

DESENVOLVIMENTO ÁGIL

CONCEITOS-
-CHAVE

agilidade	82
Crystal	97
desenvolvimento de software adaptativo	94
desenvolvimento de software enxuto (LSD)	99
DSDM	96
Extreme Programming – XP (programação extrema)	87
FDD	98
histórias	89
processo ágil	85
processo unificado ágil	101
processo XP	89
programação em duplas	88

Em 2001, Kent Beck e outros dezesseis renomados desenvolvedores, autores e consultores da área de software [Bec01a] (batizados de “Agile Alliance”- “Aliança dos Ágeis”) assinaram o “Manifesto para o Desenvolvimento Ágil de Software” (“Manifesto for Agile Software Development”), que se inicia da seguinte maneira:

Desenvolvendo e ajudando outros a desenvolver software, estamos desvendando 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.

Normalmente, um manifesto é associado a um movimento político emergente: atacando a velha guarda e sugerindo uma mudança revolucionária (espera-se que para melhor). De certa forma, é exatamente do que trata o desenvolvimento ágil.

Embora as ideias básicas que norteiam o desenvolvimento ágil tenham estado conosco por muitos anos, só há menos de duas décadas que se consolidaram como um “movimento”.

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 de incremental prévio; equipes de projeto pequenas e altamente motivadas; métodos informais; artefato de engenharia de software mínimos e, acima de tudo, simplicidade no desenvolvimento geral. Os princípios de desenvolvimento priorizam a entrega mais que análise e 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 finais) 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 a seu serviço).

Por que é importante? O moderno ambiente dos sistemas e dos produtos da área é acelerado e está em constante mudança. A engenharia de software ágil constitui uma razoável alternativa para a engenharia convencional voltada para certas

classes de software e para certos tipos de projetos, e 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 permanecem: comunicação, planejamento, modelagem, construção e emprego. Entretanto, estas 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 como 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 concordar que o processo funciona e essa equipe produz incrementos de software passíveis de entrega e que satisfaçam o cliente, então, o trabalho está correto.

retrabalho	87
scrum	95
velocidade de projeto.	86
XP Industrial. . .	91

Em essência, métodos ágeis¹ 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 consistente e pode ser aplicado como uma filosofia geral para todos os trabalhos de software.

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

Fluidez implica mudanças, e mudanças são caras. Particularmente, se forem sem controle e mal gerenciadas. Uma das características mais convincentes da abordagem ágil é sua habilidade de reduzir os custos da mudança ao longo de todo o processo de software.

Isso significa que o reconhecimento dos desafios apresentados pela moderna realidade faz com que se descartem valiosos princípios da engenharia de software, conceitos, métodos e ferramentas? Absolutamente não! Como todas as disciplinas de engenharia, a engenharia de software continua a evoluir, podendo ser adaptada facilmente aos desafios apresentados pela demanda por agilidade.

Em um texto que nos leva à reflexão sobre desenvolvimento de software ágil, Alistair Cockburn [Coc02] argumenta que o modelo de processo prescritivo, apresentado no Capítulo 2, tem uma falha essencial: esquece das fragilidades 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 consistência 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 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.

“Agilidade: 1,
todo o resto: 0.”

Tom DeMarco

3.1 O QUE É AGILIDADE?

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

Atualmente, *agilidade* tornou-se a palavra da moda quando se descreve um moderno processo de software. Todo mundo é ágil. Uma equipe ágil é aquela rápida e capaz de responder apropriadamente a mudanças. Mudanças têm muito a ver com desenvolvimento de software. Mudanças no software que está sendo criado, mudanças nos membros da equipe, mudanças 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 para mudanças deve ser incorporado em 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.

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



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

Segundo Jacobson, a penetração da mudança é o principal condutor para a agilidade. Os engenheiros de software devem ser rápidos em seus passos caso queiram assimilar as rápidas mudanças que Jacobson descreve.

Porém, agilidade consiste em algo mais que uma resposta à mudança, abrangendo 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); assume o cliente como parte da equipe de desenvolvimento e trabalha para eliminar a atitude de “nós e eles”, que continua a invadir 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 obtê-la, é essencial que seja projetado para que a equipe possa adaptar e alinhar (racionalizar) tarefas; possa conduzir o planejamento compreendendo a fluidez de uma abordagem do 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.

3.2 AGILIDADE E O CUSTO DAS MUDANÇAS

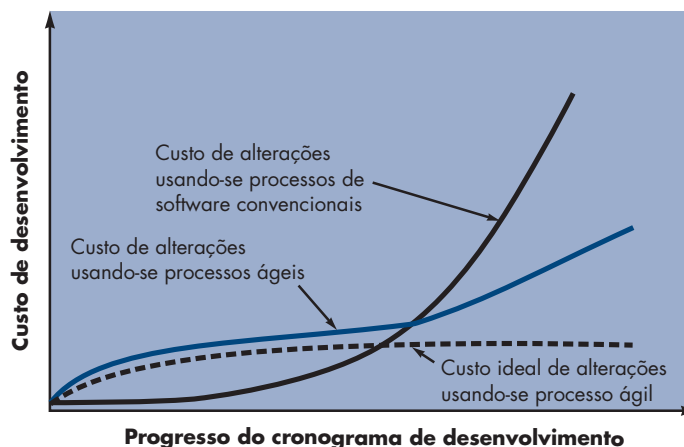
“A agilidade é dinâmica, de conteúdo específico, abrange mudanças agressivas e é orientada ao crescimento.”

Steven Goldman et al.

A sabedoria convencional em desenvolvimento de software (baseada em décadas de experiência) afirma que os custos de mudanças aumentam de forma não linear conforme o projeto avança (Figura 3.1, curva em preto contínuo). É relativamente fácil assimilar uma mudança quando uma equipe de software está juntando requisitos (no início de um projeto). Pode-se ter de alterar um detalhamento do uso, ampliar uma lista de funções ou editar uma especificação por escrito. Os custos de tal trabalho são mínimos e o tempo demandado não afetará negativamente o resultado do projeto. Mas se adiarmos alguns meses, o que aconteceria? A equipe está em meio aos testes de validação (que ocorre relativamente no final do projeto) e um importante interessado está requisitando uma mudança funcional de vulto. A mudança requer uma alteração no projeto da arquitetura do software, o 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 não serão triviais o tempo e custos necessários para assegurar que a mudança seja feita sem efeitos colaterais inesperados.

FIGURA 3.1

Custos de alterações como uma função do tempo em desenvolvimento



PONTO-CHAVE

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.

Os defensores da agilidade (por exemplo, [Bec00], [Amb04]) argumentam que um processo ágil bem elaborado “achata” o custo da curva de mudança (Figura 3.1, curva em linha verde), 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 por pares, (discutida adiante neste capítulo). Há evidências [Coc01a] que sugerem que se pode alcançar redução significativa nos custos de alterações, embora haja um debate contínuo sobre qual o nível em que a curva de custos torna-se “achatada”.

3.3 O QUE É PROCESSO ÁGIL?

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

1. É difícil afirmar antecipadamente quais requisitos de software irão persistir e quais sofrerão alterações. É igualmente difícil prever de que maneira as prioridades do cliente sofrerão alterações conforme o projeto avança.
2. Para muitos tipos de software, o projeto e a construção são “interconduzidos”. Ou seja, ambas as atividades devem ser realizadas em sequência (uma atrás da outra), para que os modelos de projeto sejam provados conforme sejam 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.

WebRef

Uma vasta coleção de artigos sobre processo ágil pode ser encontrada em www.aanpo.org/articles/index.

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

Mas adaptação contínua sem progressos que levem em frente o desenvolvimento realiza muito pouco. Um processo ágil de software deve adaptar *incrementalmente*. Para conseguir uma adaptação incremental, a equipe ágil precisa de feedback do cliente (de modo que as adaptações apropriadas possam ser feitas). Um efetivo catalisador para feedback de 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 de processo feitas para incluir adequadamente o feedback.

PONTO-CHAVE

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

3.3.1 Princípios da agilidade

A Agile Alliance (veja [Agi03], [Fow01]) estabelece 12 princípios de agilidade para quem quiser ter agilidade:

1. A maior prioridade é satisfazer o cliente por meio de entrega adiantada e contínua de software valioso.
2. Acolha bem os pedidos de alterações, mesmo atrasados no desenvolvimento. Os processos ágeis se aproveitam das mudanças como uma vantagem competitiva na relação com o cliente.



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.

