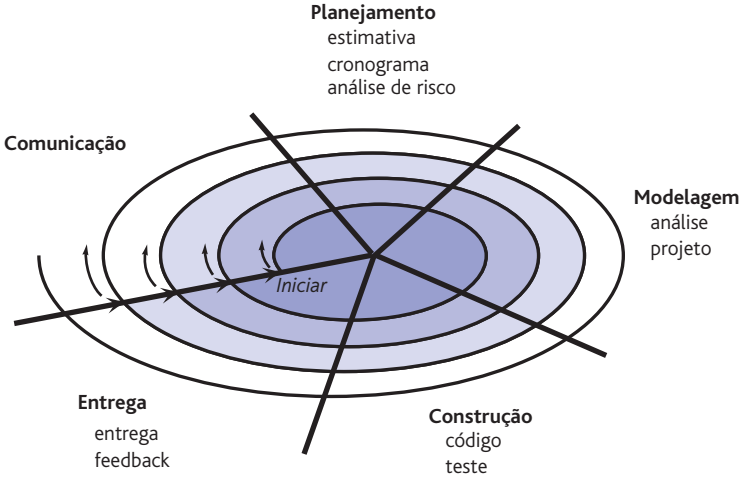


FIGURA 4.5 Modelo espiral típico.

[4] O modelo espiral discutido nesta seção é uma variação do modelo proposto por Boehm. Para mais informações sobre o modelo espiral original, consulte [Boe88]. Um material mais recente sobre o modelo espiral de Boehm pode ser encontrado em [Boe98].

Diferentemente 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. Logo, 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[5] até que o desenvolvimento de conceitos esteja concluído. Se o conceito for desenvolvido para ser um produto final, o processo prossegue na espiral pelas "bordas" e um "novo projeto de desenvolvimento de produto" se inicia. O novo produto evoluirá, passando por iterações em torno da espiral. Mais tarde, uma volta em torno da espiral pode ser usada para representar um "projeto de aperfeiçoamento do produto". Basicamente, a espiral, quando 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).

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

O modelo espiral é uma abordagem realista para o desenvolvimento de sistemas e de software em larga escala. Como o software evolui 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. Ele 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 exige consideração direta dos riscos técnicos em todos os estágios do projeto e, se aplicado apropriadamente, reduz os riscos antes de se tornarem problemáticos.

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

Dave Matthews Band

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, sem dúvida ocorrerão problemas.

4.1.4 Modelos concorrentes

O *modelo de desenvolvimento concorrente*, por vezes chamado de *engenharia concorrente*, possibilita à equipe de software representar elementos concorrentes e iterativos de qualquer um dos modelos de processo descritos neste capítulo. Por exemplo, a atividade de modelagem definida para o modelo espiral é realizada invocando uma ou mais destas ações de engenharia de software: prototipação, análise e projeto.[6]

Os planos de projeto devem ser considerados documentos vivos; o progresso deve ser avaliado frequentemente e revisado para levar em conta as alterações.

[5] As setas que apontam para dentro ao longo do eixo, separando a região de *disponibilização* da região de *comunicação*, indicam potencial para iteração local ao longo do mesmo trajeto da espiral.

[6] Deve-se notar que a análise e o projeto são tarefas complexas que exigem discussão substancial. A Parte II deste livro considera esses tópicos em detalhes.

CASASEGURA

Seleção de um modelo de processo, parte 2

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

Atores: 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 evolucionário.)

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, reformulamos e, então, entregamos outro incremento. Também se encaixa na natureza do produto. Podemos colocar alguma coisa no mercado rapidamente 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 manter os pés no chão. Acho que é melhor ir ajustando o planejamento à medida que formos aprendendo mais e as mudanças forem sendo solicitadas. É muito mais realista. Para que serve um plano se não para refletir a realidade?

Lee (franzindo a testa): Suponho que esteja certo, porém... a alta direção não vai gostar disso... querem um plano fixo.

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

A Figura 4.6 mostra um exemplo de abordagem de modelagem concorrente. Uma atividade – **modelagem** – poderia estar em qualquer um dos estados[7] observados em qualquer momento 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 simultaneamente, porém estão em diferentes estados.

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

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 **nenhum**, enquanto a comunicação inicial era concluída) agora faz uma transição para o estado **em desenvolvimento**. Entretanto, se o cliente indicar que devem ser feitas mudanças nos requisitos, a atividade de modelagem passa do estado **em desenvolvimento** para o estado **aguardando modificações**.

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

V. Daniel Hunt

A modelagem concorrente define uma série de eventos que vão disparar transições de um estado para outro para cada uma das atividades, ações ou tarefas da engenharia de software. Por exemplo, durante os estágios iniciais do projeto (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 vai 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

[7] Um *estado* é algum modo do comportamento observável externamente.

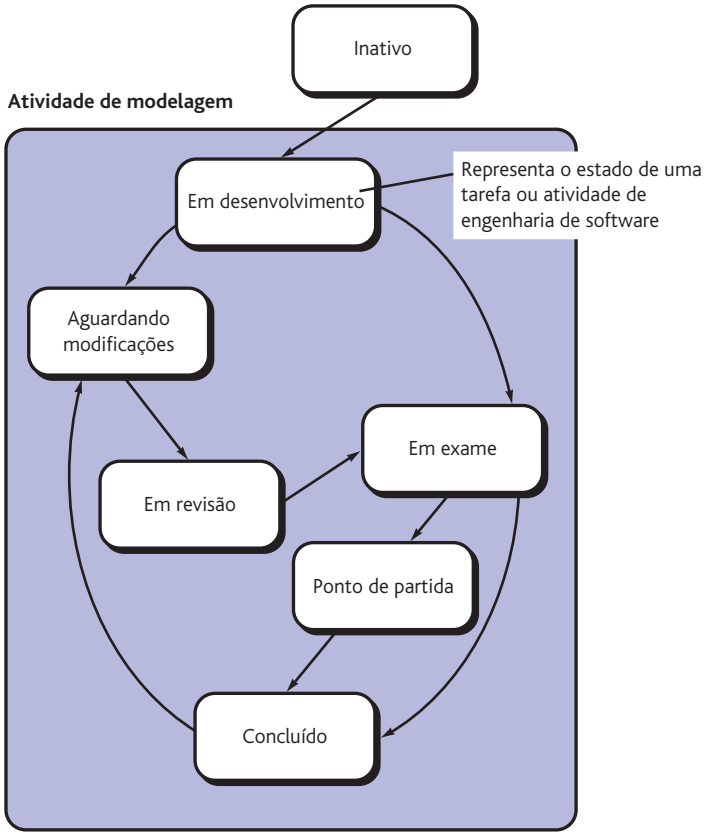


FIGURA 4.6 Um elemento do modelo de processo concorrente.

ou tarefas. Eventos gerados em um ponto da rede de processos disparam transições entre os estados associados a cada atividade.

4.1.5 Um comentário final sobre processos evolucionários

Conforme já mencionado, software moderno é caracterizado 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.[8]

Os modelos de processo evolucionário foram concebidos para lidar com essas questões, mas 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 das inquestionáveis vantagens dos processos de software evolucionários, temos algumas preocupações. A primeira delas é que a prototipação [e outros

[8] É importante notar, entretanto, que ser o primeiro a chegar ao mercado não é sinônimo de sucesso. Na verdade, 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).

> processos evolucionários mais sofisticados] traz um problema para o planejamento do projeto, devido ao número incerto, de ciclos necessários para construir o produto...
>
> A segunda é que os processos de software evolucionários não estabelecem a velocidade máxima da evolução. Se as evoluções ocorrerem em uma 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 pode ser afetada...
>
> A terceira é que os processos de software [evolucionário] devem se concentrar mais na flexibilidade e na extensibilidade do que na alta qualidade. Essa afirmação parece assustadora.

Quais são os possíveis pontos fracos dos modelos de processo evolucionário?

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

O objetivo dos modelos evolucionários é desenvolver software[9] de alta qualidade de modo iterativo ou incremental. Entretanto, é possível usar um processo evolucionário para enfatizar a flexibilidade, a extensibilidade e a velocidade de 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).

4.2 Modelos de processo especializado

Os modelos de processo especializado incluem muitas das características de um ou mais dos modelos tradicionais apresentados nas seções anteriores. Eles tendem a ser aplicados quando se opta por uma abordagem de engenharia de software especializada ou definida de forma restrita.[10]

4.2.1 Desenvolvimento baseado em componentes

Informações úteis sobre desenvolvimento baseado em componentes podem ser encontradas em: **www.cbd-hq.com**.

Componentes de software comercial de prateleira ou COTS (commercial off-the-shelf), desenvolvidos para serem oferecidos como produtos, disponibilizam a funcionalidade almejada com interfaces bem definidas que 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 por natureza [Nie92], demandando uma abordagem iterativa para a criação de software. O modelo de desenvolvi-

[9] Neste contexto, a qualidade de software é definida de forma bastante abrangente para englobar não apenas a satisfação dos clientes, mas também uma série de critérios técnicos, discutidos na Parte II deste livro.

[10] 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, eles realmente implicam um processo.

mento baseado em componentes compreende aplicações de componentes de software previamente empacotados.

As atividades de modelagem e construção começam com a identificação de componentes candidatos. Esses componentes podem ser projetados como módulos de software convencionais, como classes orientadas a objeto ou pacotes[11] de classes. Seja qual for a 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 à arquitetura.
5. Testes completos são realizados para garantir a funcionalidade adequada.

O modelo de desenvolvimento baseado em componentes leva à reutilização de software, e a capacidade de reutilização oferece aos engenheiros de software diversas vantagens mensuráveis, como a redução no tempo do ciclo de desenvolvimento e nos custos do projeto, caso a reutilização de componentes se torne parte da cultura de sua organização. O desenvolvimento baseado em componentes é discutido em mais detalhes no Capítulo 14.

4.2.2 O modelo de métodos formais

O *modelo de métodos formais* inclui um conjunto de atividades que conduzem à especificação matemática formal do software. Eles permitem especificar, desenvolver e verificar um sistema baseado em computador pela aplicação de uma notação matemática rigorosa. Uma variação dessa abordagem, chamada *engenharia de software sala limpa* (*cleanroom*) [Mil87, Dye92], é aplicada atualmente por algumas empresas de desenvolvimento de software.

O uso de métodos formais (Apêndice 3) durante o desenvolvimento oferece um mecanismo de eliminação de muitos dos problemas difíceis de serem superados com o uso 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, eles servem como base para verificar a programação e, portanto, possibilitam a descoberta e a correção de erros que, de outra forma, poderiam passar despercebidos.

Embora não seja uma das abordagens mais adotadas, o modelo de métodos formais oferece a promessa de software sem defeitos. Ainda assim, há

[11] Conceitos de orientação a objetos são discutidos no Apêndice 2 e usados ao longo da Parte II deste livro. Neste contexto, uma classe engloba um conjunto de dados e os procedimentos que os processam. Pacote de classes é um conjunto de classes relacionadas que operam juntas para alcançar algum resultado final.

motivos para preocupação quanto à sua aplicabilidade em um ambiente de negócios:

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

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

Apesar dessas preocupações, a abordagem dos 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 de voo para aeronaves e equipamentos médicos), bem como entre desenvolvedores que sofreriam pesadas sanções econômicas se ocorressem erros no software.

4.2.3 Desenvolvimento de software orientado a aspectos

Uma ampla variedade de recursos e informações sobre AOP pode ser encontrada em: **aosd.net**.

Para qualquer processo de software escolhido, os desenvolvedores de software complexo 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. Conforme os sistemas baseados em computadores se tornam mais sofisticados (e complexos), certas preocupações – propriedades exigidas pelo cliente ou áreas de interesse técnico – se estendem por toda a arquitetura. Algumas preocupações 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), enquanto outras são sistêmicas (por exemplo, sincronização de tarefas ou gerenciamento de memória).

A AOSD define "aspectos" que representam preocupações do cliente que transcendem várias funções, recursos e informações do sistema.

Quando as preocupações transcendem várias funções, recursos e informações do sistema, elas costumam ser chamadas de *preocupações transversais*. Os *requisitos de aspectos* definem as preocupações transversais que têm impacto em toda a arquitetura do software. O *desenvolvimento de software orientado a aspectos* (AOSD, *aspect-oriented software development*), também conhecido como *programação orientada a aspectos* (AOP, *aspect-oriented programming*) ou *engenharia de componentes orientada a aspectos* (AOCE, *aspect-oriented component engineering*), 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 preocupação cruzada" [Elr01].

Um processo orientado a aspectos distinto ainda não atingiu a maturidade. Entretanto, é provável que um processo desses adote características tanto dos modelos de processo evolucionário quanto de processo concorrente. O modelo evolucionário é apropriado quando os aspectos são identificados e, então, construídos. A natureza paralela do desenvolvimento concorrente é

FERRAMENTAS DO SOFTWARE

Gerenciamento de processos

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

Mecanismos: As ferramentas de gerenciamento de processo possibilitam que uma organização ou equipe de software defina um modelo de processo de software completo (atividades metodológicas, ações, tarefas, pontos de verificação para garantia da qualidade, marcos e artefatos). Além disso, tais ferramentas fornecem um guia, 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:[12]

GDPA, um conjunto de ferramentas para definição de processos de pesquisa, desenvolvido na Universidade de Bremen, na Alemanha (**www.informatik.unibremen.de/uniform/gdpa/home.htm**), fornece uma grande quantidade de funções de gerenciamento e modelagem de processos.

ALM Studio, desenvolvida pela Kovair Corporation (**http://www.kovair.com/**), engloba um conjunto de ferramentas para definição de processo, gerenciamento de requisitos, solução de problemas, planejamento e acompanhamento de projetos.

ProVision BPMx, desenvolvida pela OpenText (**http://bps.opentext.com/**), é representativa de muitas ferramentas que auxiliam na definição de processo e automação de fluxo de trabalho.

Uma lista valiosa de muitas ferramentas diferentes associadas ao processo de software pode ser encontrada em **www.computer.org/portal/web/swebok/html/ch10**.

essencial, porque os aspectos são criados de modo independente dos componentes de software localizados, apesar disso, os aspectos têm impacto direto sobre esses componentes. Portanto, é essencial estabelecer comunicação assíncrona entre as atividades de processos de software aplicadas à engenharia e à construção de aspectos e componentes.

Deixamos a discussão detalhada sobre desenvolvimento de software orientado a aspectos para livros dedicados ao assunto. Se tiver mais interesse, consulte [Ras11], [Saf08], [Cla05], [Fil05], [Jac04] e [Gra03].

4.3 O processo unificado

No livro que deu origem ao *Processo Unificado* (PU), 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":

> Hoje, o software tende em direção a sistemas maiores e mais complexos. Isso se deve, em parte, ao fato de que os computadores se tornam mais potentes a cada ano, aumentando a expectativa dos usuários em relação a eles. Essa tendência também tem sido influenciada pelo uso crescente da Internet para troca de todos os tipos de informação... Nosso apetite por software cada vez mais sofisticado aumenta à medida que tomamos conhecimento de como um produto pode ser aperfeiçoado de uma versão para a seguinte. Queremos software que seja cada

[12] A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

vez mais adaptado às nossas necessidades, mas isso, por sua vez, simplesmente torna o software mais complexo. Em suma, queremos cada vez mais.

Sob certos aspectos, 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 5). O Processo Unificado reconhece a importância da comunicação com o cliente e de métodos racionalizados para descrever a visão do cliente sobre um sistema (os casos de uso).[13] 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.

4.3.1 Um breve histórico

No 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 adotariam 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 da indústria para o desenvolvimento de software orientado a objetos.

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

4.3.2 Fases do processo unificado[14]

No Capítulo 3, discutimos cinco atividades metodológicas genéricas, expondo que elas poderiam ser usadas para descrever qualquer modelo de processo de software. O Processo Unificado não é exceção. A Figura 4.7 descreve as "fases" do PU e as relaciona com as atividades genéricas discutidas no Capítulo 1 e anteriormente neste capítulo.

---

[13] Um *caso de uso* (Capítulo 8) é uma narrativa textual ou modelo que descreve uma função ou recurso de um sistema do ponto de vista do usuário. Ele é escrito pelo usuário e serve como base para a criação de um modelo de análise mais amplo.

