CAPÍTULO

3

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26

Desenvolvimento ágil de software

Objetivos

O objetivo deste capítulo é apresentar os métodos ágeis de desenvolvimento de software. Ao terminar de ler este capítulo, você:

- compreenderá a lógica dos métodos ágeis de desenvolvimento de software, o manifesto ágil, e as diferenças entre desenvolvimento ágil e desenvolvimento dirigido a planos;
- conhecerá as práticas mais importantes da Extreme Programming e o modo como elas se relacionam com os princípios gerais dos métodos ágeis;
- compreenderá a abordagem Scrum para gerenciamento ágil de projetos;
- estará ciente das questões e problemas de escalamento de métodos ágeis de desenvolvimento para o desenvolvimento de sistemas de software de grande porte.

Conteúdo

Nos dias de hoje, as empresas operam em um ambiente global, com mudanças rápidas. Assim, precisam responder a novas oportunidades e novos mercados, a mudanças nas condições econômicas e ao surgimento de produtos e serviços concorrentes. Softwares fazem parte de quase todas as operações de negócios, assim, novos softwares são desenvolvidos rapidamente para obterem proveito de novas oportunidades e responder às pressões competitivas. O desenvolvimento e entrega rápidos são, portanto, o requisito mais crítico para o desenvolvimento de sistemas de software. Na verdade, muitas empresas estão dispostas a trocar a qualidade e o compromisso com requisitos do software por uma implantação mais rápida do software de que necessitam.

Essas empresas operam em um ambiente de mudanças rápidas, e por isso, muitas vezes, é praticamente impossível obter um conjunto completo de requisitos de software estável. Os requisitos iniciais inevitavelmente serão alterados, pois os clientes acham impossível prever como um sistema afetará as práticas de trabalho, como irá interagir com outros sistemas e quais operações do usuário devem ser automatizadas. Pode ser que os requisitos se tornem claros apenas após a entrega do sistema e à medida que os usuários ganhem experiência. Mesmo assim, devido a fatores externos, os requisitos são suscetíveis a mudanças rápidas e imprevisíveis. Por exemplo, quando for entregue, o software poderá estar desatualizado.

Processos de desenvolvimento de software que planejam especificar completamente os requisitos e, em seguida, projetar, construir e testar o sistema não estão adaptados ao desenvolvimento rápido de software. Com as mudanças nos requisitos ou a descoberta de problemas de requisitos, o projeto do sistema ou sua implementação precisa ser refeito ou

retestado. Como consequência, um processo convencional em cascata ou baseado em especificações costuma ser demorado, e o software final é entregue ao cliente bem depois do prazo acordado.

Para alguns tipos de software, como sistemas críticos de controle de segurança, em que uma análise completa do sistema é essencial, uma abordagem dirigida a planos é a melhor opção. No entanto, em um ambiente de negócio que se caracteriza por mudanças rápidas, isso pode causar problemas reais. Quando o software estiver disponível para uso, a razão original para sua aquisição pode ter mudado tão radicalmente que o software será efetivamente inútil. Portanto, para os sistemas de negócios, particularmente, os processos de desenvolvimento que se caracterizem por desenvolvimento e entrega rápidos de software são essenciais.

Há algum tempo já se reconhecia a necessidade de desenvolvimento rápido e de processos capazes de lidar com mudanças nos requisitos. Na década de 1980, a IBM introduziu o desenvolvimento incremental (MILLS et al., 1980). A introdução das linguagens de quarta geração, também em 1980, apoiou a ideia de desenvolvimento e entrega rápidos de software (MARTIN, 1981). No entanto, a ideia realmente decolou no final da década de 1990, com o desenvolvimento da noção de abordagens ágeis, como Metodologia de Desenvolvimento de Sistemas Dinâmicos (DSDM, do inglês *dynamic systems development method*) (STAPLETON, 1997), Scrum (SCHWABER e BEEDLE, 2001) e Extreme Programming (BECK, 1999; BECK, 2000).