3. Entregue software em funcionamento frequentemente, de algumas semanas para alguns meses, dando preferência a intervalos mais curtos.
4. O pessoal comercial e os desenvolvedores devem trabalhar em conjunto diariamente ao longo de todo o projeto.
5. Construa projetos em torno de indivíduos motivados. Dê a eles o ambiente e apoio necessários e confie neles para ter o trabalho feito.
6. O método mais eficiente e efetivo de transmitir informações para e dentro de uma equipe de desenvolvimento é uma conversa aberta, de forma presencial
7. Software em funcionamento é a principal medida de progresso.
8. Os processos ágeis promovem desenvolvimento sustentável. Os proponentes, desenvolvedores e usuários devem estar capacitados para 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 efetuado — é essencial.
11. As melhores arquiteturas, requisitos e projetos emergem de equipes que se auto-organizam.
12. A intervalos regulares, a equipe se avalia para ver como tornar-se 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 relevam) 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.

3.3.2 A política do desenvolvimento ágil

Há debates consideráveis (algumas vezes acirrados) sobre os benefícios e a aplicabilidade do desenvolvimento de software ágil em contraposição a 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”.



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

Como toda argumentação sobre tecnologia de software, o debate sobre metodologia corre o risco de descambar para uma guerra santa. Se for deflagrada uma guerra, a racionalidade desaparece 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, como desenvolver software que atenda às necessidades atuais dos clientes e que apresente características de qualidade que permitirão que seja estendido e ampliado para responder às necessidades dos clientes no longo prazo?

Não há respostas absolutas para nenhuma dessas questões. Mesmo na própria escola ágil, existem vários modelos de processos propostos (Seção 3.4), 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 denegando uma ou outra abordagem.

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

“Muito da agilidade dos métodos deriva do fato de terem suas bases no conhecimento tácito incorporado pela e na equipe, em vez de registrar por escrito tal conhecimento em planejamentos.”

Barry Boehm

? Quais são as características-chave que devem estar presentes entre as pessoas integrantes de uma equipe de software eficiente?

“O que é visto como razoavelmente suficiente por uma equipe pode ser avaliado como mais do que suficiente ou insuficiente por uma outra equipe.”

Alistair Cockburn

PONTO-CHAVE

Uma equipe auto-organizada está no controle do trabalho que realiza. A equipe estabelece seus próprios compromissos e define planos para cumprí-los.

3.3.3 Fatores humanos

Os defensores do desenvolvimento de software ágil se esmeram para enfatizar a importância dos “fatores humanos”. Como afirmam Cockburn e Highsmith [Coc01a], “O desenvolvimento ágil foca talentos e habilidades de indivíduos, moldando o processo de acordo com as pessoas e as equipes específicas”. O ponto-chave nessa afirmação é que *o processo se amolda às necessidades das pessoas e equipes*, e não o caminho inverso.²

Se os membros da equipe de software devem orientar as características do processo que é aplicado para construir software, deve existir um certo número de traços-chave entre as pessoas de uma equipe ágil e a equipe em si:

Competência. No contexto do desenvolvimento ágil (assim como no da engenharia de software), a “competência” abrange talento inato, habilidades específicas relacionadas a software e conhecimento generalizado do processo que a equipe escolheu para aplicar. Habilidade e conhecimento de processo podem e devem ser ensinados para todas as pessoas que sejam membros de uma equipe ágil.

Foco comum. Embora os membros de uma equipe ágil possam realizar diferentes tarefas e tragam diferentes habilidades para o projeto, todos devem estar focados em um único objetivo — entregar um incremento de software funcionando ao cliente, dentro do prazo prometido. Para alcançar essa meta, a equipe também irá focar em adaptações contínuas (pequenas e grandes) que farão com que o processo se ajuste às necessidades da equipe.

Colaboração. Engenharia de software (independentemente do processo) trata de avaliação, análise e uso de informações comunicadas à equipe de software; criar informações que ajudarão todos os envolvidos a compreender o trabalho da equipe e a construir informações (software para computadores e bancos de dados relevantes) que forneçam valor de negócio para o cliente. Para realizar essas tarefas, os membros da equipe devem colaborar — entre si e com todos os demais envolvidos.

Habilidade na tomada de decisão. Qualquer boa equipe de software (até mesmo as equipes ágeis) deve ter liberdade para controlar seu próprio destino. Isso implica que seja dada autonomia à equipe — autoridade na tomada de decisão, tanto em assuntos técnicos como de projeto.

Habilidade de solução de problemas confusos. Os gerentes de software devem reconhecer que a equipe ágil terá de lidar continuamente com a ambiguidade e que será continuamente atingida por mudanças. Em alguns casos, a equipe tem de aceitar o fato de que o problema que eles estão solucionando hoje talvez não seja o problema que necessita ser solucionado amanhã. Entretanto, lições aprendidas de qualquer atividade de solução de problemas (inclusive aquelas que resolvem o problema errado) podem ser, futuramente, benéficas para a equipe no projeto.

Confiança mútua e respeito. A equipe ágil deve tornar-se uma equipe tal qual a que DeMarco e Lister [DeM98] denominam de equipe “consistente” (Capítulo 24). Uma equipe consistente demonstra a confiança e o respeito necessários para torná-la “tão fortemente unida que o todo fica maior do que a soma das partes”. [DeM98]

Auto-organização. No contexto do desenvolvimento ágil, a auto-organização implica três fatores: (1) a equipe ágil se organiza para o trabalho a ser feito, (2) a equipe organiza o processo para melhor se adequar ao seu ambiente local, (3) a equipe organiza o cronograma de trabalho para melhor cumprir a entrega do incremento de software. A auto-organização possui uma série de benefícios técnicos, porém, mais importante, é o fato de servir para melhorar a colaboração e levantar o moral da equipe. Em essência, a equipe faz seu próprio

² As organizações de engenharia de software bem-sucedidas reconhecem essa realidade independentemente do modelo de processos por elas escolhido.

gerenciamento. Ken Schwaber [Sch02] menciona tais características ao escrever: “A equipe seleciona quanto trabalho acredita ser capaz de realizar dentro da iteração e se compromete com trabalho. Nada desmotiva tanto uma equipe como um terceiro assumir compromissos por ela. Nada motiva tanto uma equipe quanto aceitar a responsabilidade de cumprir completamente o prometido feito por ela própria”.

3.4 EXTREME PROGRAMMING – XP (PROGRAMAÇÃO EXTREMA)

Para ilustrar um processo ágil de forma um pouco mais detalhada, segue uma visão geral de *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 durante o final dos anos 1980, o trabalho seminal sobre o tema foi escrito por Kent Beck [Bec04a]. Mais recentemente, foi proposta uma variação da XP, denominada *Industrial XP (IXP)* [Ker05]. A IXP refina a XP e visa o processo ágil especificamente para uso em grandes organizações.

3.4.1 Valores da XP

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

Para conseguir a *comunicação* efetiva entre engenheiros de software e outros envolvidos (por exemplo, estabelecer os fatores e funções necessárias para o software), a XP enfatiza a colaboração estreita, embora informal (verbal), entre clientes e desenvolvedores, o estabelecimento de metáforas eficazes³ para comunicar conceitos importantes, feedback (realimentação) contínuo e evitar documentação volumosa como meio de comunicação.

Para alcançar a *simplicidade*, a XP restringe os desenvolvedores a projetar apenas para as necessidades imediatas, em vez de considerarem as necessidades futuras. O intuito é criar um projeto simples que possa ser facilmente implementado em código. Se o projeto tiver que ser melhorado, ele poderá ser *refabricado*⁴ mais tarde.

O *feedback* provém de três fontes: do próprio software implementado, do cliente e de outros membros da equipe de software. Através da elaboração do projeto e da implementação de uma estratégia de testes eficaz (Capítulos 17 a 20), o software (via resultados de testes) propicia um feedback para a equipe ágil. A XP faz uso do *teste de unidades* como sua tática de testes primária. À medida que cada classe é desenvolvida, a equipe desenvolve um teste de unidades para exercitar cada operação de acordo com sua funcionalidade especificada. À medida que um incremento é entregue a um cliente, as *histórias de usuários* ou *casos de uso* (Capítulo 5) implementados pelo incremento são usados como base para 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.

Beck [Bec04a] afirma que a adoção 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 ágil deve ter disciplina (coragem) para projetar para hoje, reconhecendo que as



AVISO

Simplifique sempre que puder, mas tenha ciência de que um “retrabalho” (refabricação, redesenvolvimento) contínuo consegue absorver tempo e recursos significativos.

“A XP é a resposta para a pergunta: ‘Qual o mínimo possível que se pode realizar e mesmo assim desenvolver um software grandioso?’.”

Anônimo

³ No contexto da XP, uma *metáfora* é “uma história que todos — clientes, programadores e gerentes — podem contar sobre como o sistema funciona” [Bec04a].

⁴ A refabricação permite a um engenheiro de software aperfeiçoar a estrutura interna de um projeto (ou código-fonte) sem alterar sua funcionalidade ou comportamento externos. Em essência, a refabricação pode ser usada para melhorar a eficiência, a legibilidade ou o desempenho de um projeto ou o código que implementa um projeto.

necessidades futuras podem mudar dramaticamente exigindo, conseqüentemente, substancial retrabalho em relação ao projeto e ao código implementado.