[14] O Processo Unificado é, algumas vezes, chamado de *Processo Unificado Racional* (RUP, *Rational Unified Process*) em homenagem à Rational Corporation (posteriormente adquirida pela IBM), uma das primeiras colaboradoras para o desenvolvimento e refinamento do PU e desenvolvedora de ambientes completos (ferramentas e tecnologia) que dão suporte ao processo.

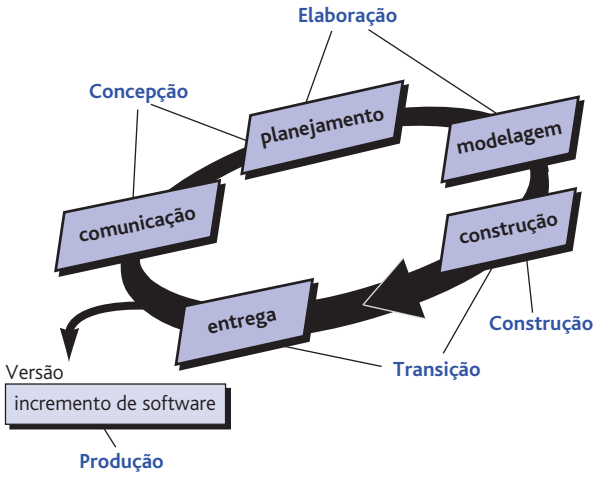


FIGURA 4.7 O Processo Unificado.

A *fase de concepção* do PU inclui a atividade de comunicação com o cliente e a de planejamento. A partir da colaboração com os envolvidos, identificam-se as necessidades de negócio para o software, propõe-se uma arquitetura rudimentar para o sistema e desenvolve-se um planejamento para a natureza iterativa e incremental do projeto decorrente. Requisitos de negócio fundamentais são descritos em um conjunto de casos de uso preliminares (Capítulo 8), 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 das funções e recursos que os compõem. Posteriormente, a arquitetura será refinada e expandida para um conjunto de modelos que representarão diferentes visões do sistema. O planejamento identifica recursos, avalia os principais riscos, define um cronograma e estabelece uma base para as fases que serão aplicadas à medida que o incremento de software for desenvolvido.

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

A *fase de elaboração* inclui as atividades de comunicação e de modelagem do modelo de processo genérico (Figura 4.7). A elaboração refina e expande os casos de uso preliminares, desenvolvidos como parte da fase de concepção, e amplia a representação arquitetural, incluindo cinco diferentes visões do software: modelo de caso de uso, modelo de análise, modelo de projeto, modelo de implementação e modelo de disponibilização. Em alguns casos, a elaboração gera uma "base de arquitetura executável" [Arl02], consistindo em um sistema executável "de degustação".[15] 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 adequados. Normalmente, as modificações no planejamento são feitas nessa oportunidade.

A *fase de construção* do PU é idêntica à atividade de construção definida para o processo de software genérico. Tendo como entrada o modelo de arqui-

Uma interessante abordagem sobre o PU, no contexto do desenvolvimento ágil, pode ser encontrada em **www.ambysoft.com/unifiedprocess/agileUP.html**.

[15] É importante observar que a base da arquitetura não é um protótipo, já que não é descartada. Ao contrário, a base ganha corpo durante a fase seguinte do PU.

tetura, a fase de construção desenvolve ou adquire componentes de software; esses componentes farão com que cada caso de uso se torne operacional para os usuários. Para tanto, os modelos de análise e de projeto, iniciados durante a fase de elaboração, são concluídos para refletir a versão final do incremento de software. Então, implementam-se, no código-fonte, todos os recursos e funções necessários e exigidos para o incremento de software (isto é, para a versão). À medida que os componentes são implementados, desenvolvem-se e executam-se testes[16] de unidades para cada um deles. Além disso, realizam-se atividades de integração (montagem de componentes e testes de integração). Os casos de uso são utilizados para se 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 feedback. Entrega-se o software aos usuários 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 solução de problemas, procedimentos de instalação) necessárias para o lançamento da versão. Na conclusão da fase de transição, o incremento torna-se uma versão utilizável do software.

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

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 concomitante e escalonada.

Um fluxo de trabalho de engenharia de software é distribuído ao longo de todas as fases do PU. No contexto do PU um *fluxo de trabalho* é análogo a um conjunto de tarefas (descrito no Capítulo 3). Ou seja, um fluxo de trabalho identifica as tarefas exigidas para realizar uma importante ação de engenharia de software e os artefatos produzidos como consequência da conclusão bem-sucedida das tarefas. 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) para ficar de acordo com suas necessidades.

4.4 Modelos de processo pessoal e de equipe

"Pessoas bem-sucedidas simplesmente desenvolveram o hábito de realizar coisas que as fracassadas não vão 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 somente será eficaz se for passível de grandes adaptações, a fim de atender às necessidades da equipe de projeto (aquela que está efetivamente realizando o trabalho de engenharia de software). Em

[16] Uma discussão abrangente sobre testes de software (inclusive *testes de unidades*) é apresentada nos Capítulos 22 a 26.

um cenário ideal, seria desenvolvido um processo que melhor se adequasse às suas necessidades e, simultaneamente, atendesse às necessidades mais amplas da equipe e da empresa. Outra opção é a equipe criar seu próprio processo para atender, ao mesmo tempo, às necessidades mais específicas dos indivíduos e às necessidades mais amplas da organização. Watts Humphrey ([Hum05] e [Hum00]) diz que é possível criar um "processo de software pessoal" e/ou um "processo de software de equipe". Ambos exigem trabalho árduo, treinamento e coordenação, mas são alcançáveis.[17]

4.4.1 Processo de Software Pessoal

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, realmente existe um "processo". Watts Humphrey [Hum05] sugere que, para modificar um processo pessoal ineficaz, um profissional deve passar por quatro fases, cada uma exigindo treinamento e orquestração cuidadosa. O *Processo de Software Pessoal* (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 permite controlar a qualidade de todos os artefatos de software desenvolvidos. O modelo PSP define cinco atividades estruturais:

Uma grande quantidade de recursos para PSP pode ser encontrada em **http://www.sei.cmu.edu/tsp/tools/academic/**.

Planejamento. Isola os requisitos e desenvolve as estimativas de tamanho e 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. Por último, identificam-se as tarefas de desenvolvimento e faz-se um cronograma para o projeto.

Projeto de alto nível. Especificações externas são desenvolvidas para cada componente a ser construído e um projeto de componentes é elaborado. Quando há incerteza, constroem-se protótipos. Todos os problemas são registrados e monitorados.

Quais atividades metodológicas são utilizadas durante o PSP?

Revisão de projeto de alto nível. Métodos de verificação formais (Apêndice 3) são aplicados para revelar erros no projeto. São mantidas métricas para resultados de trabalho e tarefas importantes.

Desenvolvimento. O projeto em nível de componentes é refinado e revisado. Um código é gerado, revisado, compilado e testado. São mantidas métricas para resultados de trabalho e tarefas importantes.

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.

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

[17] Vale notar que os defensores do desenvolvimento ágil de software (Capítulo 5) também afirmam que o processo deve ficar próximo à equipe. Eles propõem um método alternativo para conseguir isso.

No PSP é enfatizada a necessidade de identificar erros precocemente e, tão importante quanto, compreender os tipos de erro que provavelmente ocorrerão. Isso é obtido por meio de uma rigorosa atividade de avaliação em todos os artefatos de software gerados.

O PSP é 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 amplamente adotado 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, altos. O nível de medição exigido é culturalmente difícil para muitos profissionais da área de software.

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 o PSP introduz são importantes e vale a pena aprendê-los.

4.4.2 Processo de software de equipe

Informações sobre a formação de equipes com alto desempenho empregando TSP e PSP podem ser obtidas em **www.sei.cmu.edu/tsp/**.

Como muitos projetos de software para nível industrial são conduzidos por uma equipe de profissionais, Watts Humphrey estendeu as lições aprendidas com a introdução do PSP e propôs um *Processo de Software de 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 planejem 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[18] 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.

Para formar uma equipe autodirigida, é preciso haver boa colaboração interna e boa comunicação externa.

Uma equipe autodirigida tem um bom entendimento de suas metas e objetivos globais; define papéis e responsabilidades para cada um dos membros; monitora dados quantitativos do projeto (produtividade e qualidade); identi-

[18] O Modelo de Maturidade de Capacidade (CMM, Capability Maturity Model), uma medida da eficiência de um processo de software, é discutido no Capítulo 37.

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

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

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 processo 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.

O TSP reconhece que as melhores equipes de software são autodirigidas.[19] 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 sua abordagem à engenharia de software.

Assim como o PSP, o TSP é uma abordagem rigorosa 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 aplicada adequadamente.

4.5 Tecnologia de processos

Um ou mais dos modelos de processo discutidos nas seções anteriores devem ser adaptados para emprego por uma equipe de software. Para tanto, foram desenvolvidas *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.

As ferramentas de tecnologia de processos permitem que uma empresa de software construa um modelo automatizado da metodologia de processos, dos conjuntos de tarefas e das atividades de apoio, discutidos no Capítulo 3. O modelo, normalmente representado como uma rede, pode, então, ser analisado para determinar o fluxo de trabalho típico e examinar estruturas de processo alternativas que possam levar à redução de custos e tempo de desenvolvimento.

[19] No Capítulo 5, discutimos a importância das equipes "auto-organizadas" como um elemento-chave no desenvolvimento de software ágil.

FERRAMENTAS DO SOFTWARE

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 a fim de 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.

Mecanismos: As ferramentas nesta categoria permitem a uma equipe de desenvolvimento definir os elementos de um modelo único de processo (ações, tarefas, artefatos, 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:[20]

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

Adeptia BPM Server – projetada para gerenciar, automatizar e otimizar processos de negócio (**www.adeptia.com**)

ALM Studio Suite – conjunto de seis ferramentas com forte ênfase no gerenciamento das atividades de comunicação e modelagem (**http://www.kovair.com/**)

Uma vez criado um processo aceitável, outras ferramentas de tecnologia de processos 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 essas ferramentas para desenvolver uma lista de controle das tarefas a serem realizadas, dos artefatos de software a serem gerados e das atividades de garantia da qualidade a serem realizadas. A ferramenta de tecnologia de processos também pode ser usada para coordenar o uso de outras ferramentas de engenharia de software apropriadas para determinada tarefa.

4.6 Produto e processo

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:

> A cada 10 anos (com uma margem de erro de cinco anos), aproximadamente, a comunidade de software redefine "o problema", mudando seu foco de itens do produto para itens do 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 Maturidade de Capacitação de Desenvolvimento de Software (processo), do Software Engineering Institute [seguido por métodos orientados a objetos, seguido pelo desenvolvimento de software ágil.

[20] A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

Embora a tendência natural de um pêndulo seja a de repousar em um ponto intermediário entre dois extremos, o foco da comunidade de software muda constantemente, pois uma 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 de 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 destrinchar ou compreender o artefato completo, seu contexto, uso, significado e valor se o enxergarmos apenas como um processo ou como um produto.

Todas as atividades humanas podem ser um processo, mas todos se sentem 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 reutilização no desenvolvimento de software aumenta potencialmente a satisfação dos profissionais de software, 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 um 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 com suas pinceladas quanto com o resultado final de seu quadro. Um escritor sente prazer tanto com a procura da metáfora apropriada quanto com o livro finalizado. Como profissional de software criativo, você também deve extrair tanta satisfação do processo quanto 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.

4.7 Resumo

Os modelos de processo prescritivos são aplicados há anos, na tentativa de organizar e estruturar o desenvolvimento de software. Cada um desses modelos sugere um fluxo de processo um pouco diferente, mas todos realizam o

mesmo conjunto de atividades metodológicas genéricas: comunicação, planejamento, modelagem, construção e disponibilização.

Os modelos de processo sequenciais, como os modelos cascata e V, são os mais antigos paradigmas da engenharia de software. Eles sugerem um fluxo de processos linear que, muitas vezes, é inadequado para os sistemas modernos (por exemplo, alterações contínuas, sistemas em evolução, prazos apertados). Entretanto, eles têm 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 se adequar às mudanças. Esses modelos, como prototipação e o modelo espiral, produzem rapidamente artefatos de software incrementais (ou versões operacionais do software). Podem ser adotados para serem aplicados por todas as atividades de engenharia de software – desde o desenvolvimento de conceitos até a manutenção do sistema em longo prazo.

O modelo de processo concorrente permite 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 estimula uma abordagem matemática para o desenvolvimento e a verificação de software) e o modelo orientado a aspectos (que considera preocupações transversais que se estendem por toda a arquitetura do sistema). O Processo Unificado é um processo de software "dirigido a casos de uso, centrado na arquitetura, iterativo e incremental", desenvolvido como uma metodologia para os métodos e ferramentas da UML.

Foram propostos modelos pessoais e em equipe para o processo de software. Ambos enfatizam medida, planejamento e autodireção como ingredientes importantes para um processo de software bem-sucedido.

Problemas e pontos a ponderar

4.1. Dê três exemplos de projetos de software que seriam suscetíveis ao modelo cascata. Seja específico.

4.2. Dê três exemplos de projetos de software que seriam suscetíveis ao modelo de prototipação. Seja específico.

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

4.4. Dê três exemplos de projetos de software que seriam suscetíveis ao modelo incremental. Seja específico.

4.5. 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?

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

4.7. 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.

4.8. Quais são as vantagens e desvantagens de 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?

4.9. Dê três exemplos de projetos de software que seriam suscetíveis ao modelo baseado em componentes. Seja específico.

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

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

Leituras e fontes de informação complementares

A maioria dos livros sobre engenharia de software mencionados na seção *Leituras e fontes de informação complementares* do Capítulo 2 trata dos modelos de processo prescritivo com algum detalhe.

Cynkovic e Larsson (*Building Reliable Component-Based Systems,* Addison-Wesley, 2002) e Heineman e Council (*Component-Based Software Engineering,* Addison-Wesley, 2001) descrevem o processo exigido para implementar sistemas baseados em componentes. Jacobson e Ng (*Aspect-Oriented Software Development with Use Cases,* Addison-Wesley, 2005) e Filman e seus colegas (*Aspect-Oriented Software Development,* Addison-Wesley, 2004) discutem a natureza única do processo orientado a aspectos. Monin e Hinchey (*Understanding Formal Methods,* Springer, 2003) apresentam uma interessante introdução e Boca e seus colegas (*Formal Methods,* Springer, 2009) discutem o estado atual e novos rumos.

Livros de Kenett e Baker (*Software Process Quality: Management and Control,* Marcel Dekker, 1999) e Chrissis, Konrad e Shrum (*CMMI for Development: Guidelines for Process Integration and Product Improvement,* 3ª ed., Addison-Wesley, 2011) consideram como o gerenciamento da qualidade e o projeto de processos estão intimamente ligados.

Além do livro seminal de Jacobson, Rumbaugh e Booch sobre o Processo Unificado [Jac99], livros como os de Shuja e Krebs (*IBM Rational Unified Process Reference and Certification Guide,* IBM Press, 2008), 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,* IRM 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. Dennis, Wixom e Tegarden (*Systems Analysis and Design with UML,* 4ª ed., Wiley, 2012) aborda programação e modelagem de processo de negócio relacionado ao PU.

Uma ampla gama de fontes de informação sobre modelos de processo de software se encontra à disposição na Internet. Uma lista atualizada de referências relevantes (em inglês) para o processo para o software pode ser encontrada no site: **www.mhhe.com/pressman.**

5 Desenvolvimento ágil

Conceitos-chave

Em 2001, Kent Beck e outros 16 renomados desenvolvedores, autores e consultores da área de software [Bec01] (batizados de "Agile Alliance" – "Aliança dos Ágeis") assinaram o "Manifesto para o Desenvolvimento Ágil de Software" ("Manifesto for Agile Software Development"). Ele declarava:

> Ao desenvolver e ajudar outros a desenvolver software, desvendamos formas melhores de desenvolvimento. Por meio deste trabalho passamos a valorizar:
>
> *Indivíduos e interações* acima de processos e ferramentas
>
> *Software operacional* acima de documentação completa
>
> *Colaboração dos clientes* acima de negociação contratual
>
> *Respostas a mudanças* acima de seguir um plano
>
> Ou seja, embora haja valor nos itens à direita, valorizaremos os da esquerda mais ainda.