Os processos de desenvolvimento rápido de software são concebidos para produzir, rapidamente, softwares úteis. O software não é desenvolvido como uma única unidade, mas como uma série de incrementos — cada incremento inclui uma nova funcionalidade do sistema. Embora existam muitas abordagens para o desenvolvimento rápido de software, elas compartilham algumas características fundamentais:

1. Os processos de especificação, projeto e implementação são intercalados. Não há especificação detalhada do sistema, e a documentação do projeto é minimizada ou gerada automaticamente pelo ambiente de programação usado para implementar o sistema. O documento de requisitos do usuário apenas define as características mais importantes do sistema.
2. O sistema é desenvolvido em uma série de versões. Os usuários finais e outros *stakeholders* do sistema são envolvidos na especificação e avaliação de cada versão. Eles podem propor alterações ao software e novos requisitos que devem ser implementados em uma versão posterior do sistema.
3. Interfaces de usuário do sistema são geralmente desenvolvidas com um sistema interativo de desenvolvimento que permite a criação rápida do projeto de interface por meio de desenho e posicionamento de ícones na interface. O sistema pode, então, gerar uma interface baseada na Web para um navegador ou uma interface para uma plataforma específica, como o Microsoft Windows.

Os métodos ágeis são métodos de desenvolvimento incremental em que os incrementos são pequenos e, normalmente, as novas versões do sistema são criadas e disponibilizadas aos clientes a cada duas ou três semanas. Elas envolvem os clientes no processo de desenvolvimento para obter *feedback* rápido sobre a evolução dos requisitos. Assim, minimiza-se a documentação, pois se utiliza mais a comunicação informal do que reuniões formais com documentos escritos.

3.1 Métodos ágeis

Na década de 1980 e início da de 1990, havia uma visão generalizada de que a melhor maneira para conseguir o melhor software era por meio de um planejamento cuidadoso do projeto, qualidade da segurança formalizada, do uso de métodos de análise e projeto apoiado por ferramentas CASE (*Computer-aided software engineering*) e do processo de desenvolvimento de software rigoroso e controlado. Essa percepção veio da comunidade de engenharia de software, responsável pelo desenvolvimento de sistemas de software grandes e duradouros, como sistemas aeroespaciais e de governo.

Esse software foi desenvolvido por grandes equipes que trabalham para diferentes empresas. Geralmente, as equipes eram dispersas geograficamente e trabalhavam com o software por longos períodos. Um exemplo desse tipo de software é o sistema de controle de uma aeronave moderna, que pode demorar até dez anos desde a especificação inicial até a implantação. Tais abordagens dirigidas a planos envolvem um *overhead* significativo no planejamento, projeto e documentação do sistema. Esse *overhead* se justifica quando o trabalho de várias equipes de desenvolvimento tem de ser coordenado, quando o sistema é um sistema crítico e quando muitas pessoas diferentes estão envolvidas na manutenção do software durante sua vida.

No entanto, quando essa abordagem pesada de desenvolvimento dirigido a planos é aplicada aos sistemas corporativos de pequeno e médio porte, o *overhead* envolvido é tão grande que domina o processo de desenvolvimento de software. Gasta-se mais tempo em análises de como o sistema deve ser desenvolvido do que no

desenvolvimento de programas e testes. Como os requisitos do sistema se alteram, o retrabalho é essencial, e, pelo menos em princípio, a especificação e o projeto devem mudar com o programa.

A insatisfação com essas abordagens pesadas da engenharia de software levou um grande número de desenvolvedores de software a proporem, na década de 1990, novos 'métodos ágeis'. Estes permitiram que a equipe de desenvolvimento focasse no software em si, e não em sua concepção e documentação. Métodos ágeis, universalmente, baseiam-se em uma abordagem incremental para a especificação, o desenvolvimento e a entrega do software. Eles são mais adequados ao desenvolvimento de aplicativos nos quais os requisitos de sistema mudam rapidamente durante o processo de desenvolvimento. Destinam-se a entregar o software rapidamente aos clientes, em funcionamento, e estes podem, em seguida, propor alterações e novos requisitos a serem incluídos nas iterações posteriores do sistema. Têm como objetivo reduzir a burocracia do processo, evitando qualquer trabalho de valor duvidoso de longo prazo e qualquer documentação que provavelmente nunca será usada.