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

3.4.2 O Processo XP

WebRef

Uma excelente visão geral das “regras” para XP pode ser encontrada em www.extremeprogramming.org/rules.html.

A *Extreme Programming* (programação extrema) emprega uma abordagem orientada a objetos (Apêndice 2) como seu paradigma de desenvolvimento preferido e envolve um conjunto de regras e práticas constantes no contexto de quatro atividades metodológicas: planejamento, projeto, codificação e testes. A Figura 3.2 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.

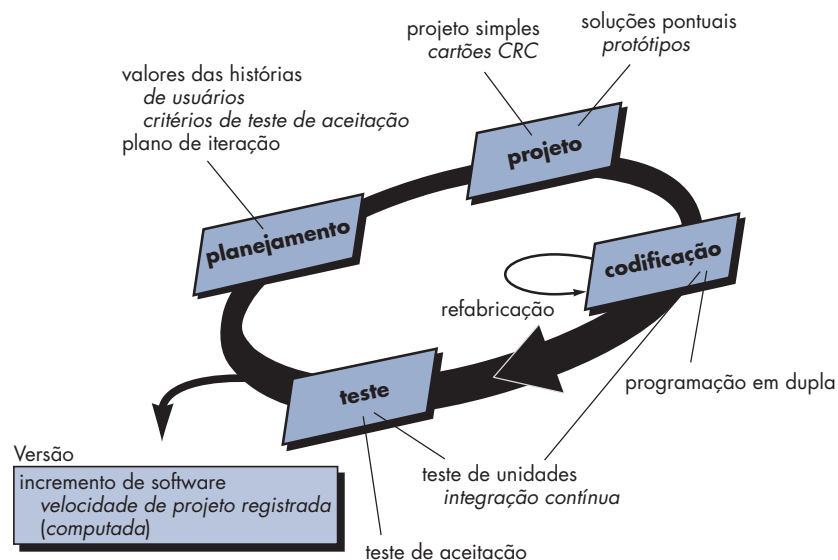
Planejamento. A atividade de planejamento (também denominada *o jogo do planejamento*) se inicia com a atividade de *ouvir* — uma atividade de levantamento de requisitos que capacita os membros técnicos da equipe XP a entender o ambiente de negócios do software e possibilita que se consiga ter uma percepção ampla sobre os resultados solicitados, fatores principais e funcionalidade

A atividade de “Ouvir” conduz à criação de um conjunto de “histórias” (também denominado *histórias de usuários*) que descreve o resultado, as características e a funcionalidade requisitados para o software a ser construído. Cada *história* (similar aos casos de uso descritos no Capítulo 5) 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.⁵ Os membros da equipe XP avaliam então cada história e atribuem um *custo* — medido em semanas de desenvolvimento — a ela. Se a história requerer, por estimativa, mais do que três semanas de desenvolvimento, é solicitado ao cliente para dividir a história em histórias menores e a atribuição de valor e custo ocorre novamente. É importante notar que podem ser escritas novas histórias a qualquer momento.

? O que é uma “história” XP?

FIGURA 3.2

O processo da Extreme Programming (XP)



5 O valor de uma história também pode depender da presença de uma outra história.

WebRef

Um “jogo de planejamento” XP bastante interessante pode ser encontrado em: c2.com/cgi/wiki?planningGame.

PONTO-CHAVE

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



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

WebRef

Técnicas de refabricação e ferramentas podem ser encontrados em: www.refactoring.com.

PONTO-CHAVE

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

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. Consequindo 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 das 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.

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.

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

Projeto. O projeto XP segue rigorosamente o princípio KIS (*keep it simple*, ou seja, preserve a simplicidade). É preferível sempre um projeto simples do que 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 desencorajado.⁶

A XP encoraja o uso de cartões CRC (Capítulo 7) como um mecanismo eficaz para pensar sobre o software em um contexto orientado a objetos. Os cartões CRC (classe-responsabilidade-colaborador) identificam e organizam as classes orientadas a objetos⁷ relevantes para o incremento de software corrente. A equipe XP conduz o exercício de projeto usando um processo similar ao descrito no Capítulo 8. Os cartões CRC são o único artefato de projeto produzidos como parte do processo XP.

Se um difícil problema de projeto for encontrado 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.

Na seção anterior, foi feita a observação de que a XP encoraja a *refatoração* — uma técnica de construção que também é um método para otimização de projetos. Fowler [Fow00] descreve a refabricação da seguinte maneira:

Refabricação é o processo de alteração de um sistema de software de tal forma que não se altere o comportamento externo do código, mas se aprimore a estrutura interna. 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 refabricar, se está aperfeiçoando o projeto de codificação depois de este ter sido feito.

Como o projeto XP não usa praticamente nenhuma notação e produz poucos, se algum, artefatos, 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 refabricação é controlar tais modificações sugerindo pequenas mudanças de projeto “capazes de melhorá-lo radicalmente” [Fow00]. Deve ser observado, no entanto, que o esforço

⁶ Tais diretrizes de projeto deveriam ser seguidas em todos os métodos de engenharia de software, apesar de ocorrer situações em que sofisticadas terminologia e notação possam constituir obstáculo para a simplicidade.

⁷ As classes orientadas a objetos são discutidas no Apêndice 2, no Capítulo 8 e ao longo da Parte 2 deste livro.

necessário para a refabricação pode aumentar dramaticamente 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 *como depois* de se ter iniciado a codificação. Refabricação 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 à aprimoração do projeto.

WebRef

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

? O que é programação em dupla?



Muitas equipes de software são constituídas por individualistas. Deverá haver empenho para modificar tal cultura, para que a programação em dupla funcione efetivamente.

? Como são usados os testes de unidade na XP?

PONTO-CHAVE

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

Codificação. Depois de desenvolvidas as histórias e 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 incluídas na versão corrente (incremento de software).⁸ Uma vez criado o teste de unidades⁹, o desenvolvedor poderá melhor focar-se no que deve ser implementado para ser aprovado no teste. Nada estranho é adicionado (KIS). Estando o código completo, este pode ser testado em unidade imediatamente, e, dessa forma, prover, instantaneamente, feedback para os desenvolvedores.

Um conceito-chave na atividade de codificação (e um dos mais discutidos aspectos da XP) é a *programação em dupla*. 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 resolução 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 focados no problema em questão. Na prática, cada pessoa assume um papel ligeiramente diferente. Por exemplo, uma pessoa poderia pensar nos detalhes de 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.

Conforme a dupla de programadores completa 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 interfaceamento, além de criar um ambiente “teste da fumaça” (Capítulo 17) que ajuda a revelar erros precocemente.

Testes. Já foi observado que a criação de testes de unidade, antes de começar a codificação, é um elemento-chave da abordagem XP. Os testes de unidade criados devem ser implementados usando-se uma metodologia que os capacite a ser automatizados (assim, poderão ser executados fácil e repetidamente). Isso encoraja uma estratégia de testes de regressão (Capítulo 17), toda vez em que o código for modificado (o que é frequente, dada a filosofia de refabricação 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, 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 de software.

⁸ Essa abordagem é como conhecer 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.

⁹ O teste de unidades, discutido detalhadamente no Capítulo 17, 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.

3.4.3 Industrial XP

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 espírito minimalista, centrado no cliente e orientado a testes da XP. Difere principalmente da XP original 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 assegurar que um projeto XP funcione com êxito em empreendimentos significativos em uma grande organização.

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

“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. Antes do início de um projeto IXP, a organização deve realizar uma *avaliação imediata*. A avaliação verifica se (1) existe um ambiente de desenvolvimento apropriado para sustentar a IXP, (2) a equipe será composta por um conjunto apropriado de interessados, (3) a organização possui um programa de qualidade diferenciado e suporta contínuo aperfeiçoamento, (4) a cultura organizacional apoiará os novos valores de uma equipe ágil e (5) a comunidade de projeto ampliada será composta apropriadamente.

Comunidade de projeto. A XP clássica sugere que se aloquem as pessoas acertadas para compor a equipe ágil e garantir o sucesso. Isso implica pessoas da equipe bem treinadas, adaptáveis e experientes e que tenham temperamento apropriado para contribuir para uma equipe auto-organizada. Ao se aplicar a XP em um projeto importante de uma grande empresa, o conceito da “equipe” deve transformar-se no de *comunidade*. A comunidade pode ter um tecnólogo e clientes fundamentais para o sucesso de um projeto, assim como muitos outros envolvidos (por exemplo, responsáveis jurídicos, auditores do controle da qualidade, representantes da área de produção ou de categorias de vendas) que “frequentemente se encontram na periferia de um projeto IXP, mas que podem desempenhar importante papel no projeto” [Ker05]. Na IXP, os membros da comunidade devem ter papéis explicitamente definidos e os mecanismos de comunicação e de coordenação relativos aos elementos da comunidade devem estar determinados.