PANORAMA

O que é? A engenharia de software ágil combina filosofia com um conjunto de princípios de desenvolvimento. A filosofia defende a satisfação do cliente e a entrega incremental antecipada; equipes de projeto pequenas e altamente motivadas; métodos informais; artefatos de engenharia de software mínimos; e, acima de tudo, simplicidade no desenvolvimento geral. Os princípios de desenvolvimento priorizam a entrega mais do que a análise e o projeto (embora essas atividades não sejam desencorajadas); também priorizam a comunicação ativa e contínua entre desenvolvedores e clientes.

Quem realiza? Os engenheiros de software e outros envolvidos no projeto (gerentes, clientes, usuários) trabalham conjuntamente em uma equipe ágil – uma equipe que se auto-organiza e que controla seu próprio destino. Uma equipe ágil acelera a comunicação e a colaboração entre todos os participantes (que estão ao seu serviço).

Por que é importante? O ambiente moderno dos sistemas e dos produtos da área é acelerado e está em constante mudança. A engenharia de software ágil constitui uma alternativa razoável para a engenharia convencional voltada para certas classes de software e para certos tipos de projetos. Ela tem se mostrado capaz de entregar sistemas corretos rapidamente.

Quais são as etapas envolvidas? O desenvolvimento ágil poderia ser mais bem denominado "engenharia de software flexível". As atividades metodológicas básicas – comunicação, planejamento, modelagem, construção e entrega – permanecem. Entretanto, elas se transformam em um conjunto de tarefas mínimas que impulsiona a equipe para o desenvolvimento e para a entrega (pode-se levantar a questão de que isso é feito em detrimento da análise do problema e do projeto de soluções).

Qual é o artefato? Tanto o cliente quanto o engenheiro têm o mesmo parecer: o único artefato realmente importante consiste em um "incremento de software" operacional que seja entregue, adequadamente, na data combinada.

Como garantir que o trabalho foi realizado corretamente? Se a equipe ágil concorda que o processo funciona e essa equipe produz incrementos de software passíveis de entrega e que satisfaçam o cliente, então, o trabalho está correto.

Um manifesto normalmente é associado a um movimento político emergente: ataca a velha guarda e sugere uma mudança revolucionária (espera-se que para melhor). De certa forma, é exatamente disso que trata o desenvolvimento ágil.

Embora as ideias fundamentais que norteiam o desenvolvimento ágil tenham estado conosco por muitos anos, apenas há menos de duas décadas se consolidaram como um "movimento". Em essência, métodos ágeis[1] se desenvolveram em um esforço para sanar fraquezas reais e perceptíveis da engenharia de software convencional. O desenvolvimento ágil oferece benefícios importantes; no entanto, não é indicado para todos os projetos, produtos, pessoas e situações. Também *não* é a antítese da prática de engenharia de software confiável e pode ser aplicado como uma filosofia geral para todos os trabalhos de software.

Desenvolvimento ágil não significa que nenhum documento é criado; significa que apenas os documentos que vão ser consultados mais adiante no processo de desenvolvimento são criados.

Na economia moderna, frequentemente é difícil ou impossível prever como um sistema computacional (por exemplo, um aplicativo móvel) vai evoluir com o tempo. As condições de mercado mudam rapidamente, as necessidades dos usuários se alteram, e novas ameaças competitivas surgem sem aviso. Em muitas situações, não se conseguirá definir os requisitos completamente antes que se inicie o projeto. É preciso ser ágil o suficiente para dar uma resposta a um ambiente de negócios fluido.

Fluidez implica mudança, e mudança é cara – particularmente se for sem controle e mal gerenciada. Uma das características mais convincentes da metodologia ágil é sua habilidade de reduzir os custos da mudança no processo de software.

Será que isso significa que o reconhecimento dos desafios apresentados pela realidade moderna faz que valiosos princípios, conceitos, métodos e ferramentas da engenharia de software sejam descartados? Absolutamente não! Como todas as disciplinas de engenharia, a engenharia de software continua a evoluir. Ela pode ser adaptada facilmente aos desafios apresentados pela demanda por agilidade.

"Agilidade: 1. Todo o resto: 0."

Tom DeMarco

Em um texto instigante sobre desenvolvimento de software ágil, Alistair Cockburn [Coc02] argumenta que o modelo de processo prescritivo, apresentado no Capítulo 4, tem uma falha essencial: *esquece-se das fraquezas das pessoas que desenvolvem o software*. Os engenheiros de software não são robôs. Eles apresentam grande variação nos estilos de trabalho; diferenças significativas no nível de habilidade, criatividade, organização, consistência e espontaneidade. Alguns se comunicam bem na forma escrita, outros não. Cockburn afirma que os modelos de processos podem "lidar com as fraquezas comuns das pessoas com disciplina e/ou tolerância" e que a maioria dos modelos de processos prescritivos opta por disciplina. Segundo ele: "Como a coerência nas ações é uma fraqueza humana, as metodologias com disciplina elevada são frágeis".

Para que funcionem, os modelos de processos devem fornecer um mecanismo realista que estimule a disciplina necessária ou, então, devem ter características que apresentem "tolerância" com as pessoas que realizam trabalhos de engenharia de software. Invariavelmente, práticas tolerantes são mais

[1] Os métodos ágeis são, algumas vezes, conhecidos como *métodos light* ou *métodos enxutos* (*lean methods*).

facilmente adotadas e sustentadas pelas pessoas envolvidas, porém (como o próprio Cockburn admite) podem ser menos produtivas. Como a maioria das coisas na vida, deve-se considerar os prós e os contras.

5.1 O que é agilidade?

Afinal, o que é agilidade no contexto da engenharia de software? Ivar Jacobson [Jac02a] apresenta uma discussão útil:

> Atualmente, *agilidade* se tornou a palavra da moda quando se descreve um processo de software moderno. Todo mundo é ágil. Uma equipe ágil é aquela rápida e capaz de responder de modo adequado às mudanças. Mudança tem tudo a ver com desenvolvimento de software. Mudança no software que está sendo criado, mudança nos membros da equipe, mudança devido a novas tecnologias, mudanças de todos os tipos que poderão ter um impacto no produto que está em construção ou no projeto que cria o produto. Suporte à mudança deve ser incorporado a tudo o que fazemos em software, algo que abraçamos porque é o coração e a alma do software. Uma equipe ágil reconhece que o software é desenvolvido por indivíduos trabalhando em equipes e que as habilidades dessas pessoas, suas capacidades em colaborar estão no cerne do sucesso do projeto.

De acordo com Jacobson, a difusão da mudança é o principal condutor para a agilidade. Os engenheiros de software devem ser rápidos, caso queiram assimilar as rápidas mudanças que Jacobson descreve.

Não cometa o erro de supor que a agilidade lhe dará licença para abreviar soluções. Processo é um requisito, e disciplina é essencial.

Entretanto, agilidade é mais do que uma resposta à mudança. Ela abrange também a filosofia proposta no manifesto citado no início deste capítulo. Ela incentiva a estruturação e as atitudes em equipe que tornam a comunicação mais fácil (entre membros da equipe, entre o pessoal ligado à tecnologia e o pessoal da área comercial, entre os engenheiros de software e seus gerentes). Enfatiza a entrega rápida do software operacional e diminui a importância dos artefatos intermediários (nem sempre um bom negócio); aceita o cliente como parte da equipe de desenvolvimento e trabalha para eliminar a atitude de "nós e eles" que continua a impregnar muitos projetos de software; reconhece que o planejamento em um mundo incerto tem seus limites e que o plano (roteiro) de projeto deve ser flexível.

A agilidade pode ser aplicada a qualquer processo de software. Entretanto, para alcançá-la, é essencial que o processo seja projetado de modo que a equipe possa adaptar e alinhar (racionalizar) tarefas; possa conduzir o planejamento, compreendendo a fluidez de uma metodologia de desenvolvimento ágil; possa eliminar tudo, exceto os artefatos essenciais, conservando-os enxutos; e enfatize a estratégia de entrega incremental, conseguindo entregar ao cliente, o mais rapidamente possível, o software operacional para o tipo de produto e ambiente operacional.

5.2 Agilidade e o custo das mudanças

O pensamento convencional em desenvolvimento de software (baseada em décadas de experiência) é que os custos de mudanças aumentam de forma não linear conforme o projeto avança (Figura 5.1, curva em preto contínua). É relativamente

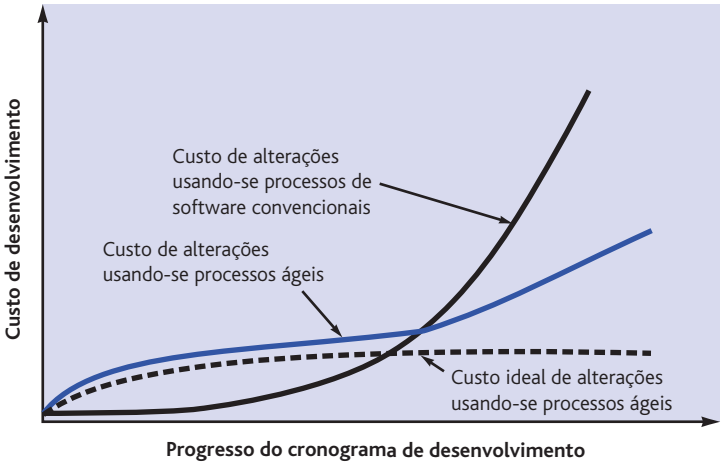


FIGURA 5.1 Custos de alterações como uma função do tempo em desenvolvimento.

fácil acomodar uma mudança quando a equipe de software está reunindo requisitos (no início de um projeto). Talvez seja necessário alterar um detalhamento do uso, ampliar uma lista de funções ou editar uma especificação por escrito. Os custos desse trabalho são mínimos, e o tempo demandado não afetará negativamente o resultado do projeto. Mas, se adiantarmos alguns meses, o que aconteceria? A equipe está em meio aos testes de validação (que ocorrem relativamente no final do projeto), e um importante envolvido está solicitando uma mudança funcional grande. A mudança exige uma alteração no projeto da arquitetura do software, projeto e desenvolvimento de três novos componentes, modificações em outros cinco componentes, projeto de novos testes e assim por diante. Os custos crescem rapidamente, e o tempo e os custos necessários para assegurar que a mudança seja feita sem efeitos colaterais inesperados não serão insignificantes.

"A agilidade é dinâmica, de conteúdo específico, abrange mudanças agressivas e é orientada ao crescimento."

Steven Goldman *et al.*

Os proponentes da agilidade (por exemplo, [Bec00], [Amb04]) argumentam que um processo ágil bem elaborado "achata" o custo da curva de mudança (Figura 5.1, curva em linha azul), permitindo que uma equipe de software assimile as alterações, realizadas posteriormente em um projeto de software, sem um impacto significativo nos custos ou no tempo. Já foi mencionado que o processo ágil envolve entregas incrementais. O custo das mudanças é atenuado quando a entrega incremental é associada a outras práticas ágeis, como testes contínuos de unidades e programação em pares (discutida mais adiante neste capítulo). Há evidências [CocO1a] que sugerem ser possível alcançar redução significativa nos custos de alterações, embora haja um debate contínuo sobre qual o nível em que a curva de custos se torna "achatada".

Um processo ágil reduz o custo das alterações porque o software é entregue (liberado) de forma incremental e as alterações podem ser mais bem controladas dentro de incrementais.

5.3 O que é processo ágil?

Qualquer processo ágil de software é caracterizado de uma forma que trate de uma série de preceitos-chave [Fow02] acerca da maioria dos projetos de software:

1. É difícil prever quais requisitos de software vão persistir e quais sofrerão alterações. É igualmente difícil prever de que maneira as prioridades do cliente sofrerão alterações conforme o projeto avança.

Uma vasta coleção de artigos sobre processo ágil pode ser encontrada em **http://www.agilemodeling.com/**.

2. Para muitos tipos de software, o projeto e a construção são intercalados. Ou seja, ambas as atividades devem ser realizadas em conjunto para que os modelos de projeto sejam provados conforme são criados. É difícil prever quanto de trabalho de projeto será necessário antes que a sua construção (desenvolvimento) seja implementada para avaliar o projeto.
3. Análise, projeto, construção (desenvolvimento) e testes não são tão previsíveis (do ponto de vista de planejamento) quanto gostaríamos que fosse.

Dados esses três preceitos, surge uma importante questão: como criar um processo capaz de administrar a *imprevisibilidade*? A resposta, conforme já observado, está na adaptabilidade do processo (alterar rapidamente o projeto e as condições técnicas). Portanto, um processo ágil deve ser *adaptável*.

Adaptação contínua sem progressos, entretanto, de pouco adianta. Um processo ágil de software deve adaptar *de modo incremental*. Para conseguir uma adaptação incremental, a equipe ágil precisa de feedback do cliente (de modo que as adaptações apropriadas possam ser feitas). Um catalisador eficaz para o feedback do cliente é um protótipo operacional ou parte de um sistema operacional. Dessa forma, deve-se instituir uma *estratégia de desenvolvimento incremental*. Os *incrementos de software* (protótipos executáveis ou partes de um sistema operacional) devem ser entregues em curtos períodos de tempo, de modo que as adaptações acompanhem o mesmo ritmo das mudanças (imprevisibilidade). Essa abordagem iterativa capacita o cliente a avaliar o incremento de software regularmente, fornecer o feedback necessário para a equipe de software e influenciar as adaptações feitas no processo para incluir o feedback adequadamente.

5.3.1 Princípios da agilidade

Embora processos ágeis considerem as alterações, examinar as razões para tais mudanças ainda continua sendo importante.

A Agile Alliance (consulte [Agi03], [Fow01]) estabelece 12 princípios para alcançar a agilidade:

1. A maior prioridade é satisfazer o cliente com entrega adiantada e contínua de software funcionando.
2. Aceite bem os pedidos de alterações, mesmo com o desenvolvimento adiantado. Os processos ágeis se aproveitam das mudanças para a vantagem competitiva do cliente.
3. Entregue software em funcionamento frequentemente, de algumas semanas a alguns meses, dando preferência a intervalos mais curtos.
4. O pessoal do comercial e os desenvolvedores devem trabalhar em conjunto diariamente ao longo de todo o projeto.
5. Construa projetos em torno de pessoas motivadas. Dê a elas o ambiente e o apoio necessários e acredite que elas farão o trabalho corretamente.
6. O método mais eficiente e efetivo de transmitir informações para e dentro de uma equipe de desenvolvimento é uma conversa aberta, presencial.
7. Software em funcionamento é a principal medida de progresso.

Software ativo é importante, mas não se deve esquecer que também deve apresentar uma série de atributos de qualidade, incluindo confiabilidade, usabilidade e facilidade de manutenção.

8. Os processos ágeis promovem desenvolvimento sustentável. Proponentes, desenvolvedores e usuários devem estar aptos a manter um ritmo constante indefinidamente.
9. Atenção contínua para com a excelência técnica e para com bons projetos aumenta a agilidade.
10. Simplicidade – a arte de maximizar o volume de trabalho não realizado – é essencial.
11. As melhores arquiteturas, requisitos e projetos surgem de equipes auto-organizadas.
12. Em intervalos regulares, a equipe se avalia para ver como pode se tornar mais eficiente, então, sintoniza e ajusta seu comportamento de acordo.

Nem todo modelo de processo ágil aplica esses 12 princípios atribuindo-lhes pesos iguais, e alguns modelos preferem ignorar (ou pelo menos subestimam) a importância de um ou mais desses princípios. Entretanto, os princípios definem um *espírito ágil* mantido em cada um dos modelos de processo apresentados neste capítulo.

5.3.2 A política do desenvolvimento ágil

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

Tem havido debates consideráveis (algumas vezes acirrados) sobre os benefícios e a aplicabilidade do desenvolvimento de software ágil, em contraposição aos processos de engenharia de software mais convencionais. Jim Highsmith [Hig02a] (em tom jocoso) estabelece extremos ao caracterizar o sentimento do grupo pró-agilidade ("os agilistas"). "Os metodologistas tradicionais são um bando de 'pés na lama' que preferem produzir documentação sem falhas em vez de um sistema que funcione e atenda às necessidades do negócio". Em um contraponto, ele apresenta (mais uma vez em tom jocoso) a posição do grupo da engenharia de software tradicional: "Os metodologistas de pouco peso, quer dizer, os metodologistas 'ágeis' são um bando de hackers pretensiosos que vão acabar tendo uma grande surpresa ao tentarem transformar seus brinquedinhos em software de porte empresarial".