A filosofia por trás dos métodos ágeis é refletida no manifesto ágil, que foi acordado por muitos dos principais desenvolvedores desses métodos. Esse manifesto afirma:

> *Estamos descobrindo melhores maneiras de desenvolver softwares, fazendo-o e ajudando outros a fazê-lo. Através desse trabalho, valorizamos mais:*
>
> *Indivíduos e interações do que processos e ferramentas*
>
> *Software em funcionamento do que documentação abrangente*
>
> *Colaboração do cliente do que negociação de contrato*
>
> *Respostas a mudanças do que seguir um plano*
>
> *Ou seja, embora itens à direita sejam importantes, valorizamos mais os que estão à esquerda.*

Provavelmente, o método ágil mais conhecido é a Extreme Programming (BECK, 1999; BECK, 2000), que descrevo adiante neste capítulo. Outras abordagens ágeis incluem Scrum (COHN, 2009; SCHWABER, 2004; SCHWABER e BEEDLE, 2001), Crystal (COCKBURN, 2001; COCKBURN, 2004), Desenvolvimento de Software Adaptativo (*Adaptative Software Development*), (HIGHSMITH, 2000), DSDM (STAPLETON, 1997; STAPLETON, 2003) e Desenvolvimento Dirigido a Características (*Feature Driven Development*), (PALMER e FELSING, 2002). O sucesso desses métodos levou a uma certa integração com métodos mais tradicionais de desenvolvimento baseados na modelagem do sistema, resultando no conceito de modelagem ágil (AMBLER e JEFFRIES, 2002) e instâncias ágeis do *Rational Unified Process* (LARMAN, 2002).

Embora esses métodos ágeis sejam todos baseados na noção de desenvolvimento e entrega incremental, eles propõem diferentes processos para alcançar tal objetivo. No entanto, compartilham um conjunto de princípios, com base no manifesto ágil, e por isso têm muito em comum. Esses princípios são mostrados na Tabela 3.1. Diferentes métodos ágeis instanciam esses princípios de maneiras diferentes, e eu não tenho espaço para discutir todos os métodos ágeis. Em vez disso, foco em dois dos mais usados: Extreme Programming (Seção 3.3) e Scrum (Seção 3.4).

Tabela 3.1 Os princípios dos métodos ágeis

Princípios	Descrição
Envolvimento do cliente	Os clientes devem estar intimamente envolvidos no processo de desenvolvimento. Seu papel é fornecer e priorizar novos requisitos do sistema e avaliar suas iterações.
Entrega incremental	O software é desenvolvido em incrementos com o cliente, especificando os requisitos para serem incluídos em cada um.
Pessoas, não processos	As habilidades da equipe de desenvolvimento devem ser reconhecidas e exploradas. Membros da equipe devem desenvolver suas próprias maneiras de trabalhar, sem processos prescritivos.
Aceitar as mudanças	Deve-se ter em mente que os requisitos do sistema vão mudar. Por isso, projete o sistema de maneira a acomodar essas mudanças.
Manter a simplicidade	Focalize a simplicidade, tanto do software a ser desenvolvido quanto do processo de desenvolvimento. Sempre que possível, trabalhe ativamente para eliminar a complexidade do sistema.

Métodos ágeis têm sido muito bem-sucedidos para alguns tipos de desenvolvimento de sistemas:

1. O desenvolvimento de produtos, em que uma empresa de software está desenvolvendo um produto pequeno ou médio para venda.
2. Desenvolvimento de sistema personalizado dentro de uma organização, em que existe um compromisso claro do cliente de se envolver no processo de desenvolvimento, e em que não há muitas regras e regulamentos externos que afetam o software.