Mapeamento do projeto. A própria equipe IXP avalia o projeto para determinar se este se justifica em termos de negócios e se irá ultrapassar as metas e objetivos globais da organização. O mapeamento também examina o contexto do projeto para estabelecer como este complementa, amplia ou substitui sistemas ou processos existentes.

Gerenciamento orientado a testes. Um projeto IXP requer critérios mensuráveis para avaliar o estado do projeto e do progresso obtido até então. O gerenciamento orientado a testes estabelece uma série de “destinos” mensuráveis [Ker05] e define mecanismos para determinar se estes foram atingidos ou não.

Retrospectivas. Uma equipe IXP conduz uma revisão técnica especializada (Capítulo 15) após a entrega de um incremento de software. Denominada *retrospectiva*, a revisão examina “itens, eventos e lições aprendidas” [Ker05] ao longo do processo de incremento de software e/ou do desenvolvimento da versão completa do software. O objetivo é aprimorar o processo da IXP.

Aprendizagem contínua. Sendo a aprendizagem uma parte vital para o aperfeiçoamento contínuo do processo, os membros da equipe XP são encorajados (e possivelmente incentivados) a aprender novos métodos e técnicas que possam conduzir a um produto de melhor qualidade.


Somando-se às apresentadas, a IXP modifica uma série de práticas XP existentes. O desenvolvimento orientado por histórias (*story-driven development*, *SDD*) insiste que as histórias para testes de aceitação sejam escritas antes de gerar uma única linha de código. O projeto orientado por domínio (*domain-driven design*, *DDD*) é um aprimoramento do conceito “metáfora de sistema” usado na XP. O DDD [Eva03] sugere a criação evolucionária de um modelo de domínio que “represente acuradamente como pensam os especialistas de determinado domínio dentro de sua disciplina” [Ker05]. O *emparelhamento* amplia o conceito de programação em dupla da

XP, ao incluir gerentes e outros envolvidos. O intuito é ampliar o compartilhamento de conhecimentos entre os membros da equipe XP que possam não estar diretamente envolvidos no desenvolvimento técnico. A *usabilidade iterativa* desencoraja o projeto de interfaces de carregamento frontal (*front-loaded interface design*), sendo a favor do projeto de usabilidade que evolui conforme os incrementos sejam entregues e a interação entre usuários e o software seja estudada.

A IXP faz modificações menores para outras práticas XP e redefine certos papéis e responsabilidades para torná-los mais harmonizados com projetos importantes de organizações. Para uma discussão mais ampla sobre a IXP, visite <http://industrialxp.org>.

3.4.4 O Debate XP

Todos os novos métodos e modelos de processos estimulam debates úteis e, em alguns casos, debates acalorados. A *Extreme Programming* provocou ambos. Em um livro interessante que examina a eficácia da XP, Stephens e Rosenberg [Ste03] argumentam que muitas práticas XP valem a pena, mas outras foram superestimadas e algumas poucas são problemáticas. Os autores sugerem que a codependência da prática da XP representa sua força e sua fraqueza. Pelo fato de muitas organizações adotarem apenas um subconjunto de práticas XP, elas enfraquecem a eficácia de todo o processo. Seus defensores rebatem dizendo que a XP é aperfeiçoada continuamente e que muitos dos itens levantados pela crítica têm sido acessados conforme a prática da XP ganha maturidade. Entre os itens que continuam a incomodar certos críticos da XP estão:¹⁰

 **Quais são alguns dos pontos que conduzem a um debate a respeito da XP?**

- *Volatilidade de requisitos.* Pelo fato de o cliente ser um membro ativo da equipe XP, alterações de requisitos são solicitadas informalmente. Como consequência, o escopo do projeto pode mudar e trabalhos anteriores podem ter de vir a ser alterados, a fim de acomodar as necessidades de então. Seus defensores argumentam que isso acontece independentemente do processo aplicado e que a XP oferece mecanismos para controlar o surgimento incontrolado de novos escopos.
- *Necessidades conflitantes de clientes.* Projetos em quantidade possuem múltiplos clientes, cada um com seu próprio conjunto de necessidades. Na XP, a própria equipe tem a tarefa de assimilar as necessidades de diferentes clientes, um trabalho que pode estar além de seu escopo de autoridade.
- *Os requisitos são levantados informalmente.* Histórias de usuários e testes de aceitação são a única manifestação explícita de requisitos da XP. Seus críticos argumentam que, frequentemente, torna-se necessário um modelo ou especificação mais formal para assegurar que omissões, inconsistências e erros sejam descobertos antes que o sistema seja construído. Seus defensores rebatem dizendo que a natureza mutante de requisitos torna tais modelos e especificações obsoletos praticamente logo depois de terem sido desenvolvidos.
- *Falta de projeto formal.* A XP tira a ênfase da necessidade do projeto de arquitetura e, em muitos casos, sugere que todos os tipos de projetos devam ser relativamente informais. Seus críticos argumentam que em sistemas complexos deve-se enfatizar a elaboração do projeto para assegurar que a estrutura geral do software apresentará qualidade e facilidade de manutenção. Já os defensores da XP sugerem que a natureza incremental do processo XP limita a complexidade (a simplicidade é um valor fundamental) e, consequentemente, reduz a necessidade de um projeto extenso.

Deve-se observar que todo processo de software tem suas falhas e que muitas organizações de software usaram, com êxito, a XP. O segredo é reconhecer onde um processo pode apresentar fraquezas e adaptá-lo às necessidades específicas de sua organização.

¹⁰ Para uma visão detalhada de algumas críticas ponderadas feitas ao XP, visite www.softwarereality.com/ExtremeProgramming.jsp.

CASA SEGURA



Considerando o desenvolvimento de software ágil

Cena: 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 adentram à sala de Doug)

Jamie: Doug, você tem um minuto?

Doug: Com certeza, Jamie, o que há?

Jamie: Estivemos pensando a respeito da discussão sobre processos, de ontem... Sabe, que processo vamos escolher para este novo projeto *CasaSegura*.

Doug: E?

Vinod: Eu estava conversando com um amigo de uma outra empresa e ele me falou sobre a 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 realmente rápido, usa algo chamado programação em dupla para fazer checagens 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 dupla, por exemplo, e da ideia de que os envolvidos devam fazer parte da equipe.

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

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

Jamie: Jesus... Eles solicitarão 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: Portanto, meus amigos, vocês acham que deveríamos usar a XP?

Jamie: Definitivamente, vale considerar.

Doug: Eu concordo. E mesmo que optássemos por um modelo incremental como nossa abordagem, não há nenhuma razão para não podermos incorporar muito do que a XP tem a oferecer.

Vinod: Doug, mas antes você disse “algumas coisas boas, outras ruins”. O que são as “coisas ruins”?

Doug: O que não me agrada é a maneira pela qual a XP dá menos importância à análise e ao projeto... Dizem algo como: a codificação resume a ação para construir um software.

(Os membros da equipe se entreolham e sorriem.)

Doug: Então vocês concordam com a abordagem XP?

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

Doug (rindo): É verdade, mas eu gostaria de vê-lo 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.

3.5 OUTROS MODELOS DE PROCESSOS ÁGEIS

“Nossa profissão passa por metodologias como uma garota de 14 anos passa por roupas.”

Stephen Hawrysh e Jim Ruprecht

Na história da engenharia de software há dezenas de metodologias e descrições de processos, métodos e notações de modelagem, ferramentas e tecnologias obsoletas. Cada um atingiu grande notoriedade e foi então ofuscado por algo novo e (supostamente) melhor. Com a introdução de uma ampla gama de modelos de processos ágeis — todos disputando por aceitação pela comunidade de desenvolvimento de software —, o movimento ágil está seguindo o mesmo caminho histórico.¹¹

Conforme citado na última seção, o modelo mais amplamente utilizado de todos os modelos de processos ágeis é o da Extreme Programming (XP). Porém, muitos outros têm sido propostos e encontram-se em uso no setor. Entre os mais comuns, temos:

- Desenvolvimento de software adaptativo (Adaptive Software Development, ASD)
- Scrum
- Método de desenvolvimento de sistemas dinâmicos (Dynamic Systems Development Method, DSDM)
- Crystal

¹¹ Isso não é algo ruim. Antes que um ou mais modelos ou métodos sejam aceitos como um padrão *de fato*, todos devem competir para conquistar os corações e mentes dos engenheiros de software. Os “vencedores” evoluem e se transformam nas melhores práticas, enquanto os “perdedores” desaparecem ou se fundem aos modelos vencedores.