Como todo argumento sobre tecnologia de software, o debate sobre metodologia corre o risco de descambar para uma guerra santa. Se for deflagrada uma guerra, a racionalidade desaparecerá, e crenças, em vez de fatos, orientarão a tomada de decisão.

Ninguém é contra a agilidade. A verdadeira questão é: qual a melhor maneira de atingi-la? Igualmente importante é: como desenvolver software que atenda às necessidades atuais dos clientes e que apresente características de qualidade que o permitam ser estendido e ampliado para responder às necessidades dos clientes no longo prazo?

Não há respostas absolutas para nenhuma dessas perguntas. Mesmo na própria escola ágil, existem vários modelos de processos propostos (Seção 5.4), cada um com uma abordagem sutilmente diferente a respeito do problema da agilidade. Em cada modelo existe um conjunto de "ideias" (os agilistas relutam em chamá-las "tarefas de trabalho") que representam um afastamento significativo da engenharia de software tradicional. E, ainda assim, muitos conceitos

ágeis são apenas adaptações de bons conceitos da engenharia de software. Conclusão: pode-se ganhar muito considerando o que há de melhor nas duas escolas e praticamente nada denegrindo uma ou outra abordagem.

Caso se interesse mais, consulte [Hig01], [Hig02a] e [DeM02], em que é apresentado um resumo interessante a respeito de outras importantes questões técnicas e políticas.

5.4 Extreme programming – XP (Programação Extrema)

Um premiado "jogo de simulação de processos", que inclui um módulo de processo XP, pode ser encontrado em: **http://www.ics.uci.edu/~emilyo/SimSE/downloads.html**.

Para ilustrar um processo ágil de forma um pouco mais detalhada, vamos dar uma visão geral da *Extreme Programming – XP* (*Programação Extrema*), a abordagem mais amplamente utilizada para desenvolvimento de software ágil. Embora os primeiros trabalhos sobre os conceitos e métodos associados à XP tenham ocorrido no final dos anos 1980, o trabalho seminal sobre o tema foi escrito por Kent Beck [Bec04a]. Uma variante da XP, denominada *Industrial XP* (IXP), refina a XP para aplicar processo ágil especificamente em grandes organizações [Ker05].

5.4.1 O processo XP

O que é uma "história" XP?

A Extreme Programming (Programação Extrema) emprega uma metodologia orientada a objetos (Apêndice 2) como seu paradigma de desenvolvimento e envolve um conjunto de regras e práticas constantes no contexto de quatro atividades metodológicas: planejamento, projeto, codificação e testes. A Figura 5.2 ilustra o processo XP e destaca alguns conceitos e tarefas-chave associados a cada uma das atividades metodológicas. As atividades-chave da XP são sintetizadas nos parágrafos a seguir.

Planejamento. A atividade de planejamento (também chamada de *o jogo do planejamento*) se inicia com *ouvir* – uma atividade de levantamento de requisi-

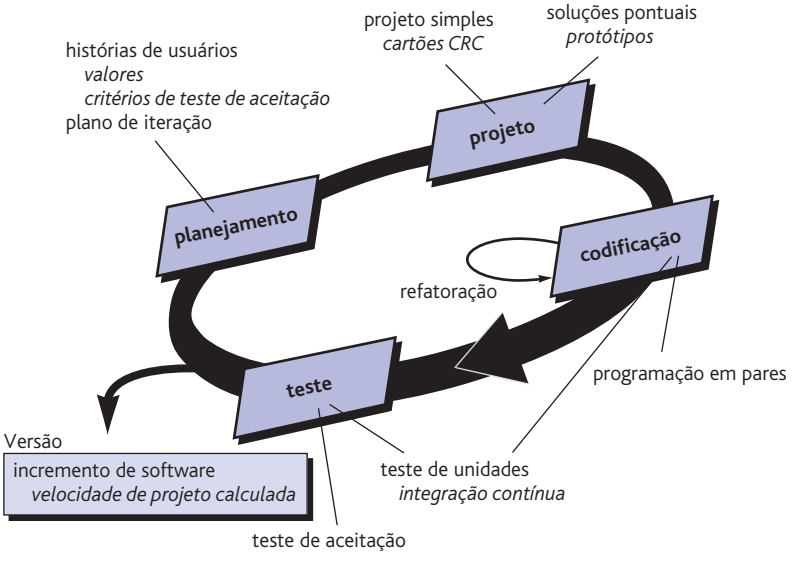


FIGURA 5.2 O processo da Extreme Programming (XP).

tos que capacita os membros técnicos da equipe XP a entender o ambiente de negócios do software e permite obter uma percepção ampla sobre os resultados solicitados, fatores principais e funcionalidade. A atividade de ouvir conduz à criação de um conjunto de "histórias" (também denominadas *histórias de usuários*) que descreve o resultado, as características e a funcionalidade solicitados para o software a ser construído. Cada *história* (similar aos casos de uso descritos no Capítulo 8) é escrita pelo cliente e é colocada em uma ficha. O cliente atribui um *valor* (uma prioridade) à história baseando-se no valor de negócio global do recurso ou função.[2] Os membros da equipe XP avaliam, então, cada história e atribuem um *custo* – medido em semanas de desenvolvimento – a ela. Se a história exigir, por estimativa, mais do que três semanas de desenvolvimento, é solicitado ao cliente que ele a divida em histórias menores, e a atribuição de valor e custo ocorre novamente. É importante notar que podem ser escritas novas histórias a qualquer momento.

Um "jogo de planejamento" XP bastante interessante pode ser encontrado em: **http://csis.pace.edu/~bergin/xp/planninggame.html**.

Clientes e desenvolvedores trabalham juntos para decidir como agrupar histórias para a versão seguinte (o próximo incremento de software) a ser desenvolvida pela equipe XP. Conseguindo chegar a um *compromisso* básico (concordância sobre quais histórias serão incluídas, data de entrega e outras questões de projeto) para uma versão, a equipe XP ordena as histórias a ser desenvolvidas em uma de três formas: (1) todas serão implementadas imediatamente (em um prazo de poucas semanas), (2) as histórias de maior valor serão deslocadas para cima no cronograma e implementadas primeiro ou (3) as histórias de maior risco serão deslocadas para cima no cronograma e implementadas primeiro.

A velocidade do projeto é uma medida sutil da produtividade de uma equipe.

Depois de a primeira versão do projeto (também denominada incremento de software) ter sido entregue, a equipe XP calcula a velocidade do projeto. De forma simples, a *velocidade do projeto* é o número de histórias de clientes implementadas durante a primeira versão. Assim, a velocidade do projeto pode ser utilizada para (1) ajudar a estimar as datas de entrega e o cronograma para versões subsequentes e (2) determinar se foi assumido um compromisso exagerado para todas as histórias ao longo de todo o projeto de desenvolvimento. Se ocorrer um exagero, o conteúdo das versões é modificado – ou as datas finais de entrega são alteradas.

A XP tira a ênfase da importância do projeto. Nem todos concordam. De fato, há ocasiões em que o projeto deve ser enfatizado.

Conforme o trabalho de desenvolvimento prossegue, o cliente pode acrescentar histórias, mudar o valor de uma já existente, dividir algumas ou eliminá-las. Em seguida, a equipe XP reconsidera todas as versões remanescentes e modifica seus planos de forma correspondente.

Projeto. O projeto XP segue rigorosamente o princípio KISS (*keep it simple, stupid!*, ou seja, não complique!). É sempre preferível um projeto simples a uma representação mais complexa. Como acréscimo, o projeto oferece um guia de implementação para uma história à medida que é escrita – nada mais, nada menos. O projeto de funcionalidade extra (pelo fato de o desenvolvedor supor que ela será necessária no futuro) é desestimulado.[3]

[2] O valor de uma história também pode depender da presença de outra história.

[3] Tais diretrizes de projeto devem ser seguidas em todos os métodos de engenharia de software, apesar de ocorrerem situações em que terminologia e notação sofisticadas possam constituir obstáculo para a simplicidade.

Técnicas de refatoração e ferramentas podem ser encontradas em: **www.refactoring.com.**

A XP estimula o uso de cartões CRC (Capítulo 10) como um mecanismo eficaz para pensar o software em um contexto orientado a objetos. Os cartões CRC (classe-responsabilidade-colaborador) identificam e organizam as classes orientadas a objetos[4] relevantes para o incremento de software corrente. A equipe XP conduz o exercício de projeto usando um processo semelhante ao descrito no Capítulo 10. Os cartões CRC são o único artefato de projeto produzido como parte do processo XP.

A refatoração aprimora a estrutura interna de um projeto (ou código-fonte) sem alterar sua funcionalidade ou comportamento externos.

Se for encontrado um problema de projeto difícil, como parte do projeto de uma história, a XP recomenda a criação imediata de um protótipo operacional dessa parte do projeto. Denominada *solução pontual*, o protótipo do projeto é implementado e avaliado. O objetivo é reduzir o risco para quando a verdadeira implementação iniciar e validar as estimativas originais para a história contendo o problema de projeto.

A XP estimula a *refatoração* – uma técnica de construção que também é uma técnica de projeto. Fowler [Fow00] descreve a refatoração da seguinte maneira:

> Refatoração é o processo de alterar um sistema de software de modo que o comportamento externo do código não se altere, mas a estrutura interna se aprimore. É uma forma disciplinada de organizar código [e modificar/simplificar o projeto interno] que minimiza as chances de introdução de bugs. Em resumo, ao se refatorar, se está aperfeiçoando o projeto de codificação depois de este ter sido feito.

Como o projeto XP praticamente não usa notação e produz poucos artefatos, quando produz, além dos cartões CRC e soluções pontuais, o projeto é visto como algo transitório que pode e deve ser continuamente modificado conforme a construção prossegue. O objetivo da refatoração é controlar essas modificações, sugerindo pequenas mudanças de projeto "capazes de melhorá-lo radicalmente" [Fow00]. Deve-se observar, no entanto, que o esforço necessário para a refatoração pode aumentar significativamente à medida que o tamanho de uma aplicação cresça.

Um aspecto central na XP é o de que a elaboração do projeto ocorre tanto antes *quanto depois* de se ter iniciado a codificação. Refatoração significa que o "projetar" é realizado continuamente enquanto o sistema estiver em elaboração. Na realidade, a própria atividade de desenvolvimento guiará a equipe XP quanto ao aprimoramento do projeto.

Informações úteis sobre a XP podem ser obtidas em **www.xprogramming.com**.

Codificação. Depois de desenvolvidas as histórias, e de o trabalho preliminar de elaboração do projeto ter sido feito, a equipe *não* passa para a codificação, mas sim desenvolve uma série de testes de unidades que exercitarão cada uma das histórias a ser incluída na versão corrente (incremento de software).[5]

[4] As classes orientadas a objetos são discutidas no Apêndice 2, no Capítulo 10 e ao longo da Parte II deste livro.

[5] Essa abordagem equivale a saber as perguntas de uma prova antes de começar a estudar. Torna o estudo muito mais fácil, permitindo que se concentre a atenção apenas nas perguntas que serão feitas.

Uma vez criado o teste de unidades[6], o desenvolvedor poderá se concentrar melhor no que deve ser implementado para ser aprovado no teste. Nada estranho é adicionado (KISS). Estando o código completo, ele pode ser testado em unidade imediatamente e, dessa forma, fornecer feedback para os desenvolvedores instantaneamente.

O que é programação em pares?

Um conceito-chave na atividade de codificação (e um dos mais discutidos aspectos da XP) é a *programação em pares*. A XP recomenda que duas pessoas trabalhem juntas em uma mesma estação de trabalho para criar código para uma história. Isso fornece um mecanismo para solução de problemas em tempo real (duas cabeças normalmente funcionam melhor do que uma) e garantia da qualidade em tempo real (o código é revisto à medida que é criado). Ele também mantém os desenvolvedores concentrados no problema em questão. Na prática, cada pessoa assume um papel ligeiramente diferente. Por exemplo, uma pessoa poderia pensar nos detalhes da codificação de determinada parte do projeto, enquanto outra assegura que padrões de codificação (uma parte exigida pela XP) sejam seguidos ou que o código para a história passará no teste de unidades desenvolvido para validação do código em relação à história.[7]

Muitas equipes de software são constituídas por individualistas. É preciso mudar tal cultura para que a programação em pares funcione efetivamente.

À medida que a dupla de programadores conclui o trabalho, o código que desenvolveram é integrado ao trabalho de outros. Em alguns casos, isso é realizado diariamente por uma equipe de integração. Em outros, a dupla de programadores é responsável pela integração. A estratégia de "integração contínua" ajuda a evitar problemas de compatibilidade e de interface, além de criar um ambiente "teste da fumaça" (Capítulo 22) que ajuda a revelar erros precocemente.

Como são usados os testes de unidade na XP?

Testes. Os testes de unidades criados devem ser implementados usando-se uma metodologia que os capacite a ser automatizados (assim, poderão ser executados fácil e repetidamente). Isso estimula uma estratégia de testes de regressão (Capítulo 22) toda vez que o código for modificado (o que é frequente, dada a filosofia de refatoração da XP).

Como os testes de unidades individuais são organizados em um "conjunto de testes universal" [Wel99], os testes de integração e validação do sistema podem ocorrer diariamente. Isso dá à equipe XP uma indicação contínua do progresso e também permite lançar alertas logo no início, caso as coisas não andem bem. Wells [Wel99] afirma: "Corrigir pequenos problemas em intervalos de poucas horas leva menos tempo do que corrigir problemas enormes próximo ao prazo de entrega".

Os testes de aceitação da XP são elaborados com base nas histórias de usuários.

Os *testes de aceitação* da XP, também denominados *testes de cliente,* são especificados pelo cliente e mantêm o foco nas características e na funcionalidade do sistema total que são visíveis e que podem ser revistas pelo cliente. Os testes de aceitação são obtidos de histórias de usuários implementadas como parte de uma versão do software.

---

[6] O teste de unidades, discutido detalhadamente no Capítulo 22, concentra-se em um componente de software individual, exercitando a interface, a estrutura de dados e a funcionalidade do componente, em uma tentativa de que se revelem erros pertinentes ao componente.

[7] A programação em pares se tornou tão difundida em toda a comunidade do software, que o tema virou manchete no *The Wall Street Journal* [Wal12].

5.4.2 Industrial XP

Que novas práticas são acrescidas à XP para elaborar a IXP?

Joshua Kerievsky [Ker05] descreve a *Industrial Extreme Programming* (IXP, Programação Extrema Industrial) da seguinte maneira: "A IXP é uma evolução orgânica da XP. Ela é imbuída do mesmo espírito minimalista, centrado no cliente e orientado a testes da XP. Difere da XP original principalmente por sua maior inclusão do gerenciamento, por seu papel expandido para os clientes e por suas práticas técnicas atualizadas". A IXP incorpora seis novas práticas desenvolvidas para ajudar a garantir que um projeto XP funcione com êxito em empreendimentos significativos em uma grande organização:

"Habilidade consiste no que se é capaz de fazer. Motivação determina o que você faz. Atitude determina quão bem você faz."

Lou Holtz

Avaliação imediata. A equipe IXP verifica se todos os membros da comunidade de projeto (por exemplo, envolvidos, desenvolvedores, gerentes) estão a bordo, têm o ambiente correto estabelecido e entendem os níveis de habilidade envolvidos.

Comunidade de projeto. A equipe IXP determina se as pessoas certas, com as habilidades e o treinamento corretos, estão prontas para o projeto. A "comunidade" abrange tecnólogos e outros envolvidos.

Mapeamento do projeto. A própria equipe IXP avalia o projeto para determinar se ele se justifica em termos de negócios e se vai ultrapassar as metas e objetivos globais da organização.

Gerenciamento orientado a testes. A equipe IPX estabelece uma série de "destinos" mensuráveis [Ker05] que avaliam o progresso até a data e, então, define mecanismos para determinar se estes foram atingidos ou não.

Retrospectivas. Uma equipe IXP conduz uma revisão técnica especializada (Capítulo 20) após a entrega de um incremento de software. Denominada *retrospectiva,* a revisão examina "problemas, eventos e lições aprendidas" [Ker05] ao longo do processo de incremento de software e/ou do desenvolvimento da versão completa do software.