Como discuto na seção final deste capítulo, o sucesso dos métodos ágeis indica que há um grande interesse em usar esses métodos para outros tipos de desenvolvimento de software. No entanto, devido a seu foco em pequenas equipes bem integradas, existem problemas em escalá-los para grandes sistemas. Existem também experiências de uso de abordagens ágeis para engenharia de sistemas críticos (DROBNA et al., 2004). No entanto, devido à necessidade de proteção, segurança e análise de confiança em sistemas críticos, exigem-se modificações significativas nos métodos ágeis antes que possam ser rotineiramente usados para a engenharia de sistemas críticos.

Na prática, os princípios básicos dos métodos ágeis são, por vezes, difíceis de se concretizar:

1. Embora a ideia de envolvimento do cliente no processo de desenvolvimento seja atraente, seu sucesso depende de um cliente disposto e capaz de passar o tempo com a equipe de desenvolvimento, e que possa representar todos os *stakeholders* do sistema. Frequentemente, os representantes dos clientes estão sujeitos a diversas pressões e não podem participar plenamente do desenvolvimento de software.
2. Membros individuais da equipe podem não ter personalidade adequada para o intenso envolvimento que é típico dos métodos ágeis e, portanto, não interagem bem com outros membros da equipe.
3. Priorizar as mudanças pode ser extremamente difícil, especialmente em sistemas nos quais existem muitos *stakeholders*. Normalmente, cada *stakeholder* dá prioridades diferentes para mudanças diferentes.
4. Manter a simplicidade exige um trabalho extra. Sob a pressão de cronogramas de entrega, os membros da equipe podem não ter tempo para fazer as simplificações desejáveis.
5. Muitas organizações, principalmente as grandes empresas, passaram anos mudando sua cultura para que os processos fossem definidos e seguidos. É difícil para eles mudar de um modelo de trabalho em que os processos são informais e definidos pelas equipes de desenvolvimento.

Outro problema não técnico — que é um problema geral do desenvolvimento e da entrega incremental — ocorre quando o cliente do sistema usa uma organização externa para desenvolver o sistema. O documento de requisitos do software é normalmente parte do contrato entre o cliente e o fornecedor. Como a especificação incremental é inerente aos métodos ágeis, escrever contratos para esse tipo de desenvolvimento pode ser difícil.

Consequentemente, os métodos ágeis têm de contar com contratos nos quais o cliente paga pelo tempo necessário para o desenvolvimento do sistema, e não pelo desenvolvimento de um determinado conjunto de requisitos. Enquanto tudo vai bem, isso beneficia o cliente e o desenvolvedor. No entanto, se surgirem problemas, poderão acontecer disputas nas quais fica difícil definir quem é culpado e quem deve pagar pelo tempo extra, bem como os recursos necessários para a solução dos problemas.

A maioria dos livros e artigos que descrevem os métodos ágeis e experiências com eles fala sobre o uso desses métodos para o desenvolvimento de novos sistemas. No entanto, como explico no Capítulo 9, um enorme esforço da engenharia de software é dedicado à manutenção e à evolução de sistemas de software já existentes. Existe apenas um pequeno número de relatos de experiências com o uso de métodos ágeis na manutenção de software (POOLE e HUISMAN, 2001). Há duas questões que devem ser consideradas ao tratarmos de métodos ágeis e manutenção:

1. É possível fazer manutenção dos sistemas desenvolvidos em uma abordagem ágil, dada a ênfase do processo de desenvolvimento em minimização da documentação formal?
2. Os métodos ágeis podem, efetivamente, ser usados para a evolução de um sistema em resposta às solicitações de mudança do cliente?

Supostamente, a documentação formal deve descrever o sistema e, assim, tornar sua compreensão mais fácil para as pessoas que fazem as mudanças. Porém, na prática, a documentação formal nem sempre é atualizada, e, portanto, não reflete exatamente o código do programa. Por essa razão, os entusiastas de métodos ágeis argumentam que é um desperdício de tempo criar essa documentação e que a chave para a implementação de software manutenível é a produção de códigos de alta qualidade, legíveis. Práticas ágeis, portanto, enfatizam a importância de se escrever códigos bem-estruturados e investir na melhoria do código. Portanto, a falta de documentação não deve ser um problema na manutenção dos sistemas desenvolvidos por meio de uma abordagem ágil.