- Desenvolvimento dirigido a Funcionalidades (Feature Drive Development, FDD)
- Desenvolvimento de software enxuto (Lean Software Development, LSD)
- Modelagem ágil (Agile Modeling, AM)
- Processo unificado ágil (Agile Unified Process, AUP)

Nas seções seguintes, apresenta-se uma visão geral muito breve de cada um desses modelos de processos ágeis. É importante observar que *todos* estão em conformidade (em maior ou menor grau) com o *Manifesto for Agile Software Development* e com os princípios citados na Seção 3.3.1. Para mais detalhes, veja as referências citadas em cada subseção ou, para uma pesquisa, examine a entrada “agile software development” na Wikipedia.¹²

3.5.1 Desenvolvimento de Software Adaptativo (ASD)

WebRef

Recursos úteis para ASD podem ser encontrados em www.adaptivesd.com.

O desenvolvimento de software adaptativo (*Adaptive Software Development*) foi proposto por Jim Highsmith [Hig00] como uma técnica para construção de software e sistemas complexos. As bases filosóficas do ASD se concentram na colaboração humana e na auto-organização das equipes.

Highsmith argumenta que uma abordagem de desenvolvimento ágil e adaptativo baseada na colaboração constitui “um recurso para organizar nossas complexas interações, tanto quanto disciplina e engenharia o são”. Ele define um “ciclo de vida” ASD (Figura 3.3) que incorpora três fases: especulação, colaboração e aprendizagem.

Durante a *especulação*, o projeto é iniciado e conduzido o *planejamento de ciclos adaptáveis*. O *planejamento de ciclos adaptáveis* usa as informações do início de projeto — o estabelecimento da missão do cliente, as restrições do projeto (por exemplo, datas de entrega ou descrições de usuários) e os requisitos básicos — para definir o conjunto de ciclos de versão (incrementos de software) que serão requisitados para o projeto.

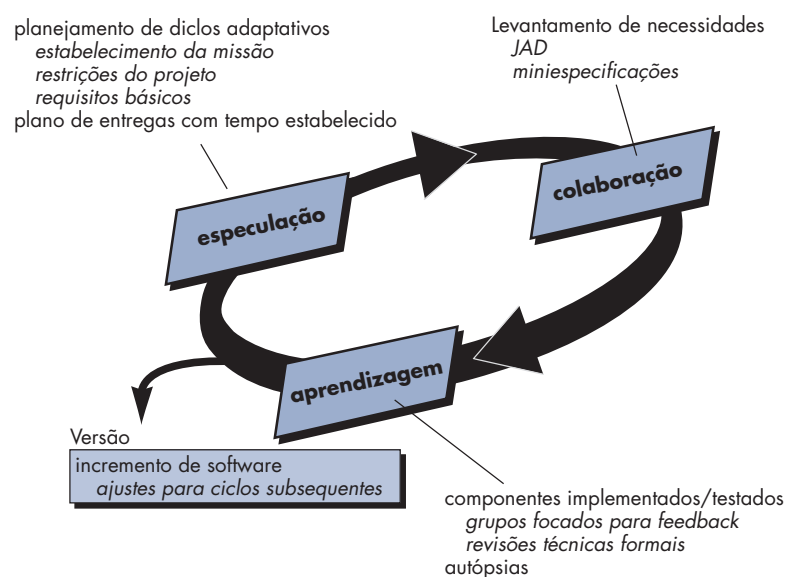
Não importa quão completo e com visão de futuro seja o plano de ciclos, invariavelmente sofrerá mudanças. Baseando-se nas informações obtidas ao se completar o primeiro ciclo, o plano é revisto e ajustado de modo que o trabalho planejado melhor se ajuste à realidade na qual a equipe ASD está trabalhando.



A colaboração efetiva com seu cliente ocorrerá somente se você extinguir quaisquer atitudes de “nós e eles”.

FIGURA 3.3

Desenvolvimento de software adaptável



¹² Veja http://en.wikipedia.org/wiki/Agile_software_development#Agile_methods.

As pessoas motivadas usam a *colaboração* de uma forma que multiplique seus talentos e produções criativas além de seus números absolutos. Tal abordagem é tema recorrente em todos os métodos ágeis. Porém, colaboração não é algo fácil, envolve comunicação e trabalho em equipe, mas também enfatiza o individualismo, pois a criatividade individual desempenha um importante papel no pensamento colaborativo. Trata-se, sobretudo, de uma questão de confiança. Pessoas que trabalham juntas têm de confiar umas nas outras para (1) criticar sem animosidade, (2) auxiliar sem ressentimentos, (3) trabalhar tão arduamente ou mais do que elas fazem, (4) possuir o conjunto de habilidades que contribua com o atual trabalho e (5) comunicar problemas ou preocupações de forma que conduzam a ações efetivas.

PONTO-CHAVE

O ASD enfatiza o aprendizado como elemento-chave para conseguir uma equipe “auto-organizada”.

Conforme os membros de uma equipe ASD comecem a desenvolver os componentes que fazem parte de um ciclo adaptável, a ênfase reside no “aprendizado” tanto quanto reside no progresso para um ciclo completado. De fato, Highsmith [Hig00] argumenta que os desenvolvedores de software normalmente superestimam seu próprio entendimento (da tecnologia, do processo e do projeto) e que a aprendizagem irá ajudá-los a aumentar seus níveis reais de entendimento. As equipes ASD aprendem de três maneiras: grupos focados (Capítulo 5), revisões técnicas (Capítulo 14) e autópsias de projetos (análises *postmortems*).

A filosofia ASD tem seus méritos independentemente do modelo de processos utilizado. A ênfase global da ASD está na dinâmica das equipes auto-organizadas, na colaboração interpessoal e na aprendizagem individual e da equipe que levam as equipes de projeto de software a uma probabilidade muito maior de sucesso.

3.5.2 Scrum

Scrum (o nome provém de uma atividade que ocorre durante a partida de rugby¹³) é um método de desenvolvimento ágil de software concebido por Jeff Sutherland e sua equipe de desenvolvimento no início dos anos 1990. Mais recentemente, foram realizados desenvolvimentos adicionais nos métodos gráficos Scrum por Schwaber e Beedle [Sch01a].

Os princípios do Scrum são consistentes com o manifesto ágil e são usados para orientar as atividades de desenvolvimento dentro de um processo que incorpora as seguintes atividades estruturais: requisitos, análise, projeto, evolução e entrega. Em cada atividade metodológica, ocorrem tarefas a realizar 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 é ilustrado na Figura 3.4.

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 críticos de negócio. Cada um desses padrões de processos define um conjunto de ações de desenvolvimento:

Registro pendente de trabalhos (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 em qualquer momento (é assim que as alterações são introduzidas). O gerente de produto avalia o registro e atualiza as prioridades conforme requisitado.

Urgências (corridas de curta distância) sprints — consistem de 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)¹⁴ (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.

WebRef

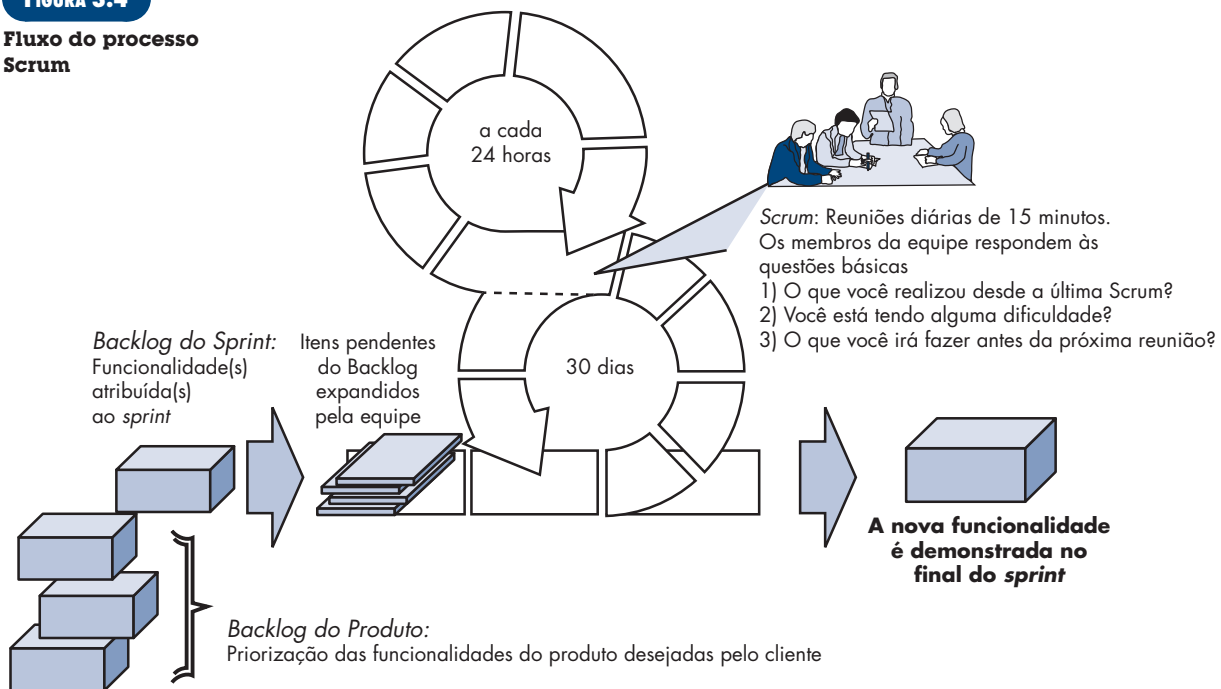
Informações e recursos úteis sobre o Scrum podem ser encontrados em www.controlchaos.com.