Aprendizagem contínua. A equipe IXP é estimulada (e possivelmente incentivada) a aprender novos métodos e técnicas que possam levar a um produto de qualidade mais alta.

CASASEGURA

Considerando o desenvolvimento de software ágil

Cena: Escritório de Doug Miller.

Atores: Doug Miller, gerente de engenharia de software; Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software.

Conversa:
(Batendo à porta, Jamie e Vinod entram na sala de Doug.)

Jamie: Doug, você tem um minuto?

Doug: Com certeza, Jamie, o que há?

Jamie: Estivemos pensando a respeito da discussão de ontem sobre processos... sabe, que processo vamos escolher para o *CasaSegura*.

Doug: E?

Vinod: Eu estava conversando com um amigo de outra empresa e ele me falou sobre Extreme Programming. É um modelo de processo ágil... já ouviu falar?

Doug: Sim, algumas coisas boas, outras ruins.

Jamie: Bem, pareceu muito bom para nós. Permite que se desenvolva software rapidamente, usa algo chamado programação em pares para fazer verificações de qualidade em tempo real... é bem legal, eu acho.

Doug: Realmente, apresenta um monte de ideias muito boas. Gosto do conceito de programação em pares, por exemplo, e da ideia de que os envolvidos devam fazer parte da equipe.

Jamie: Hã? Quer dizer que o pessoal de marketing trabalhará conosco na equipe de projeto?

Doug (confirmando com a cabeça): Eles estão envolvidos, não?

Jamie: Jesus... eles vão solicitar alterações a cada cinco minutos.

Vinod: Não necessariamente. Meu amigo me disse que existem formas de se "abarcar" as mudanças durante um projeto XP.

Doug: Então, meus amigos, vocês acham que deveríamos usar a XP?

Jamie: Definitivamente vale considerar.

Doug: Concordo. E mesmo que optássemos por um modelo incremental, não há razão para não podermos incorporar muito do que a XP tem a oferecer.

Vinod: Doug, mas antes você disse "algumas coisas boas, outras ruins". Quais são as coisas ruins?

Doug: O que não me agrada é a maneira como a XP dá menos importância à análise e ao projeto... diz mais ou menos que a codificação é onde a ação está...

(Os membros da equipe se entreolham e sorriem.)

Doug: Então vocês concordam com a metodologia XP?

Jamie (falando por ambos): Escrever código é o que fazemos, chefe!

Doug (rindo): É verdade, mas eu gostaria de vê-los perdendo um pouco menos de tempo codificando para depois recodificar e dedicando um pouco mais de tempo analisando o que precisa ser feito e projetando uma solução que funcione.

Vinod: Talvez possamos ter as duas coisas, agilidade com um pouco de disciplina.

Doug: Acho que sim, Vinod. Na realidade, tenho certeza disso.

Além das seis novas práticas apresentadas, a IXP modifica várias práticas XP existentes e redefine certas funções e responsabilidades para torná-las mais receptivas para projetos importantes de grandes empresas. Para uma discussão mais ampla sobre a IXP, visite **http://industrialxp.org**.

5.5 Outros modelos de processos ágeis

"Nossa profissão troca de metodologias como uma garota de 14 anos troca de roupas."

Stephen Hawrysh e Jim Ruprecht

A história da engenharia de software é recheada de metodologias e descrições de processos, métodos e notações de modelagem, ferramentas e tecnologias obsoletas. Todas atingiram certa notoriedade e foram ofuscadas por algo novo e (supostamente) melhor. Com a introdução de uma ampla variedade de modelos de processos ágeis – todos disputando aceitação pela comunidade de desenvolvimento de software –, o movimento ágil está seguindo o mesmo caminho histórico.[8]

Conforme citado na última seção, o modelo mais utilizado entre os modelos de processos ágeis é o Extreme Programming (XP). Porém, muitos outros têm sido propostos e encontram-se em uso no setor. Nesta seção, apresentamos um breve panorama de quatro métodos ágeis comuns: Scrum, DSSD, Modelagem Ágil (AM) e Processo Unificado Ágil (AUP).

[8] Isso não é algo ruim. Antes que um ou mais modelos ou métodos sejam aceitos como um padrão, todos devem competir para conquistar os corações e mentes dos engenheiros de software. Os "vencedores" evoluem e se transformam nas boas práticas, enquanto os "perdedores" desaparecem ou se fundem aos modelos vencedores.

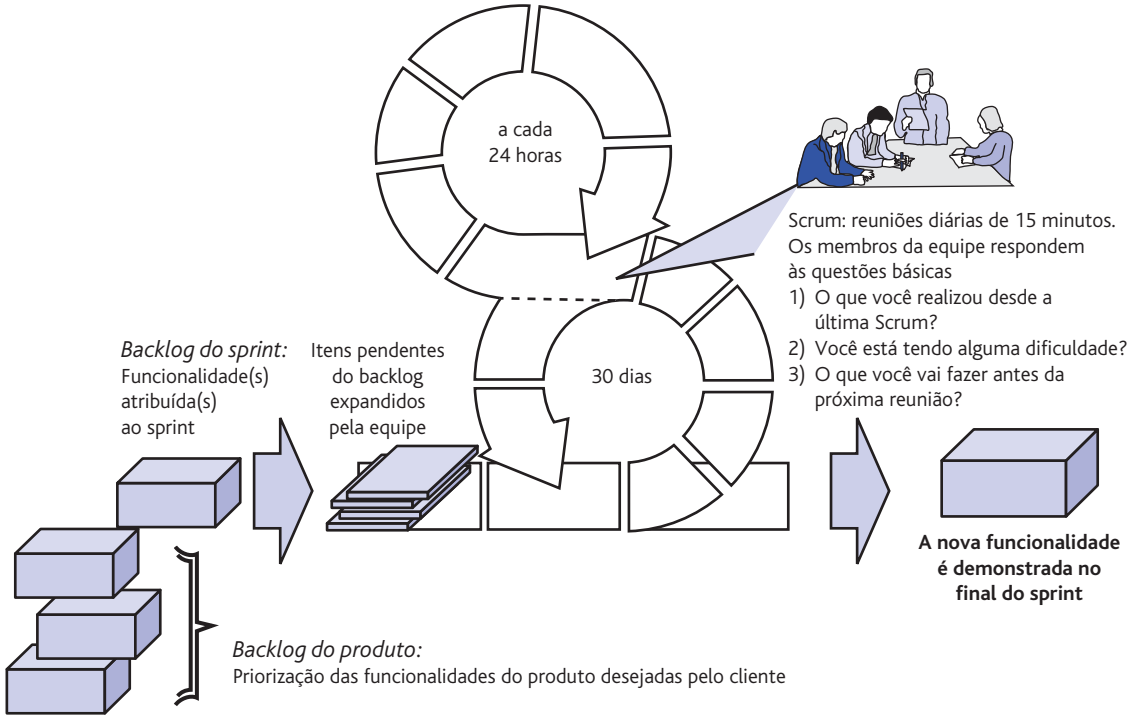


FIGURA 5.3 Fluxo do processo Scrum.

5.5.1 Scrum

Informações e recursos úteis sobre o Scrum podem ser encontrados em **www.controlchaos.com**.

Scrum (o nome provém de uma atividade que ocorre durante a partida de rugby)[9] é um método de desenvolvimento ágil de software concebido por Jeff Sutherland e sua equipe de desenvolvimento no início dos anos 1990. Mais recentemente, Schwaber e Beedle [Sch01b] realizaram desenvolvimentos adicionais nos métodos Scrum.

Os princípios do Scrum são coerentes com o manifesto ágil e são usados para orientar as atividades de desenvolvimento dentro de um processo que incorpora as seguintes atividades metodológicas: requisitos, análise, projeto, evolução e entrega. Em cada atividade metodológica, ocorrem tarefas realizadas dentro de um padrão de processo (discutido no parágrafo a seguir) chamado *sprint*. O trabalho realizado dentro de um sprint (o número de sprints necessários para cada atividade metodológica varia dependendo do tamanho e da complexidade do produto) é adaptado ao problema em questão e definido, e muitas vezes modificado em tempo real, pela equipe Scrum. O fluxo geral do processo Scrum está ilustrado na Figura 5.3.

O Scrum enfatiza o uso de um conjunto de padrões de processos de software [Noy02] que provaram ser eficazes para projetos com prazos de entrega apertados, requisitos mutáveis e urgência do negócio. Cada um desses padrões de processos define um conjunto de atividades de desenvolvimento:

[9] Um grupo de jogadores faz uma formação em torno da bola, e seus companheiros de equipe trabalham juntos (às vezes, de forma violenta!) para avançar com a bola em direção ao fundo do campo.

O Scrum engloba um conjunto de padrões de processos enfatizando prioridades de projeto, unidades de trabalho compartimentalizadas, comunicação e feedback frequente por parte dos clientes.

Backlog – uma lista com prioridades dos requisitos ou funcionalidades do projeto que fornecem valor comercial ao cliente. Os itens podem ser adicionados a esse registro a qualquer momento (é assim que as alterações são introduzidas). O gerente de produto avalia o registro e atualiza as prioridades conforme solicitado.

Sprints – consistem em unidades de trabalho solicitadas para atingir um requisito estabelecido no registro de trabalho (backlog) e que precisa ser ajustado dentro de um prazo já fechado (janela de tempo)[10] (tipicamente 30 dias). Alterações (por exemplo, itens do registro de trabalho – *backlog work items*) não são introduzidas durante execução de urgências (sprint). Portanto, o sprint permite que os membros de uma equipe trabalhem em um ambiente de curto prazo, porém estável.

Reuniões Scrum – são reuniões curtas (tipicamente 15 minutos), realizadas diariamente pela equipe Scrum. São feitas três perguntas-chave que são respondidas por todos os membros da equipe [Noy02]:

- O que você realizou desde a última reunião de equipe?
- Quais obstáculos está encontrando?
- O que planeja realizar até a próxima reunião da equipe?

Um líder de equipe, chamado *Scrum master,* conduz a reunião e avalia as respostas de cada integrante. A reunião Scrum, realizada diariamente, ajuda a equipe a revelar problemas em potencial o mais cedo possível. Ela também leva à "socialização do conhecimento" [Bee99] e, portanto, promove uma estrutura de equipe auto-organizada.

Demos – entrega do incremento de software ao cliente para que a funcionalidade implementada possa ser demonstrada e avaliada por ele. É importante notar que a demo pode não ter toda a funcionalidade planejada, mas sim funções que possam ser entregues no prazo estipulado.

Beedle e seus colegas [Bee99] apresentam uma ampla discussão sobre esses padrões: "O Scrum pressupõe a existência do caos...". Os padrões de processos do Scrum capacitam uma equipe de software a trabalhar com sucesso em um mundo onde é impossível eliminar a incerteza.

5.5.2 Método de Desenvolvimento de Sistemas Dinâmicos (DSDM)

Recursos úteis para o DSDM podem ser encontrados em **www.dsdm.org**.

O *Método de Desenvolvimento de Sistemas Dinâmicos* (DSDM, *Dynamic Systems Development Method*) [Sta97] é uma abordagem de desenvolvimento de software ágil que "oferece uma metodologia para construir e manter sistemas que satisfaçam restrições de prazo apertado por meio do uso da prototipação incremental em um ambiente de projeto controlado" [CCS02]. A filosofia DSDM baseia-se em uma versão modificada do princípio de Pareto – 80% de uma aplicação pode ser entregue em 20% do tempo que levaria para entregar a aplicação completa (100%).

[10] *Janela de tempo* (*time box*) é um termo de gerenciamento de projetos (consulte a Parte IV deste livro) que indica um período de tempo destinado para cumprir alguma tarefa.

O DSDM é um processo de software iterativo em que cada iteração segue a regra dos 80%. Ou seja, somente o trabalho suficiente é requisitado para cada incremento, para facilitar o movimento para o próximo incremento. Os detalhes restantes podem ser concluídos depois, quando outros requisitos do negócio forem conhecidos ou alterações tiverem sido solicitadas e acomodadas.

O DSDM Consortium (**www.dsdm.org**) é um grupo mundial de empresas-membro que assume coletivamente o papel de "mantenedor" do método. Esse consórcio definiu um modelo de processos ágeis, chamado *ciclo de vida DSDM*, que começa com um *estudo de viabilidade*, o qual estabelece os requisitos básicos e as restrições do negócio, e é seguido por um *estudo do negócio*, o qual identifica os requisitos de função e informação. Então, o DSDM define três diferentes ciclos iterativos:

O DSDM é uma metodologia de processos que pode adotar a tática de outra metodologia ágil, como a XP.

Iteração de modelos funcionais – produz um conjunto de protótipos incrementais que demonstram funcionalidade para o cliente. (Observação: todos os protótipos DSDM são feitos com a intenção de que evoluam para a aplicação final entregue ao cliente.) Durante esse ciclo iterativo, o objetivo é reunir requisitos adicionais ao se obter feedback dos usuários, à medida que eles testam o protótipo.

Iteração de projeto e desenvolvimento – revê os protótipos desenvolvidos durante a iteração de modelos funcionais para assegurar-se de que cada um tenha passado por um processo de engenharia para capacitá-los a oferecer, aos usuários, valor de negócio em termos operacionais. Em alguns casos, a iteração de modelos funcionais e a iteração de projeto e desenvolvimento ocorrem ao mesmo tempo.

Implementação – coloca a última versão do incremento de software (um protótipo "operacionalizado") no ambiente operacional. Deve-se notar que: (1) o incremento pode não estar 100% completo ou (2) alterações podem vir a ser solicitadas conforme o incremento seja alocado. Em qualquer um dos casos, o trabalho de desenvolvimento do DSDM continua, retornando-se à atividade de iteração do modelo funcional.

O DSDM pode ser combinado com a XP (Seção 5.4) para fornecer uma abordagem combinada que defina um modelo de processos confiável (o ciclo de vida do DSDM) com as práticas básicas (XP) necessárias para construir incrementos de software.

5.5.3 Modelagem Ágil (AM)

Muita informação sobre a modelagem ágil pode ser encontrada em: **www.agilemodeling.com**.

Existem muitas situações em que engenheiros de software têm de desenvolver sistemas grandes e críticos para o negócio. O escopo e a complexidade desses sistemas devem ser modelados de modo que (1) todas as partes envolvidas possam entender melhor quais requisitos devem ser atingidos, (2) o problema possa ser subdividido eficientemente entre as pessoas que têm de solucioná-lo e (3) a qualidade possa ser avaliada enquanto se está projetando e desenvolvendo o sistema. Porém, em alguns casos pode ser desencorajador gerenciar o volume de notação exigido, o grau de formalismo sugerido, o mero tamanho dos modelos para grandes projetos e a dificuldade em manter o(s) modelo(s) à

medida que ocorrem mudanças. Existe uma metodologia ágil para a modelagem de engenharia de software que possa fornecer algum alívio?

No "The Official Agile Modeling Site", Scott Ambler [Amb02a] descreve *modelagem ágil* (AM) da seguinte maneira:

> Modelagem ágil (AM) consiste em uma metodologia prática, voltada para a modelagem e documentação de sistemas baseados em software. Simplificando, modelagem ágil consiste em um conjunto de valores, princípios e práticas voltados para a modelagem do software que podem ser aplicados a um projeto de desenvolvimento de software de forma leve e eficiente. Os modelos ágeis são mais eficientes do que os tradicionais pelo fato de serem simplesmente bons, pois não têm a obrigação de ser perfeitos.

"Um dia, estava em uma farmácia tentando achar um remédio para resfriado... Não foi fácil. Havia uma parede inteira de produtos. Fica-se lá procurando: 'Bem, este tem ação imediata, mas este outro tem efeito mais duradouro...' O que é mais importante, o presente ou o futuro?"

Jerry Seinfeld

A modelagem ágil adota todos os valores coerentes com o manifesto ágil. Sua filosofia reconhece que uma equipe ágil deve ter a coragem de tomar decisões que possam causar a rejeição de um projeto e sua refatoração. A equipe também deve ter humildade para reconhecer que os profissionais de tecnologia não possuem todas as respostas e que os experts em negócios e outros envolvidos devem ser respeitados e integrados ao processo.