No entanto, minha experiência em manutenção de sistemas sugere que o documento-chave é o documento de requisitos do sistema, que informa ao engenheiro de software o que o sistema deve fazer. Sem esse conhecimento, é difícil avaliar o impacto das mudanças propostas. Muitos métodos ágeis coletam requisitos informalmente, de forma incremental, e não desenvolvem um documento coerente de requisitos. Nesse sentido, o uso de métodos ágeis pode tornar mais difícil e cara a manutenção posterior do sistema.

Práticas ágeis, usadas no processo de manutenção em si, provavelmente são mais eficazes, independente de ter sido usada uma abordagem ágil para o desenvolvimento do sistema. Entrega incremental, projeto para mudanças e manutenção da simplicidade — tudo isso faz sentido quando o software está sendo alterado. Na verdade, você pode pensar em um processo ágil de desenvolvimento como um processo de evolução do software.

No entanto, a principal dificuldade após a entrega do software é manter o envolvimento dos clientes no processo. Apesar de, durante o desenvolvimento do sistema, um cliente poder justificar o envolvimento de um representante em tempo integral, isso é menos provável durante a manutenção, período em que as mudanças não são contínuas. Representantes do cliente são propensos a perder o interesse no sistema. Portanto, para criar novos requisitos do sistema podem ser necessários mecanismos alternativos, como as propostas de mudanças discutidas no Capítulo 25.

O outro problema que pode surgir está relacionado à continuidade da equipe de desenvolvimento. Métodos ágeis dependem de os membros da equipe compreenderem aspectos do sistema sem consultar a documentação. Se uma equipe de desenvolvimento ágil é alterada, esse conhecimento implícito é perdido, e é difícil para os novos membros da equipe construir o mesmo entendimento do sistema e seus componentes.

Os defensores dos métodos ágeis foram evangelizadores na promoção do uso desses métodos e tenderam a negligenciar suas deficiências. Isso provocou uma resposta igualmente extrema, que, em minha opinião, exagera os problemas dessa abordagem (STEPHENS e ROSENBERG, 2003). Críticos mais fundamentados, como DeMarco e Boehm (2002), destacam as vantagens e desvantagens dos métodos ágeis. Eles propõem uma abordagem híbrida, na qual os métodos ágeis incorporam algumas técnicas do desenvolvimento dirigido a planos, que pode ser o melhor caminho a seguir.

3.2 Desenvolvimento ágil e dirigido a planos

Abordagens ágeis de desenvolvimento de software consideram o projeto e a implementação como atividades centrais no processo de software. Eles incorporam outras atividades, como elicitação de requisitos e testes no projeto e na implementação. Em contrapartida, uma abordagem de engenharia de software dirigida a planos identifica estágios distintos do processo de software com saídas associadas a cada estágio. As saídas de um estágio são usadas como base para o planejamento da atividade do processo a seguir. A Figura 3.1 mostra as distinções entre as abordagens dirigidas a planos e ágil para a especificação do sistema.

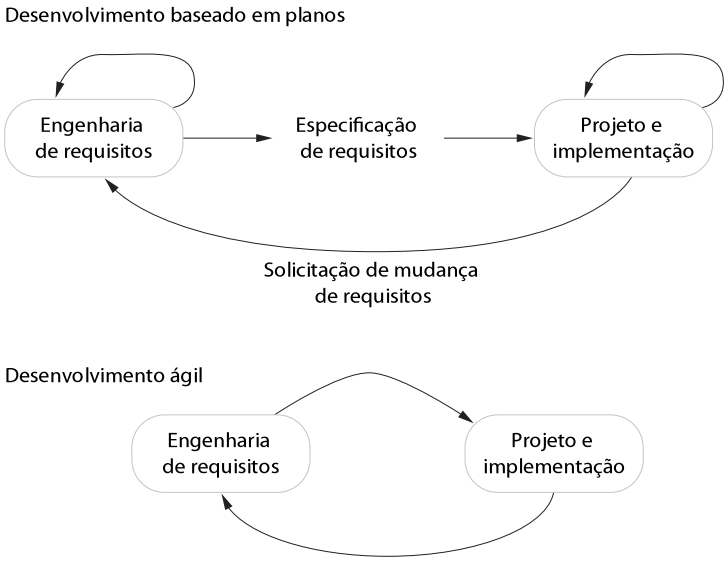
Em uma abordagem dirigida a planos, ocorrem iterações no âmbito das atividades com documentos formais, usados para estabelecer a comunicação entre os estágios do processo. Por exemplo, os requisitos vão evoluir e, finalmente, será produzida uma especificação de requisitos. Essa é, então, uma entrada para o processo de projeto e implementação. Em uma abordagem ágil, iterações ocorrem em todas as atividades. Portanto, os requisitos e o projeto são desenvolvidos em conjunto, e não separadamente.