PONTO-CHAVE

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.

¹³ 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.

¹⁴ *Janela de tempo (time boxing)* é um termo de gerenciamento de projetos (veja a Parte 4 deste livro) que indica um período de tempo destinado para cumprir alguma tarefa.

FIGURA 3.4**Fluxo do processo Scrum**

Reuniões Scrum — são reuniões curtas (tipicamente 15 minutos), realizadas diariamente pela equipe Scrum. São feitas três perguntas-chave e 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 potenciais 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 pelo cliente. 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 a eliminação da incerteza é impossível.

3.5.3 Método de Desenvolvimento de Sistemas Dinâmicos (DSDM)

O *método de desenvolvimento de sistemas dinâmicos* (Dynamic Systems Development Method) [Sta97] é uma abordagem de desenvolvimento de software ágil que “oferece uma metodologia para construir e manter sistemas que atendem restrições de prazo apertado através do uso da prototipagem 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%).

WebRef

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

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 remanescentes podem ser completados depois, quando outros requisitos de negócio forem conhecidos ou alterações tiverem sido solicitadas e acomodadas.

O DSDM Consortium (www.dsdm.org) é um grupo mundial de empresas-membro que coletivamente assume o papel de “mantenedor” do método. Esse consórcio definiu um modelo de processos ágeis, chamado *ciclo de vida DSDM* que define três ciclos iterativos diferentes, precedidos por duas atividades de ciclo de vida adicionais:

Estudo da viabilidade — estabelece os requisitos básicos de negócio e restrições associados à aplicação a ser construída e depois avalia se a aplicação é ou não um candidato viável para o processo DSDM.

Estudo do negócio — estabelece os requisitos funcionais e de informação que permitirão à aplicação agregar valor de negócio; define também a arquitetura básica da aplicação e identifica os requisitos de facilidade de manutenção para a aplicação.

Iteração de modelos funcionais — produz um conjunto de protótipos incrementais que demonstram funcionalidade para o cliente. (Nota: 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 juntar requisitos adicionais ao se obter feedback dos usuários, conforme testam o protótipo.

Iteração de projeto e desenvolvimento — revisita 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 finais, 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 — aloca 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 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 3.4) para fornecer uma abordagem combinatória que define um modelo de processos consistente (o ciclo de vida do DSDM) com as práticas básicas (XP) necessárias para construir incrementos de software. Além disso, os conceitos de colaboração e de equipes auto-organizadas do ASD podem ser adaptados a um modelo de processos combinado.

3.5.4 Crystal

Alistair Cockburn [Coc05] e Jim Highsmith [Hig02b] criaram a *família Crystal de métodos ágeis*¹⁵ visando conseguir elaborar uma abordagem de desenvolvimento de software que priorizasse a adaptabilidade (“maneuverability”) durante o que Cockburn caracteriza como um “jogo de invenção e comunicação cooperativo e com recursos limitados, tendo como primeiro objetivo entregar software útil em funcionamento e como segundo objetivo preparar-se para o jogo seguinte” [Coc02].

Para conseguir adaptabilidade, Cockburn e Highsmith definiram um conjunto de metodologias com elementos essenciais comuns a todas, mas com papéis, padrões de processos, produtos de trabalho e prática únicos para cada uma delas. A família Crystal é, na verdade, um conjunto de exemplos de processos ágeis que provaram ser efetivos para diferentes tipos de projetos. A intenção é possibilitar que equipes ágeis selecionem o membro da família Crystal mais apropriado para seu projeto e seu ambiente.

PONTO-CHAVE

Crystal é uma família de modelos de processos com o mesmo “código genético”, mas com diferentes métodos para se adaptarem às características do projeto.

PONTO-CHAVE

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

15 O nome “crystal” (cristal) é derivado das características dos cristais geológicos, cada qual com sua cor, forma e dureza próprias.

3.5.5 Desenvolvimento Dirigido a Funcionalidades (FDD)

O desenvolvimento dirigido a funcionalidades (Feature Driven Development) foi concebido originalmente por Peter Coad e seus colegas [Coa99] como um modelo de processos prático para a engenharia de software orientada a objetos. Stephen Palmer e John Felsing [Pal02] estenderam e aperfeiçoaram o trabalho de Coad, descrevendo um processo ágil adaptativo que pode ser aplicado a projetos de software de porte moderado e a projetos maiores.

WebRef

Uma ampla variedade de artigos e apresentações sobre o FDD pode ser encontrada em: www.featuredrivendevelopment.com/.

Como outras abordagens ágeis, o FDD adota uma filosofia que (1) enfatiza a colaboração entre pessoas da equipe FDD; (2) gerencia problemas e complexidade de projetos utilizando a decomposição baseada em funcionalidades, seguida pela integração dos incrementos de software, e (3) comunicação de detalhes técnicos usando meios verbais, gráficos e de texto. O FDD enfatiza as atividades de garantia da qualidade de software por meio do encorajamento de uma estratégia de desenvolvimento incremental, o uso inspeções do código e do projeto, a aplicação de auditorias para garantia da qualidade de software (Capítulo 16), a coleta de métricas e o uso de padrões (para análise, projeto e construção).

No contexto do FDD, funcionalidade “é uma função valorizada pelo cliente passível de ser implementada em duas semanas ou menos” [Coa99]. A ênfase na definição de funcionalidades gera os seguintes benefícios:

- Como as funcionalidades formam pequenos blocos que podem ser entregues, os usuários podem descrevê-las mais facilmente; compreender como se relacionam entre si mais prontamente; e revisá-las melhor em termos de ambiguidade, erros ou omissões.
- As funcionalidades podem ser organizadas em um agrupamento hierárquico relacionado com o negócio.
- Como uma funcionalidade é o incremento de software do FDD que pode ser entregue, a equipe desenvolve funcionalidades operacionais a cada duas semanas.
- Pelo fato de o bloco de funcionalidades ser pequeno, seus projeto e representações de código são mais fáceis de inspecionar efetivamente.
- O planejamento, cronograma e acompanhamento do projeto são guiados pela hierarquia de funcionalidades, em vez de um conjunto de tarefas de engenharia de software arbitrariamente adotado.

Coad e seus colegas [Coa99] sugerem o seguinte modelo para definir uma funcionalidade:

<acao> o <resultado> <por| para quem |de |para que> um <objeto>

em que um **<objeto>** é “uma pessoa, local ou coisa (inclusive papéis, momentos no tempo ou intervalos de tempo ou descrições parecidas com aquelas encontradas em catálogos)”. Exemplos de funcionalidades para uma aplicação de comércio eletrônico poderiam ser:

Adicione o produto ao carrinho

Mostre as especificações técnicas do produto

Armazene as informações de remessa para o cliente

Um conjunto de funcionalidades agrupa funcionalidades em categorias correlacionadas por negócio e é definido [Coa99] com:

<acao> um <objeto>

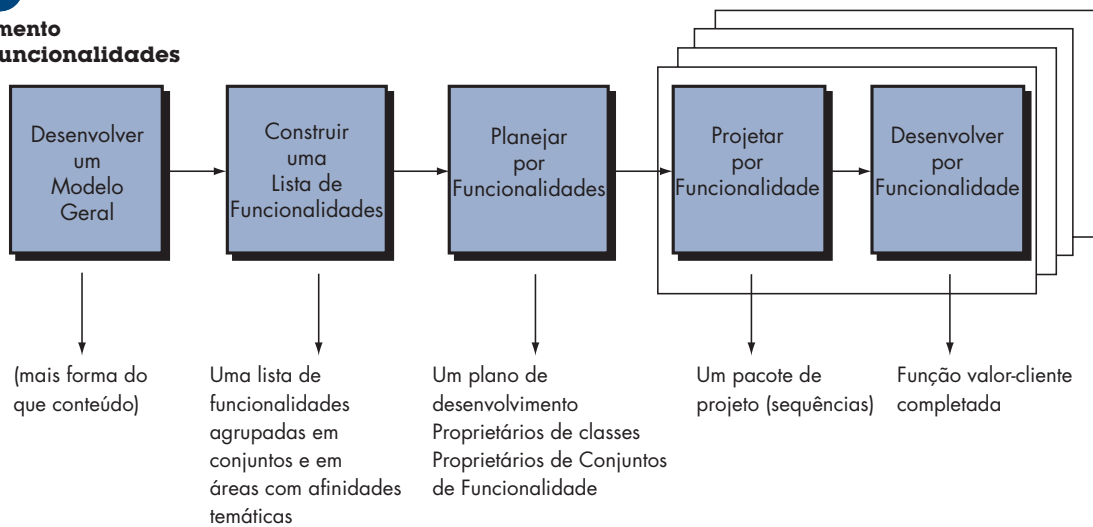
Por exemplo: *Fazer a venda de um produto* é um conjunto de funcionalidades que abrangeria os fatores percebidos anteriormente e outros.