Embora a AM sugira uma ampla variedade de princípios de modelagem "básicos" e "suplementares", os que a tornam única são [Amb02a]:

Modelar com um objetivo. O desenvolvedor que utilizar a AM deve ter um objetivo antes de criar o modelo (por exemplo, comunicar informações ao cliente ou ajudar a compreender melhor algum aspecto do software). Uma vez identificado o objetivo, ficará mais evidente o tipo de notação a ser utilizado e o nível de detalhamento necessário.

Usar vários modelos. Há muitos modelos e notações diferentes que podem ser usados para descrever software. Para a maioria dos projetos, somente um subconjunto é essencial. A AM sugere que, para propiciar a percepção necessária, cada modelo deve apresentar um aspecto diferente do sistema e devem ser usados somente aqueles que valorizem esses modelos para o público pretendido.

"Viajar leve" é uma filosofia apropriada para todo o trabalho de engenharia de software. Construa apenas os modelos que forneçam valor... nem mais, nem menos.

Viajar leve. Conforme o trabalho de engenharia de software prossegue, conserve apenas aqueles modelos que terão valor no longo prazo e desfaça-se do restante. Todo artefato mantido deve sofrer manutenção à medida que mudanças ocorram. Isso representa trabalho que retarda a equipe. Ambler [Amb02a] observa que "Toda vez que se opta por manter um modelo, troca-se a agilidade pela conveniência de ter aquela informação acessível para a equipe de uma forma abstrata (já que, potencialmente, aumenta a comunicação dentro da equipe, assim como com os envolvidos no projeto)".

Conteúdo é mais importante do que a representação. A modelagem deve transmitir informações para seu público pretendido. Um modelo sintaticamente perfeito que transmita pouco conteúdo útil não possui tanto valor quanto aquele com notações falhas que, no entanto, fornece conteúdo valioso para seu público-alvo.

Conhecer os modelos e as ferramentas utilizadas para criá-los. Compreenda os pontos fortes e fracos de cada modelo e as ferramenta usadas para criá-lo.

Adaptar localmente. A modelagem deve ser adaptada às necessidades da equipe ágil.

Um segmento de vulto da comunidade da engenharia de software adotou a linguagem de modelagem unificada (Unified Modeling Language, UML)[11] como o método preferido para análise representativa e para modelos de projeto. O Processo Unificado (Capítulo 4) foi desenvolvido para fornecer uma metodologia para a aplicação da UML. Scott Ambler [Amb06] desenvolveu uma versão simplificada do UP que integra sua filosofia de modelagem ágil.

5.5.4 Processo Unificado Ágil

O *Processo Unificado Ágil* (AUP, *Agile Unified Process*) adota uma filosofia "serial para o que é amplo" e "iterativa para o que é particular" [Amb06] no desenvolvimento de sistemas computadorizados. Adotando as atividades em fases UP clássicas – concepção, elaboração, construção e transição –, o AUP fornece uma camada serial (isto é, uma sequência linear de atividades de engenharia de software) que permite à equipe visualizar o fluxo do processo geral de um projeto de software. Entretanto, dentro de cada atividade, a equipe itera para alcançar a agilidade e entregar incrementos de software significativos para os usuários o mais rápido possível. Cada iteração AUP trata das seguintes atividades [Amb06]:

- *Modelagem*. Representações UML do universo do negócio e do problema são criadas. Entretanto, para permanecerem ágeis, esses modelos devem ser "suficientemente bons e adequados" [Amb06] para possibilitar que a equipe prossiga.
- *Implementação*. Os modelos são traduzidos em código-fonte.
- *Testes*. Como a XP, a equipe projeta e executa uma série de testes para descobrir erros e assegurar que o código-fonte se ajuste aos requisitos.
- *Entrega*. Como a atividade de processo genérica discutida no Capítulo 3, neste contexto a entrega se concentra no fornecimento de um incremento de software e na obtenção de feedback dos usuários.
- *Configuração e gerenciamento de projeto*. No contexto do AUP, gerenciamento de configuração (Capítulo 29) refere-se a gerenciamento de alterações, de riscos e de controle de qualquer artefato[12] persistente que sejam produzidos por uma equipe. O gerenciamento de projeto monitora e controla o progresso de uma equipe e coordena suas atividades.
- *Gerenciamento do ambiente*. Coordena a infraestrutura de processos que inclui padrões, ferramentas e outras tecnologias de suporte disponíveis para a equipe.

[11] Um breve tutorial sobre a UML é apresentado no Apêndice 1.

[12] *Artefato persistente* é um modelo ou documento ou pacote de testes produzido pela equipe que será mantido por um período de tempo indeterminado. *Não* será descartado quando o incremento de software for entregue.

Embora o AUP tenha conexões históricas e técnicas com a linguagem de modelagem unificada, é importante notar que a modelagem UML pode ser usada com qualquer modelo de processo ágil descrito neste capítulo.

FERRAMENTAS DO SOFTWARE

Engenharia de requisitos

Objetivo: O objetivo das ferramentas de desenvolvimento ágil é auxiliar em um ou mais aspectos do desenvolvimento ágil, com ênfase em facilitar a geração rápida de software operacional. Essas ferramentas também podem ser usadas quando forem aplicados modelos de processos prescritivos (Capítulo 4).

Mecanismos: O mecanismo das ferramentas é variado. Em geral, conjuntos de ferramentas ágeis englobam suporte automatizado para o planejamento de projetos, desenvolvimento de casos de uso, reunião de requisitos, projeto rápido, geração de código e testes.

Ferramentas representativas:[13]

Observação: como o desenvolvimento ágil é um tópico importante, a maioria dos fornecedores de ferramentas de software tende a vender ferramentas que aceitam a metodologia ágil. As ferramentas aqui mencionadas têm características que as tornam particularmente úteis para projetos ágeis.

OnTime, desenvolvida pela Axosoft (**www.axosoft.com**), fornece suporte para gerenciamento de processo ágil para uma variedade de atividades técnicas dentro do processo.

Ideogramic UML, desenvolvida pela Ideogramic (**http://ideogramic-uml.software.informer.com/**), é um conjunto de ferramentas UML desenvolvido para uso em processo ágil.

Together Tool Set, distribuída pela Borland (**www.borland.com**), fornece uma mala de ferramentas que dão suporte para muitas atividades técnicas na XP e em outros processos ágeis.

5.6 Um conjunto de ferramentas para o processo ágil

Alguns proponentes da filosofia ágil argumentam que as ferramentas de software automatizadas (por exemplo, ferramentas para projetos) deveriam ser vistas como um suplemento secundário para as atividades, e não como fundamental para o sucesso da equipe. Entretanto, Alistair Cockburn [Coc04] sugere que ferramentas podem trazer vantagens e que "equipes ágeis enfatizam o uso de ferramentas que permitem o fluxo rápido de compreensão. Algumas dessas ferramentas são sociais, iniciando-se até no estágio de contratação de pessoal. Algumas são tecnológicas, auxiliando equipes distribuídas a simular sua presença física. Muitas são físicas, permitindo sua manipulação em workshops."

O "conjunto de ferramentas" que suporta os processos ágeis se concentra mais nas questões pessoais do que nas questões tecnológicas.

"Ferramentas" voltadas para a comunicação e para a colaboração são, em geral, de baixa tecnologia e incorporam qualquer mecanismo ("proximidade física, quadros brancos, papéis para pôster, fichas e lembretes adesivos" [Coc04] ou modernas técnicas de rede social) que forneça informações e coordenação entre desenvolvedores ágeis. A comunicação ativa é obtida por meio de dinâmicas de grupo (por exemplo, programação em pares), enquanto a comunicação passiva é obtida por meio dos "irradiadores de informações" (por exemplo, um display de um painel fixo que apresente o status geral dos diferentes componentes de um incremento). As ferramentas de gerenciamento de projeto dão pouca ênfase ao diagrama de Gantt e o substituem por gráficos de valores ganhos ou "gráficos de testes criados e cruzados com os anteriores... outras ferramentas

[13] A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

ágeis são utilizadas para otimizar o ambiente no qual a equipe ágil trabalha (por exemplo, áreas de reunião mais eficientes), ampliar a cultura da equipe promovendo interações sociais (por exemplo, equipes próximas umas das outras), dispositivos físicos (por exemplo, lousas eletrônicas) e melhoria do processo (por exemplo, programação em pares ou janela de tempo)" [Coc04].

Algumas dessas coisas são realmente ferramentas? Serão, caso facilitem o trabalho desenvolvido por um membro da equipe ágil e venham a aprimorar a qualidade do produto final.

5.7 Resumo

Em uma economia moderna, as condições de mercado mudam rapidamente, as necessidades do cliente e do usuário evoluem e novos desafios competitivos surgem sem aviso. Os profissionais têm de assumir uma abordagem de engenharia de software que permita que permaneçam ágeis – definindo processos que sejam manipuláveis, adaptáveis e sem excessos, somente com o conteúdo essencial que possa se adequar às necessidades do mundo dos negócios moderno.

Uma filosofia ágil para a engenharia de software enfatiza quatro elementos-chave: a importância das equipes que se auto-organizam, que têm controle sobre o trabalho por elas realizado; a comunicação e a colaboração entre os membros da equipe e entre os desenvolvedores e seus clientes; o reconhecimento de que as mudanças representam oportunidades; e a ênfase na entrega rápida do software para satisfazer o cliente. Os modelos de processos ágeis foram feitos para tratar de todas essas questões.

Extreme Programming (XP) é o processo ágil mais amplamente utilizado. Organizada em quatro atividades metodológicas – planejamento, projeto, codificação e testes – a XP sugere várias técnicas poderosas e inovadoras que possibilitam a uma equipe ágil criar versões de software com frequência, propiciando recursos e funcionalidade descritos previamente e priorizados pelos envolvidos.

Outros modelos de processos ágeis também enfatizam a colaboração humana e a auto-organização das equipes, mas definem suas próprias atividades metodológicas e selecionam diferentes pontos de ênfase. Por exemplo, o Scrum enfatiza o uso de um conjunto de padrões de software que se mostrou eficaz para projetos com cronogramas apertados, requisitos mutáveis e aspectos críticos de negócio. Cada padrão de processo define um conjunto de tarefas de desenvolvimento e permite à equipe Scrum construir um processo que se adapte às necessidades do projeto. O método de desenvolvimento de sistemas dinâmicos (DSDM) defende o uso de um cronograma de tempos definidos (janela de tempo) e sugere que apenas o trabalho suficiente seja requisitado para cada incremento de software, para facilitar o movimento em direção ao incremento seguinte. A modelagem ágil (AM) afirma que modelagem é essencial para todos os sistemas, mas a complexidade, tipo e tamanhos de um modelo devem ser balizados pelo software a ser construído. O processo unificado ágil (AUP) adota a filosofia do "serial para o que é amplo" e "iterativa para o que é particular" para o desenvolvimento de software.

Problemas e pontos a ponderar

5.1. Releia o "Manifesto for Agile Software Development" no início deste capítulo. Você consegue exemplificar uma situação em que um ou mais dos quatro "valores" poderiam levar a equipe a ter problemas?

5.2. Descreva agilidade (para projetos de software) com suas próprias palavras.

5.3. Por que um processo iterativo facilita o gerenciamento de mudanças? Todos os processos ágeis discutidos neste capítulo são iterativos? É possível concluir um projeto com apenas uma iteração e ainda assim permanecer ágil? Justifique suas respostas.

5.4. Cada um dos processos ágeis poderia ser descrito usando-se as atividades metodológicas genéricas citadas no Capítulo 3? Construa uma tabela que associe as atividades genéricas às atividades definidas para cada processo ágil.

5.5. Tente elaborar mais um "princípio de agilidade" que ajudaria uma equipe de engenharia de software a se tornar mais adaptável.

5.6. Escolha um princípio de agilidade citado na Seção 5.3.1 e tente determinar se cada um dos modelos de processos apresentados neste capítulo demonstra o princípio. (Observação: apresentamos apenas uma visão geral desses modelos de processos; portanto, talvez não seja possível determinar se um princípio foi ou não tratado por um ou mais dos modelos, a menos que você pesquise mais a respeito, o que não é exigido neste problema).

5.7. Por que os requisitos mudam tanto? Afinal de contas, as pessoas não sabem o que elas querem?

5.8. A maior parte dos modelos de processos ágeis recomenda comunicação face a face. Mesmo assim, hoje em dia os membros de uma equipe de software e seus clientes podem estar geograficamente separados uns dos outros. Você acredita que isso implique que a separação geográfica seja algo a ser evitado? Você é capaz de imaginar maneiras de superar esse problema?

5.9. Escreva uma história de usuário XP que descreva o recurso "sites favoritos" ou "favoritos" disponível na maioria dos navegadores Web.

5.10. O que é uma solução pontual na XP?

5.11. Descreva com suas próprias palavras os conceitos de refatoração e programação em pares da XP.

5.12. Usando a planilha de padrões de processos apresentada no Capítulo 3, desenvolva um padrão de processo para qualquer um dos padrões Scrum da Seção 5.5.1.

5.13. Visite o site Official Agile Modeling e faça uma lista completa de todos os princípios básicos e complementares do AM.

5.14. O conjunto de ferramentas proposto na Seção 5.6 oferece suporte a muitos dos aspectos "menos prioritários" dos métodos ágeis. Como a comunicação é tão importante, recomende um conjunto de ferramentas real que poderia ser usado para melhorar a comunicação entre os envolvidos de uma equipe ágil.

Leituras e fontes de informação complementares

A filosofia geral e os princípios subjacentes do desenvolvimento de software ágil são considerados em profundidade em muitos dos livros citados neste capítulo. Além disso, livros de Pichler (*Agile Project Management with Scrum: Creating Products that Customers Love,* Addison-Wesley, 2010), Highsmith (*Agile Project Management: Creating Innovative*

Products, 2ª ed. Addison-Wesley, 2009), Shore e Chromatic (*The Art of Agile Development,* O'Reilly Media, 2008), Hunt (*Agile Software Construction,* Springer, 2005) e Carmichael e Haywood (*Better Software Faster,* Prentice Hall, 2002) trazem discussões interessantes sobre o tema. Aguanno (*Managing Agile Projects,* Multi-Media Publications, 2005) e Larman (*Agile and Iterative Development: A Manager's Guide,* Addison-Wesley, 2003) apresentam uma visão geral sobre gerenciamento e consideram as questões envolvidas no gerenciamento de projetos. Highsmith (*Agile Software Development Ecosystems,* Addison-Wesley, 2002) retrata uma pesquisa de princípios, processos e práticas ágeis. Uma discussão que vale a pena sobre o delicado equilíbrio entre agilidade e disciplina é fornecida por Booch e seus colegas (*Balancing Agility and Discipline,* Addison-Wesley, 2004).

Martin (*Clean Code: A Handbook of Agile Software Craftsmanship,* Prentice-Hall, 2009) enumera os princípios, padrões e práticas necessários para desenvolver "código limpo" em um ambiente de engenharia de software ágil. Leffingwell (*Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise*, Addison-Wesley, 2011; e *Scaling Software Agility: Best Practices for Large Enterprises,* Addison-Wesley, 2007) discute estratégias para dar maior corpo às práticas ágeis para poderem ser usadas em grandes projetos. Lippert e Rook (*Refactoring in Large Software Projects: Performing Complex Restructurings Successfully,* Wiley, 2006) discutem o uso da refatoração quando aplicada a sistemas grandes e complexos. Stamelos e Sfetsos (*Agile Software Development Quality Assurance,* IGI Global, 2007) trazem técnicas SQA que estão em conformidade com a filosofia ágil.

Foram escritos dezenas de livros sobre Extreme Programming ao longo da última década. Beck (*Extreme Programming Explained: Embrace Change,* 2ª ed., Addison-Wesley, 2004) ainda é o tratado de maior autoridade sobre o tema. Além disso, Jeffries e seus colegas (*Extreme Programming Installed,* Addison-Wesley, 2000), Succi e Marchesi (*Extreme Programming Examined,* Addison-Wesley, 2001), Newkirk e Martin (*Extreme Programming in Practice,* Addison-Wesley, 2001) e Auer e seus colegas (*Extreme Programming Applied: Play to Win,* Addison-Wesley, 2001) fornecem uma discussão básica da XP, juntamente com uma orientação sobre como melhor aplicá-la. McBreen (*Questioning Extreme Programming,* Addison-Wesley, 2003) adota uma visão crítica em relação à XP, definindo quando e onde ela é apropriada. Uma análise aprofundada da programação em pares é apresentada por McBreen (*Pair Programming Illuminated,* Addison-Wesley, 2003).