A abordagem FDD define cinco atividades metodológicas “colaborativas” [Coa99] (no FDD estas são denominadas “processos”) conforme mostra a Figura 3.5.

O FDD oferece maior ênfase às diretrizes e técnicas de gerenciamento de projeto do que muitos outros métodos ágeis. Conforme os projetos crescem em tamanho e complexidade, com

FIGURA 3.5

**Desenvolvimento
dirigido a funcionalidades
(Coa99)
(com
permissão)**



frequência o gerenciamento de projeto para finalidade local torna-se inadequado. É essencial para os desenvolvedores, seus gerentes e outros envolvidos compreenderem o posicionamento do projeto — que realizações foram feitas e que problemas foram encontrados. Se a pressão do prazo de entrega for significativa, é crítico determinar se os incrementos de software (funcionalidades) foram agendados apropriadamente. Para tanto, o FDD define seis marcos durante o projeto e a implementação de uma funcionalidade: “desenrolar (*walkthroughs*) do projeto, projeto, inspeção de projeto, codificação, inspeção de código, progressão para construção/desenvolvimento” [Coa99].

3.5.6 Desenvolvimento de Software Enxuto (LSD)

O desenvolvimento de software enxuto (*Lean Software Development*) adaptou os princípios da fabricação enxuta para o mundo da engenharia de software. Os princípios enxutos que inspiraram o processo LSD podem ser sintetizados ([Pop03], [Pop06a]) em: *eliminar desperdício, incorporar qualidade, criar conhecimento, adiar compromissos, entregar rápido, respeitar as pessoas e otimizar o todo*.

Cada um dos princípios pode ser adaptado ao processo de software. Por exemplo, *eliminar desperdício* no contexto de um projeto de software ágil pode ser interpretado como [Das05]: (1) não adicionar recursos ou funções estranhas, (2) avaliar o impacto do custo e do cronograma de qualquer requisito solicitado recentemente, (3) eliminar quaisquer etapas de processo supérfluas, (4) estabelecer mecanismos para aprimorar o modo pelo qual a equipe levanta informações, (5) assegurar-se de que os testes encontrem o maior número de erros possível, (6) reduzir o tempo necessário para solicitar e obter uma decisão que afete o software ou o processo aplicado para criá-lo e (7) racionalizar a maneira pela qual informações são transmitidas a todos envolvidos no processo.

Para uma discussão detalhada do LSD e diretrizes pragmáticas para implementação do processo, consulte [Pop06a] e [Pop06b].

3.5.7 Modelagem Ágil (AM)

Existem muitas situações em que engenheiros de software têm de desenvolver sistemas grandes, com detalhes críticos em termos de negócio. O escopo e complexidade de tais sistemas devem ser modelados de modo que (1) todas as partes envolvidas possam entender melhor

WebRef

Informação ampla sobre a modelagem ágil pode ser encontrada em: www.agilemodeling.com.

“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



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

quais requisitos deverão 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.

Ao longo dos últimos 30 anos, uma ampla variedade de notações e métodos de modelagem de engenharia de software tem sido proposta para análise e projeto (tanto no nível de componente como de arquitetura). Esses métodos têm seus méritos, mas provaram ser difíceis de ser aplicados e desafiadores para ser mantidos (ao longo de vários projetos). Parte do problema é o “peso” dos métodos de modelagem. Com isso quero dizer o volume de notação exigido, o grau de formalismo sugerido, o puro tamanho dos modelos para grandes projetos e a dificuldade em manter o(s) modelo(s) à medida que ocorrem as mudanças. Contudo, o modelamento de análise e projeto tem um benefício substancial para grandes projetos — ainda que seja apenas para torná-los intelectualmente gerenciáveis. Existe uma abordagem ágil para a modelagem de engenharia de software que poderia fornecer uma alternativa?

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

Modelagem ágil (AM) consiste em uma metodologia baseada na prática, voltada para o modelamento e documentação de sistemas com base em software. Simplificando, modelagem ágil consiste em um conjunto de valores, princípios e práticas voltados para a modelagem do software que pode ser aplicados em um projeto de desenvolvimento de software de forma leve e efetiva. Os modelos ágeis são mais efetivos do que os tradicionais pelo fato de serem meramente bons, pois não têm a obrigação de ser perfeitos.

Modelagem ágil adota todos os valores consistentes 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 refabricação. 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 gama de princípios de modelagem essenciais e suplementares, os que tornam a AM única são [Amb02a]:

Modele com um objetivo. O desenvolvedor que utilizar o 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 óbvio o tipo de notação a ser utilizado e o nível de detalhamento necessário.

Use modelos múltiplos. Há muitos modelos e notações diferentes que podem ser usados para descrever software. Somente um subconjunto é essencial para a maioria dos projetos. AM sugere que, para propiciar o insight necessário, cada modelo deve apresentar um aspecto diferente do sistema e somente aqueles que valorizem esses modelos para a audiência pretendida devem ser usados.

Viajar leve. Conforme o trabalho de engenharia de software prossegue, conserve apenas aqueles modelos que terão valor no longo prazo e despache o restante. Todo produto de trabalho mantido deve sofrer manutenção à medida que as 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 informação para sua audiência pretendida. Um modelo sintaticamente perfeito que transmite pouco conteúdo útil não possui tanto valor como aquele com notações falhas que, no entanto, fornece conteúdo valioso para seu público-alvo.

Tenha conhecimento, domínio dos modelos e das ferramentas que for utilizar. Compreenda os pontos fortes e fracos de cada modelo e ferramenta usada para criá-lo.

Adapte localmente. A abordagem de 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)¹⁶ como o método preferido para análise representativa e para modelos de projeto. O Processo unificado (Capítulo 2) 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.

3.5.8 Processo Unificado Ágil (AUP)

O *processo unificado ágil* (*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 — *início, elaboração, construção e transição* —, AUP fornece uma camada serial (isto é, uma sequência linear de atividades de engenharia de software) que capacita uma equipe a visualizar o fluxo do processo geral de um projeto de software. Entretanto, dentro de cada atividade, a equipe itera ou se repete para alcançar a agilidade e para entregar incrementos de software significativos para os usuários finais tão rapidamente quanto possível. Cada iteração AUP dirige-se para as seguintes atividades [Amb06]:

- *Modelagem.* Representações UML do universo do negócio e do problema são criadas. Entretanto, para permanecer ágil, esses modelos devem ser “suficientemente bons e adequados” [Amb06] para possibilitar que a equipe prossiga.
- *Implementação.* Os modelos são traduzidos para o código-fonte.
- *Teste.* 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.
- *Aplicação.* Como a atividade de processo genérica discutida nos Capítulos 1 e 2, a aplicação neste contexto enfoca a entrega de um incremento de software e a aquisição de feedback dos usuários finais.
- *Configuração e gerenciamento de projeto.* No contexto da AUP, gerenciamento de configuração (Capítulo 22) refere-se a gerenciamento de alterações, de riscos e de controle de qualquer artefato¹⁷ persistente que sejam produzidos por uma equipe. Gerenciamento de projeto traciona e controla o progresso de uma equipe e coordena suas atividades.

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 2).

Mecânica: A mecânica das ferramentas varia. Em geral, conjuntos de ferramentas ágeis englobam suporte automatizado para o planejamento de projetos, desenvolvimento de caso prático, coletânea de requisitos, projeto rápido, geração de código e teste.

Ferramentas representativas:¹⁸

Nota: Por ser o desenvolvimento ágil um tópico importante, a maioria dos vendedores de ferramentas de software

tende a vender ferramentas que dão suporte para a abordagem ágil. As ferramentas aqui observadas 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 (www.ideogramic.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.

¹⁶ Um breve tutorial sobre a UML é apresentado no Apêndice 1.

¹⁷ *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, uma vez que o incremento de software seja entregue.

¹⁸ Ferramentas observadas aqui não significam um aval, mas antes, uma amostra de ferramentas nesta categoria. Na maioria dos casos, os nomes das ferramentas são negociados por seus respectivos desenvolvedores.

- *Gerenciamento do ambiente.* Coordena a infraestrutura de processos que inclui padrões, ferramentas e outras tecnologias de suporte disponíveis para a equipe.

Embora o AUP possua conexões históricas e técnicas com a linguagem de modelagem unificada, é importante notar que a modelagem UML pode ser usado em conjunto com quaisquer modelos de processos ágeis descritos na Seção 3.5.

3.6 UM CONJUNTO DE FERRAMENTAS PARA O PROCESSO ÁGIL

PONTO-CHAVE

O “conjunto de ferramentas” que suporta os processos ágeis focaliza mais as questões pessoais do que as questões tecnológicas.

Alguns proponentes da filosofia ágil argumentam que as ferramentas de software automatizadas (por exemplo, ferramentas para projetos) deveriam ser vistas como um suplemento menor para as atividades, e não como pivô para o sucesso da equipe. Entretanto, Alistair Cockburn [Coc04] sugere que ferramentas podem gerar um benefício e que “equipes ágeis enfatizam o uso de ferramentas que permitam 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”.

Pelo fato de que contratar as pessoas certas, ter a colaboração da equipe, manter a comunicação com os envolvidos e conseguir gerenciar de forma indireta constituírem elementos-chave em praticamente todos os modelos de processos ágeis, Cockburn afirma que “ferramentas” destinadas a esses itens são fatores críticos para a agilidade. Por exemplo, uma “ferramenta” alugada pode vir a ser um requisito para ter um membro de equipe de prospecção destinado a despendar algumas poucas horas em programação em dupla, com um membro já existente da equipe. O “encaixe” pode ser avaliado imediatamente.

“Ferramentas” voltadas para a comunicação e para a colaboração são, em geral, de tecnologia de base e incorporam qualquer mecanismo (“proximidade física, quadros brancos, papéis para pôster, fichas e lembretes adesivos” [Coc04]) que fornece informações e coordenação entre desenvolvedores. A comunicação ativa é obtida por meio de dinâmicas de grupo (por exemplo, programação em dupla), enquanto a comunicação passiva é obtida através 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 não enfatizam tanto o diagrama de Gantt e o realoca com quadros de valores ganhos ou “gráficos de testes criados e cruzados com os anteriores... Outras ferramentas ágeis são utilizadas para otimizar o ambiente no qual a equipe ágil trabalha (por exemplo, mais áreas eficientes de encontro), também para ampliar a cultura da equipe por meio de incentivos para interações sociais (por exemplo, equipes alocadas juntas), para dispositivos físicos (por exemplo, lousas eletrônicas) e para ampliação (por exemplo, programação em dupla ou janela de tempo)” [Coc04].

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

3.7 RESUMO

Em uma economia moderna, as condições de mercado mudam rapidamente, tanto o cliente quanto o usuário final devem evoluir e novos desafios competitivos surgem sem aviso. Os desenvolvedores têm de assumir uma abordagem de engenharia de software para permitir que permaneçam ágeis — definindo processos que sejam manipuláveis, adaptáveis, sem excessos, somente com o conteúdo essencial que possa adequar-se às necessidades do moderno mundo de negócios.

Uma filosofia ágil para a engenharia de software enfatiza quatro elementos-chave: a importância das equipes que se auto-organizam, que tenham controle sobre o trabalho por elas realizado, sobre a comunicação e sobre a colaboração entre os membros da equipe e entre os

desenvolvedores e seus clientes; o reconhecimento de que as mudanças representam oportunidades e ênfase na entrega rápida do software para satisfazer o cliente.

A Extreme Programming (XP) é o processo ágil mais amplamente utilizado. Organizada em quatro atividades metodológicas, planejamento, projeto, codificação e testes, a XP sugere um número de técnicas poderosas e inovadoras que possibilitam a uma equipe ágil criar versões de software frequentemente, propiciando recursos e funcionalidade estabelecidos anteriormente, e, então, priorizando os 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, ASD usa um processo iterativo que incorpora um planejamento cíclico iterativo, métodos de levantamento de requisitos relativamente rigorosos, e um ciclo de desenvolvimento iterativo que incorpora grupos focados nos clientes e revisões técnicas formais como mecanismos de feedback em tempo real. O Scrum enfatiza o uso de um conjunto de padrões de software que se mostrou efetivo 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 adapta 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 ao incremento seguinte. Crystal é uma família de modelos de processos ágeis que podem ser desenvolvidos para uma característica específica de um projeto.

O desenvolvimento dirigido a funcionalidades (FDD) é ligeiramente mais “formal” que os outros métodos, mas ainda mantém agilidade ao focar a equipe do projeto no desenvolvimento de funcionalidades — validadas pelo cliente que possam ser implementadas em duas semanas ou menos. O desenvolvimento de software enxuto (LSD) adaptou os princípios de uma fabricação enxuta para o mundo da engenharia de software. 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 de “serial para o que é amplo” e “iterativa para o que é particular” para o desenvolvimento de software.

PROBLEMAS E PONTOS A PONDERAR

- 3.1.** Leia “The 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?
- 3.2.** Descreva agilidade (para projetos de software) com suas próprias palavras.
- 3.3.** Por que um processo iterativo facilita o gerenciamento de mudanças? Todos os processos ágeis discutidos neste capítulo são iterativos? É possível completar um projeto com apenas uma iteração e ainda assim permanecer ágil? Justifique suas respostas.
- 3.4.** Cada um dos processos ágeis poderia ser descrito usando-se as atividades estruturais genéricas citadas no Capítulo 2? Construa uma tabela que associe as atividades genéricas às atividades definidas para cada processo ágil.
- 3.5.** Tente elaborar mais um “princípio de agilidade” que ajudaria uma equipe de engenharia de software a se tornar mais manobrável.
- 3.6.** Escolha um princípio de agilidade citado na Seção 3.3.1 e tente determinar se cada um dos modelos de processos apresentados neste capítulo demonstra o princípio. [Nota: Apresentei apenas uma visão geral desses modelos de processos; portanto, talvez não seja possível determinar se determinado princípio foi ou não tratado por um ou mais dos modelos, a menos que você pesquise mais a respeito (o que não é exigido para o presente problema).

- 3.7.** Por que os requisitos mudam tanto? Afinal de contas, as pessoas não sabem o que elas querem?
- 3.8.** A maior parte dos modelos de processos ágeis recomenda comunicação cara a cara. 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 para superar esse problema?
- 3.9.** Escreva uma história de usuário XP que descreva o recurso “sites favoritos” ou “bookmarks” disponível na maioria dos navegadores para Web.
- 3.10.** O que é uma solução pontual na XP?
- 3.11.** Descreva com suas próprias palavras os conceitos de refabricação e programação em dupla da XP.
- 3.12.** Leia um pouco mais a respeito e descreva o que é uma janela de tempo. Como isso ajuda uma equipe ASD na entrega de incrementos de software em um curto período de tempo?
- 3.13.** A regra dos 80% do DSDM e a abordagem de janelas de tempo definida para o ASD alcançam os mesmos resultados?
- 3.14.** Usando a planilha de padrões de processos apresentada no Capítulo 2, desenvolva um padrão de processo para qualquer um dos padrões Scrum da Seção 3.5.2.
- 3.15.** Por que o Crystal é considerado uma família de métodos ágeis?
- 3.16.** Usando o gabarito de recursos FDD descrito na Seção 3.5.5, defina um conjunto de recursos para um navegador Web. Agora, desenvolva vários recursos para o conjunto de recursos.
- 3.17.** Visite “The Official Agile Modeling Site” e faça uma lista completa de todos os princípios básicos e complementares do AM.
- 3.18.** O conjunto de ferramentas proposto na Seção 3.6 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 interessados 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 destes, livros como os de Shaw e Warden (*The Art of Agile Development*, O'Reilly Media, Inc., 2008), Hunt (*Agile Software Building*, 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), Highsmith (*Agile Project Management: Creating Innovative Products*, Addison-Wesley, 2004) 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 (*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 refabricação 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*, 2. ed., Addison-Wesley, 2004) ainda é o tratado de maior autoridade sobre o tema. Além desse, 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), bem como 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 dupla é apresentada por McBreen (*Pair Programming Illuminated*, Addison-Wesley, 2003).

A ASD é tratada em profundidade por Highsmith [Hig00]. Schwaber (*The Enterprise and Scrum*, Microsoft Press, 2007) discute o uso do Scrum para projetos que possuem um grande impacto sobre as empresas. Os detalhes práticos do Scrum são debatidos por 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*, 2. ed., Pearson Education, 2003) e Stapleton (*DSDM: The Method in Practice*, Addison-Wesley, 1997). Cockburn (*Crystal Clear*, Addison-Wesley, 2005) traz uma excelente visão geral da família Crystal de processos. Palmer e Felsing [Pal02] apresentam um tratado detalhado acerca do FDD. Carmichael e Haywood (*Better Software Faster*, Prentice-Hall, 2002) é mais um tratado útil sobre o FDD que inclui uma jornada passo a passo através da mecânica do processo. 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 na Web relevantes ao processo ágil pode ser encontrada no site **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.