Kohut (*Professional Agile Development Process: Real World Development Using SCRUM,* Wrox, 2013), Rubin (*Essential Scrum: A Practical Guide to the Most Popular Agile Process*, Addison-Wesley, 2012), Larman e Vodde (*Scaling Lean and Agile Development: Thinking and Organizational Tools for Large Scale Scrum,* Addison-Wesley, 2008) e Schwaber (*The Enterprise and Scrum,* Microsoft Press, 2007) discutem o uso de Scrum para projetos que têm grande impacto comercial. Os detalhes práticos do Scrum são debatidos por Cohn (*Succeeding with Agile*, Addison-Wesley, 2009) e Schwaber e Beedle (*Agile Software Development with SCRUM,* Prentice-Hall, 2001). Tratados úteis sobre o DSDM foram escritos pelo DSDM Consortium (*DSDM: Business Focused Development,* 2ª ed., Pearson Education, 2003) e Stapleton (*DSDM: The Method in Practice,* Addison-Wesley, 1997).

Livros de Ambler e Lines (*Disciplined Agile Delivery: A Practitioner's Guide to Agile Delivery in the Enterprise*, IBM Press, 2012) e Poppendieck e Poppendieck (*Lean Development: An Agile Toolkit for Software Development Managers,* Addison-Wesley, 2003) dão diretrizes para gerenciar e controlar projetos ágeis. Ambler e Jeffries (*Agile Modeling,* Wiley, 2002) discutem a AM com certa profundidade.

Uma grande variedade de fontes de informação sobre desenvolvimento de software ágil está disponível na Internet. Uma lista atualizada de referências relevantes (em inglês) para o processo ágil pode ser encontrada no site: **www.mhhe.com/pressman**.

Aspectos humanos da engenharia de software

6

Em uma edição especial da *IEEE Software*, os editores convidados [deS09] fizeram a seguinte observação:

> A engenharia de software tem uma fartura de técnicas, ferramentas e métodos projetados para melhorar tanto o processo de desenvolvimento de software quanto o produto final. Aprimoramentos técnicos continuam a surgir e a gerar resultados animadores. No entanto, software não é simplesmente um produto de soluções técnicas adequadas aplicadas a hábitos técnicos inadequados. Software é desenvolvido por pessoas, usado por pessoas e dá suporte à interação entre pessoas. Assim, características, comportamento e cooperação humanos são fundamentais no desenvolvimento prático de software.

Ao longo dos capítulos posteriores a este, vamos discutir as "técnicas, ferramentas e métodos" que resultarão na criação de um produto de software bem-sucedido. Mas, antes disso, é fundamental entender que, sem pessoas habilitadas e motivadas, o sucesso é improvável.

Conceitos-chave

PANORAMA

O que é? Todos nós temos a tendência de nos dedicarmos à linguagem de programação mais recente, aos melhores e novos métodos de projeto, ao processo ágil mais moderno ou à impressionante ferramenta de software recém lançada. Mas no frigir dos ovos são *pessoas* que constroem software de computador. E, por isso, os aspectos humanos da engenharia de software frequentemente têm tanto a ver com o sucesso de um projeto quanto a melhor e mais recente tecnologia.

Quem realiza? Indivíduos e equipes realizam o trabalho de engenharia de software. Em alguns casos, apenas uma pessoa é responsável pela maior parte do trabalho, mas no caso de produção de software em nível industrial, uma equipe de pessoas o realiza.

Por que é importante? Uma equipe de software só será bem-sucedida se sua dinâmica estiver correta. Às vezes, os engenheiros de software têm a reputação de não trabalhar bem com outras pessoas. Na verdade, é fundamental que os engenheiros de software de uma equipe trabalhem bem com seus colegas e com outros envolvidos no produto a ser construído.

Quais são as etapas envolvidas? Primeiramente, é preciso entender as características pessoais de um engenheiro de software bem-sucedido e, então, tentar imitá-las. Em seguida, você deve compreender a complexa psicologia do trabalho de engenharia de software para que possa navegar por um projeto sem riscos. Então, precisa entender a estrutura e a dinâmica de uma equipe de software, pois a engenharia de software baseada no trabalho em equipe é comum em um cenário industrial. Por fim, você deve compreender o impacto das mídias sociais, da nuvem e de outras ferramentas colaborativas.

Qual é o artefato? Uma melhor compreensão das pessoas, do processo e do produto final.

Como garantir que o trabalho foi realizado corretamente? Passe um tempo observando como os engenheiros de software bem-sucedidos fazem seu trabalho e ajuste sua abordagem para tirar proveito dos pontos positivos do projeto deles.

6.1 Características de um engenheiro de software

"A maioria dos bons programadores faz seu trabalho não porque espera pagamento ou bajulação pública, mas porque é divertido programar."

Linus Torvalds

Então você quer ser engenheiro de software? Obviamente, precisa dominar o material técnico, aprender e aplicar as habilidades exigidas para entender o problema, projetar uma solução eficaz, construir o software e testá-lo com a finalidade de produzir a mais alta qualidade possível. Você precisa gerenciar mudanças, comunicar-se com os envolvidos e usar ferramentas adequadas nas situações apropriadas. Tudo isso é discutido com detalhes mais adiante neste livro.

Quais são as características pessoais de um engenheiro de software competente?

Mas existem outras coisas igualmente importantes – os aspectos humanos que o tornarão um engenheiro de software competente. Erdogmus [Erd09] identifica sete características pessoais que estão presentes quando um engenheiro de software demonstra comportamento "super profissional".

Um engenheiro de software competente tem um senso de *responsabilidade individual*. Isso implica a determinação de cumprir suas promessas para colegas, para os envolvidos e para a gerência. Significa que ele fará o que precisar ser feito, quando for necessário, executando um esforço adicional para a obtenção de um resultado bem-sucedido.

Um engenheiro de software competente tem *consciência aguçada* das necessidades dos outros membros de sua equipe, dos envolvidos que solicitaram uma solução de software para um problema existente e dos gerentes que têm controle global sobre o projeto que vai gerar essa solução. É capaz de observar o ambiente em que as pessoas trabalham e de adaptar seu comportamento a ele e às próprias pessoas.

Um engenheiro de software competente é *extremamente honesto*. Se ele vê um projeto falho, aponta os defeitos de maneira construtiva, mas honesta. Se instado a distorcer fatos sobre cronogramas, recursos, desempenho ou outras características do produto ou projeto, opta por ser realista e sincero.

Um engenheiro de software competente mostra *resiliência sob pressão*. Conforme mencionamos anteriormente no livro, a engenharia de software está sempre à beira do caos. A pressão (e o caos que pode resultar) vem em muitas formas – mudanças nos requisitos e nas prioridades, envolvidos ou colegas exigentes, um gerente irrealista ou autoritário. Mas um engenheiro de software competente é capaz de suportar a pressão de modo que seu desempenho não seja prejudicado.

Um engenheiro de software competente tem *elevado senso de lealdade*. De boa vontade, compartilha os créditos com seus colegas. Tenta evitar conflitos de interesse e nunca age no sentido de sabotar o trabalho dos outros.

Um engenheiro de software competente mostra *atenção aos detalhes*. Isso não significa obsessão com a perfeição, mas sugere que ele considera atentamente as decisões técnicas que toma diariamente, em comparação com critérios mais amplos (por exemplo, desempenho, custo, qualidade) que foram estabelecidos para o produto e para o projeto.

Por último, um engenheiro de software competente é pragmático. Reconhece que a engenharia de software não é uma religião na qual devem ser seguidas regras dogmáticas, mas sim uma disciplina que pode ser adaptada de acordo com as circunstâncias.

6.2 A psicologia da engenharia de software

Em um artigo seminal sobre a psicologia da engenharia de software, Bill Curtis e Diane Walz [Cur90] sugerem um modelo comportamental em camadas para desenvolvimento de software (Figura 6.1). No nível individual, a psicologia da engenharia de software se concentra no reconhecimento do problema a ser resolvido, nas habilidades exigidas para solucionar o problema e na motivação necessária para concluir a solução dentro das restrições estabelecidas pelas camadas externas do modelo. Nos níveis da equipe e do projeto, dinâmicas de grupo se tornam o fator dominante. Aqui, a estrutura da equipe e fatores sociais governam o sucesso. A comunicação, a colaboração e a coordenação do grupo são tão importantes quanto as habilidades dos membros individuais da equipe. Nas camadas externas, o comportamento organizacional governa as ações da empresa e sua resposta para o meio empresarial.

No nível das equipes, Sawyer e seus colegas [Saw08] sugerem que elas frequentemente estabelecem fronteiras artificiais que reduzem a comunicação e, como consequência, a eficácia da equipe. Sugerem ainda um conjunto de "papéis que ultrapassam fronteiras", que permite que os membros de uma equipe de software transponham eficazmente suas fronteiras. Os papéis a seguir podem ser atribuídos explicitamente ou evoluir naturalmente.

Quais papéis os membros de uma equipe de software desempenham?

- *Embaixador* – representa a equipe para a clientela de fora, com o objetivo de negociar tempo e recursos e obter o retorno dos envolvidos.
- *Patrulha* – cruza a fronteira da equipe para reunir informações organizacionais. "O patrulhamento pode incluir o mapeamento de mercados externos, busca de novas tecnologias, identificação de atividades relevantes fora da equipe e descoberta de focos de concorrência em potencial" [Saw08].

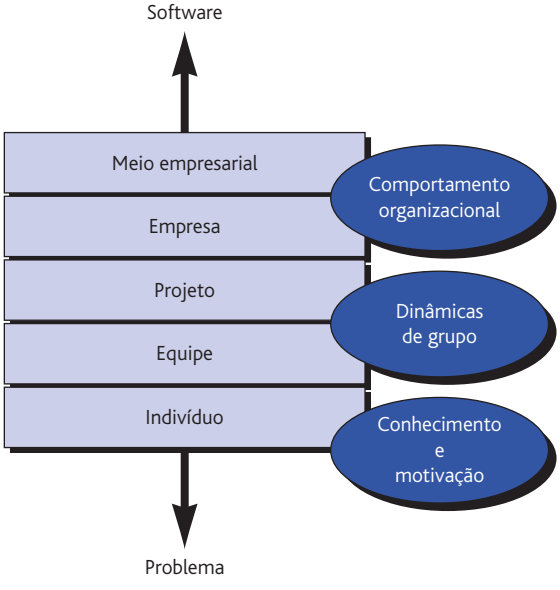


FIGURA 6.1 Um modelo comportamental em camadas para engenharia de software (adaptado de [Cur90]).

- *Guarda* – protege o acesso aos artefatos e outras informações da equipe.
- *Sentinela* – controla o fluxo de informações enviadas para a equipe pelos envolvidos e por outros.
- *Coordenador* – concentra-se na comunicação horizontal entre a equipe e dentro da organização (por exemplo, discutindo um problema de projeto específico com um grupo de especialistas da organização).

6.3 A equipe de software

Em seu livro clássico, *Peopleware,* Tom DeMarco e Tim Lister [DeM98] discutem a coesão de uma equipe de software:

> Há uma tendência em se utilizar a palavra *equipe* de forma constante e vaga na área de negócios, denominando qualquer grupo de profissionais designados para trabalharem juntos de "equipe". Entretanto, muitos deles não se assemelham a equipes. Não há uma definição comum de sucesso nem um espírito de equipe identificável. O que falta é um fenômeno que se denomina *consistência*.

O que é uma equipe "consistente"?

> Uma equipe consistente é um grupo de pessoas tão coesas, que o todo é maior que a soma das partes...
>
> Quando uma equipe começa a ser consistente, a probabilidade de sucesso aumenta muito. A equipe pode se tornar imbatível, um rolo compressor de sucesso... Não é preciso gerenciá-la do modo tradicional e, com certeza, não precisará ser motivada. Ela adquire velocidade e ímpeto.

DeMarco e Lister sustentam que os membros de equipes consistentes são significativamente mais produtivos e mais motivados do que a média. Compartilham de um objetivo comum, de uma cultura comum e, em muitos casos, de um senso de pertencimento a uma equipe de elite que os torna únicos.

Não existe nenhum método infalível para se criar uma equipe consistente. Porém, existem atributos normalmente encontrados em equipes de software eficazes.[1] Miguel Carrasco [Car08] sugere que uma equipe de software eficiente deve estabelecer um *senso de propósito*. Por exemplo, se todos os membros da equipe concordam que o objetivo dela é desenvolver software que vai transformar uma categoria de produto e, como consequência, transformar sua empresa em líder do setor, eles têm um forte senso de propósito. Uma equipe eficaz também deve incorporar um *senso de envolvimento* que permita a cada membro sentir que suas qualidades e contribuições são valiosas.

Uma equipe eficaz deve promover um *senso de confiança*. Os engenheiros de software da equipe devem confiar nas habilidades e na competência de seus colegas e gerentes. A equipe deve estimular um *senso de melhoria,* refletindo periodicamente em sua abordagem de engenharia de software e buscando maneiras de melhorar seu trabalho.

[1] Bruce Tuckman observa que as equipes bem-sucedidas passam por quatro fases (Formação, Ataque, Regulamentação e Execução) no caminho para se tornarem produtivas (**http://www.realsoftwaredevelopment.com/7-key-attributes-of-high-performance-software-development-teams/**).

Uma equipe de software eficaz é diversificada, preenchida por pessoas que têm senso de propósito, envolvimento, confiança e melhoria.

As equipes de software mais eficazes são diversificadas, no sentido de combinarem uma variedade de diferentes qualidades. Técnicos altamente capacitados são complementados por membros que podem ter menos base técnica, mas compreendem melhor as necessidades dos envolvidos.

Por que as equipes não conseguem ser consistentes?

Porém, nem todas as equipes são eficazes e nem todas são consistentes. Na verdade, muitas sofrem do que Jackman [Jac98] denomina de "toxicidade de equipe". Ela define cinco fatores que "promovem um ambiente em equipe potencialmente tóxico": uma atmosfera de trabalho frenética; alto grau de frustração que causa atrito entre os membros da equipe; um processo de software fragmentado ou coordenado de forma deficiente; uma definição nebulosa dos papéis dentro da equipe de software; e contínua e repetida exposição a falhas.

Para evitar um ambiente de trabalho frenético, a equipe deve ter acesso a todas as informações exigidas para cumprir a tarefa. As principais metas e objetivos, uma vez definidos, não devem ser modificados, a menos que seja absolutamente necessário. Uma equipe pode evitar frustrações se lhe for oferecida, tanto quanto possível, responsabilidade para tomada de decisão. Um processo inapropriado (por exemplo, tarefas onerosas ou desnecessárias ou artefatos mal selecionados) pode ser evitado por meio da compreensão do produto a ser desenvolvido, das pessoas que realizam o trabalho e pela permissão para que a equipe selecione o modelo do processo. A própria equipe deve estabelecer seus mecanismos de responsabilidades (revisões técnicas[2] são excelentes meios para conseguir isso) e definir uma série de abordagens corretivas quando um membro falhar em suas atribuições. E, por fim, a chave para evitar uma atmosfera de derrota consiste em estabelecer técnicas baseadas no trabalho em equipe voltadas para realimentação (feedback) e solução de problemas.

"Nem todo grupo é uma equipe, nem toda equipe é eficaz."

Glenn Parker

Somando-se às cinco toxinas descritas por Jackman, uma equipe de software frequentemente despende esforços com as diferentes características de seus membros. Uns são extrovertidos; outros, introvertidos. Uns coletam informações intuitivamente, destilando conceitos amplos de fatos disparatados. Outros processam informações linearmente, coletando e organizando detalhes minuciosos dos dados fornecidos. Alguns se sentem confortáveis tomando decisões apenas quando um argumento lógico e ordenado for apresentado. Outros são intuitivos, acostumados a tomar decisões baseadas em percepções. Certos desenvolvedores querem um cronograma detalhado, preenchido por tarefas organizadas que os tornem aptos a ter proximidade com elementos do projeto. Outros, ainda, preferem um ambiente mais espontâneo, no qual resultados e questões abertas são aceitáveis. Alguns trabalham arduamente para conseguir que as etapas sejam concluídas bem antes da data estabelecida, evitando, portanto, estresse na medida em que a data-limite se aproxima, enquanto outros são energizados pela correria em fazer até o último minuto da data-limite. Reconhecer as diferenças humanas, junto com outras diretrizes apresentadas nesta seção, proporciona uma maior probabilidade de se criar equipes consistentes.

[2] Revisões técnicas são tratadas em detalhes no Capítulo 20.

6.4 Estruturas de equipe

A melhor estrutura de equipe depende do estilo de gerenciamento das organizações, da quantidade de pessoas na equipe e seus níveis de habilidade e do grau de dificuldade geral do problema. Mantei [Man81] descreve vários fatores que devem ser considerados ao planejarmos a estrutura da equipe de engenharia de software: dificuldade do problema a ser resolvido; "tamanho" do programa (ou programas) resultante em linhas de código ou pontos de função;[3] tempo que a equipe irá permanecer reunida (tempo de vida da equipe); até que ponto o problema pode ser modularizado; qualidade e confiabilidade exigidas do sistema a ser construído; rigidez da data de entrega; e grau de sociabilidade (comunicação) exigida para o projeto.

Constantine [Con93] sugere quatro "paradigmas organizacionais" para equipes de engenharia de software:

Quais opções temos ao definir a estrutura de uma equipe de software?

1. O *paradigma fechado* estrutura uma equipe em termos de uma hierarquia de autoridade tradicional. Tais equipes podem trabalhar bem em produção de software bastante similar a esforços já feitos no passado, mas se mostrarão menos propícias a ser inovadoras trabalhando sob o paradigma fechado.
2. O *paradigma randômico* estrutura uma equipe de forma mais livre e depende da iniciativa individual de seus membros. Quando for necessária uma inovação ou um avanço tecnológico, as equipes que seguem o paradigma randômico se destacarão. Mas essas equipes podem brigar quando for exigido um "desempenho ordenado".

Quais fatores devem ser considerados ao se escolher a estrutura de uma equipe de software?

3. O *paradigma aberto* procura estruturar a equipe de maneira que consiga alguns dos controles associados ao paradigma fechado, mas também muito da inovação que ocorre ao se usar o paradigma randômico. O trabalho é feito de forma colaborativa, com forte comunicação e tomada de decisão baseada no consenso – características marcantes das equipes de paradigma aberto. As estruturas das equipes de paradigmas abertos são bem adequadas para a solução de problemas complexos, mas não conseguem desempenhar tão eficientemente quanto outras equipes.
4. O *paradigma sincronizado* baseia-se na compartimentalização natural de um problema e organiza os membros da equipe para trabalhar nas partes do problema com pouca comunicação entre si.

"Se deseja ser incrementalmente melhor, seja competitivo. Se deseja ser exponencialmente melhor, seja cooperativo."

Autor desconhecido

Como um comentário histórico final, uma das mais antigas organizações de equipe de software foi uma estrutura de paradigma fechado, denominada originalmente *equipe com um programador-chefe* (principal). Essa estrutura foi primeiramente proposta por Harlan Mills e descrita por Baker [Bak72]. O núcleo da equipe era composto de um *engenheiro sênior* (o programador-chefe), que planejava, coordenava e fazia a revisão de todas as atividades técnicas da equipe, o *pessoal técnico* (normalmente de duas a cinco pessoas), que conduzia as atividades de análise e de desenvolvimento, e um *engenheiro re-*

[3] Linhas de código (LOC, lines of code) e pontos de função são medidas do tamanho de um programa de computador e são discutidas no Capítulo 33.

serva que dava suporte ao engenheiro sênior em suas atividades e que podia substituí-lo com perdas mínimas de continuidade. O programador-chefe (principal) podia ter a seu dispor um ou mais especialistas (por exemplo, perito em telecomunicações, desenvolvedor de banco de dados), uma equipe de suporte (por exemplo, codificadores técnicos, pessoal de escritório) e um bibliotecário de software.

Como contraponto à estrutura da equipe de programadores-chefe, o paradigma randômico de Constantine [Con93] sugere a primeira equipe criativa, cuja abordagem de trabalho poderia ser mais bem denominada *anarquia inovadora*. Embora a abordagem de espírito livre para o trabalho de software seja atraente, a energia da criatividade direcionada para uma equipe de alta performance deve ser o objetivo central de uma organização de engenharia de software.

CASASEGURA

Estrutura da equipe

Cena: Escritório de Doug Miller antes do início do projeto do software *CasaSegura*.

Atores: Doug Miller (gerente da equipe de engenharia de software do *CasaSegura*) e Vinod Raman, Jamie Lazar e outros membros da equipe.

Conversa:

Doug: Vocês deram uma olhada no informativo preliminar do *CasaSegura* que o departamento de marketing preparou?

Vinod (balançando afirmativamente a cabeça e olhando para seus companheiros de equipe): Sim, mas temos muitas dúvidas.

Doug: Vamos deixar isso de lado por um momento. Gostaria de conversar sobre como vamos estruturar a equipe, quem será responsável pelo quê...

Jamie: Estou totalmente de acordo com a filosofia ágil, Doug. Acho que devemos ser uma equipe auto-organizada.

Vinod: Concordo. Devido ao cronograma apertado e ao grau de incertezas e pelo fato de todos sermos realmente competentes (risos), parece ser o caminho certo a tomar.

Doug: Tudo bem por mim, mas vocês conhecem o procedimento.

Jamie (sorridente e falando ao mesmo tempo): Tomamos decisões táticas sobre quem faz o que e quando, mas é nossa responsabilidade ter o produto pronto sem atraso.

Vinod: E com qualidade.

Doug: Exatamente. Mas lembrem-se de que há restrições. O marketing define os incrementos de software a serem desenvolvidos – consultando-nos, é claro.

Jamie: E...?

6.5 Equipes ágeis

Ao longo da última década, o desenvolvimento de software ágil (Capítulo 5) tem sido indicado como o antídoto para muitos problemas que se alastraram nas atividades de projeto de software. Relembrando, a filosofia ágil enfatiza a satisfação do cliente e a entrega prévia incremental de software, pequenas equipes de projeto altamente motivadas, métodos informais, mínimos artefatos de engenharia de software e total simplicidade de desenvolvimento.

6.5.1 A equipe ágil genérica

A pequena e altamente motivada equipe de projeto, também denominada *equipe ágil*, adota muitas das características das equipes de software bem-sucedidas, discutidas na seção anterior, e evita muito das toxinas geradoras

Uma equipe ágil é auto-organizada e tem autonomia para planejar e tomar decisões técnicas.

de problemas. Entretanto, a filosofia ágil enfatiza a competência individual (membro da equipe) combinada com a colaboração em grupo como fatores críticos de sucesso para a equipe. Cockburn e HighSmith [Coc01a] observam isso ao escreverem:

> Se as pessoas do projeto forem boas o suficiente, podem usar praticamente qualquer processo e cumprir sua missão. Se não forem boas o suficiente, nenhum processo irá reparar a sua inadequação. "Pessoas são o trunfo do processo" é uma forma de dizer isso. Entretanto, falta de suporte ao desenvolvedor e ao usuário pode acabar com um projeto – "política é o trunfo de pessoas". Suporte inadequado pode fazer com que até mesmo os bons fracassem na realização de seus trabalhos.

"Propriedades coletivas nada mais são do que uma instância da ideia de que os produtos deveriam ser atribuídos à equipe (ágil), não a indivíduos que compõem a equipe."

Jim Highsmith

Para uso das competências de cada membro da equipe, para fomentar colaboração efetiva ao longo do projeto, equipes ágeis são auto-organizadas. Uma equipe auto-organizada não mantém, necessariamente, uma estrutura de equipe única, mas usa elementos da aleatoriedade de Constantine, paradigmas abertos e de sincronicidade discutidos na Seção 6.2.

Muitos modelos ágeis de processo (por exemplo, Scrum) dão à equipe ágil autonomia para gerenciar o projeto e tomar as decisões técnicas necessárias à conclusão do trabalho. O planejamento é mantido em um nível mínimo, e a equipe tem a permissão para escolher sua própria abordagem (por exemplo, processo, método, ferramentas), limitada somente pelos requisitos de negócio e pelos padrões organizacionais. Conforme o projeto prossegue, a equipe se auto-organiza, concentrando-se em competências individuais para maior benefício do projeto em um determinado ponto do cronograma. Para tanto, uma equipe ágil pode fazer reuniões de equipe diariamente a fim de coordenar e sincronizar as atividades que devem ser realizadas naquele dia.

Com base na informação obtida durante essas reuniões, a equipe adapta sua abordagem para incrementar o trabalho. A cada dia que passa, auto-organizações contínuas e colaboração conduzem a equipe em direção a um incremento de software completo.

6.5.2 A equipe XP

Simplifique sempre que puder, mas reconheça que uma "refabricação" (retrabalho, redesenvolvimento) contínua pode absorver tempo e recursos significativos.

Beck [Bec04a] define um conjunto de cinco *valores* que estabelecem as bases para todo trabalho realizado como parte da programação extrema (XP) – comunicação, simplicidade, feedback (realimentação ou retorno), coragem e respeito. Cada um desses valores é usado como um direcionamenro para as atividades, ações e tarefas específicas da XP.

Para conseguir a *comunicação* efetiva entre a equipe ágil e outros envolvidos (por exemplo, estabelecer as funcionalidades necessárias para o software), a XP enfatiza a colaboração estreita, embora informal (verbal), entre clientes e desenvolvedores, o estabelecimento de metáforas[4] eficazes para comunicar conceitos importantes, feedback (realimentação) contínuo e evita documentação volumosa como um meio de comunicação.

[4] No contexto da XP, *metáfora* é "uma história que todos – clientes, programadores e gerentes – podem contar sobre como o sistema funciona" [Bec04a].

Para obter a *simplicidade,* a equipe ágil projeta apenas para as necessidades imediatas, em vez de considerar as necessidades futuras. O objetivo é criar um projeto simples que possa ser facilmente implementado em código. Se o projeto tiver que ser melhorado, ele poderá ser *refabricado*[5] mais tarde.

O feedback provém de três fontes: do próprio software implementado, do cliente e de outros membros da equipe de software. Por meio da elaboração do projeto e da implementação de uma estratégia de testes eficaz (Capítulos 22 a 26), o software (via resultados de testes) propicia um feedback para a equipe ágil. A equipe faz uso do *teste unitário* como tática de teste principal. À medida que cada classe é desenvolvida, a equipe desenvolve um teste unitário para testar cada operação de acordo com a funcionalidade especificada. À medida que um incremento é entregue a um cliente, as *histórias de usuários* ou *casos de uso* (Capítulo 9) implementados pelo incremento são utilizados para realizar os testes de aceitação. O grau em que o software implementa o produto, a função e o comportamento do caso em uso é uma forma de feedback. Por fim, conforme novas necessidades surgem como parte do planejamento iterativo, a equipe dá ao cliente um rápido feedback referente ao impacto nos custos e no cronograma.

"A XP é a resposta para a pergunta: 'Qual é o mínimo possível que se pode realizar e mesmo assim desenvolver um software excelente?'."

Anônimo

Beck [Bec04a] afirma que a obediência estrita a certas práticas da XP exige *coragem*. Uma palavra melhor poderia ser *disciplina*. Por exemplo, frequentemente, há uma pressão significativa para a elaboração do projeto pensando em futuros requisitos. A maioria das equipes de software sucumbe, argumentando que "projetar para amanhã" poupará tempo e esforço no longo prazo. Uma equipe XP deve ter disciplina (coragem) para projetar para hoje, reconhecendo que as necessidades futuras podem mudar significativamente, exigindo, consequentemente, um retrabalho substancial em relação ao projeto e ao código implementado.

Ao buscar cada um desses valores, a equipe XP fomenta *respeito* entre seus membros, entre outros envolvidos e os membros da equipe e, indiretamente, para o próprio software. Conforme consegue entregar com sucesso incrementos de software, a equipe desenvolve cada vez mais respeito pelo processo XP.

6.6 O impacto da mídia social

E-mail, mensagens de texto e videoconferência se tornaram atividades onipresentes no trabalho de engenharia de software. Mas, na verdade, esses mecanismos de comunicação nada mais são do que substitutos ou suplementos modernos para o contato face a face. A mídia social é diferente.

[5] A refabricação permite que o engenheiro de software aperfeiçoe a estrutura interna de um projeto (ou código-fonte) sem alterar sua funcionalidade ou comportamento externos. Basicamente, a refabricação pode ser usada para melhorar a eficiência, a legibilidade ou o desempenho de um projeto ou o código que implementa um projeto.

6.11 Pesquise uma das ferramentas de CDE mencionadas no quadro da Seção 6.8 (ou uma ferramenta designada por seu professor) e prepare uma breve apresentação de seus recursos para sua classe.

6.12 Com referência à Figura 6.2, por que a distância complica a comunicação? Por que acentua a necessidade de coordenação? Por que tipos de barreiras e complexidade são introduzidos pela distância?

Leituras e fontes de informação complementares

Embora muitos livros tenham tratado dos aspectos humanos da engenharia de software, dois deles podem ser legitimamente chamados "clássicos". Jerry Weinberg (*The Psychology of Computer Programming,* Silver Anniversary Edition, Dorset House, 1998) foi o primeiro a considerar a psicologia das pessoas que constroem software de computador. Tom DeMarco e Tim Lister (*Peopleware: Productive Projects and Teams,* 2ª ed., Dorset House, 1999) argumentam que os principais desafios no desenvolvimento de software são humanos, não técnicos.

Observações interessantes sobre os aspectos humanos da engenharia de software também foram feitas por Mantle e Lichty (*Managing the Unmanageable: Rules, Tools, and Insights for Managing Software People and Teams*, Addison-Wesley, 2012), Fowler (*The Passionate Programmer,* Pragmatic Bookshelf, 2009), McConnell (*Code Complete,* 2ª ed., Microsoft Press, 2004), Brooks (*The Mythical Man-Month,* 2ª ed., Addison-Wesley, 1999) e Hunt e Thomas (*The Pragmatic Programmer,* Addison-Wesley, 1999). Tomayko e Hazzan *(Human Aspects of Software Engineering,* Charles River Media, 2004) tratam da psicologia e da sociologia da engenharia de software, com ênfase em XP.

Os aspectos humanos do desenvolvimento ágil foram tratados por Rasmussen (*The Agile Samurai,* Pragmatic Bookshelf, 2010) e Davies (*Agile Coaching,* Pragmatic Bookshelf, 2010). Importantes aspectos das equipes ágeis são considerados por Adkins (*Coaching Agile Teams,* Addison-Wesley, 2010) e Derby, Larsen e Schwaber (*Agile Retrospectives: Making Good Teams Great,* Pragmatic Bookshelf, 2006).

A solução de problemas é uma atividade exclusivamente humana e é tratada em livros de Adair (*Decision Making and Problem Solving Strategies,* Kogan Page, 2010), Roam (*Unfolding the Napkin,* Portfolio Trade, 2009) e Wananabe (*Problem Solving 101,* Portfolio Hardcover, 2009).

Diretrizes para facilitar a colaboração dentro de uma equipe de software são apresentadas por Tabaka (*Collaboration Explained,* Addison-Wesley, 2006). Rosen (*The Culture of Collaboration,* Red Ape Publishing, 2009), Hansen (*Collaboration*, Harvard Business School Press, 2009) e Sawyer (*Group Genius: The Creative Power of Collaboration,* Basic Books, 2007) apresentam estratégias e diretrizes práticas para melhorar a colaboração em equipes técnicas.

Promover a inovação humana é o tema de livros de Gray, Brown e Macanufo (*Game Storming,* O-Reilly Media, 2010), Duggan (*Strategic Intuition,* Columbia University Press, 2007) e Hohmann (*Innovation Games,* Addison-Wesley, 2006).

Uma visão geral do desenvolvimento de software global é apresentada por Ebert *(Global Software and IT: A Guide to Distributed Development, Projects, and Outsourcing,* Wiley-IEEE Computer Society Press, 2011). Mite e seus colegas (*Agility Across Time and Space: Implementing Agile Methods in Global Software Projects,* Springer, 2010) editaram uma antologia que trata do uso de equipes ágeis no desenvolvimento global.

Uma ampla variedade de fontes de informação que discutem os aspectos humanos da engenharia de software está disponível na Internet. Uma lista atualizada de referências relevantes (em inglês) para o processo de software pode ser encontrada no site: **www.mhhe.com/pressman**.