Um processo de software dirigido a planos pode apoiar o desenvolvimento e a entrega incremental. É perfeitamente possível alocar requisitos e planejar as fases de projeto e desenvolvimento como uma série de incrementos. Um processo ágil não é, inevitavelmente, focado no código, e pode produzir alguma documentação de projeto. Como vou discutir na seção seguinte, a equipe de desenvolvimento ágil pode decidir incluir um '*spike*' de documentação, no qual, em vez de produzir uma nova versão de um sistema, a equipe produz documentação do sistema.

Na verdade, a maioria dos projetos de software inclui práticas das abordagens dirigidas a planos e ágil. Para optar por um equilíbrio entre as abordagens, você precisa responder a uma série de questões técnicas, humanas e organizacionais:

1. É importante ter uma especificação e um projeto muito detalhados antes de passar para a implementação? Se sim, você provavelmente necessita usar uma abordagem dirigida a planos.
2. É realista uma estratégia de entrega incremental em que você entrega o software aos clientes e rapidamente obtém um *feedback*? Em caso afirmativo, considere o uso de métodos ágeis.

Figura 3.1 Especificações dirigida a planos e ágil



3. Quão grande é o sistema que está em desenvolvimento? Os métodos ágeis são mais eficazes quando o sistema pode ser desenvolvido com uma pequena equipe colocalizada capaz de se comunicar de maneira informal. Isso pode não ser possível para sistemas de grande porte que exigem equipes de desenvolvimento maiores — nesse caso, uma abordagem dirigida a planos pode ter de ser usada.
4. Que tipo de sistema está sendo desenvolvido? Sistemas que exigem uma análise profunda antes da implementação (por exemplo, sistema de tempo real com requisitos de tempo complexos) geralmente demandam um projeto bastante detalhado para atender a essa análise. Nessas circunstâncias, uma abordagem dirigida a planos pode ser a melhor opção.
5. Qual é o tempo de vida esperado do sistema? Sistemas de vida-longa podem exigir mais da documentação de projeto, a fim de comunicar para a equipe de apoio as intenções originais dos desenvolvedores do sistema. No entanto, os defensores dos métodos ágeis argumentam, corretamente, que a documentação não é constantemente atualizada e não é de muita utilidade para a manutenção do sistema a longo prazo.
6. Que tecnologias estão disponíveis para apoiar o desenvolvimento do sistema? Métodos ágeis frequentemente contam com boas ferramentas para manter o controle de um projeto em desenvolvimento. Se você está desenvolvendo um sistema utilizando um IDE (ambiente integrado de desenvolvimento, do inglês *integrated development environment*) que não tem boas ferramentas para visualização e análise do programa, pode haver necessidade de mais documentação do projeto.
7. Como é organizada a equipe de desenvolvimento? Se está distribuída, ou se parte do desenvolvimento está sendo terceirizado, então pode ser necessário o desenvolvimento de documentos de projeto para a comunicação entre as equipes de desenvolvimento. Pode ser necessário planejar com antecedência quais serão esses documentos.
8. Existem questões culturais que podem afetar o desenvolvimento do sistema? Organizações tradicionais de engenharia têm uma cultura de desenvolvimento baseado em planos, pois essa é a norma na engenharia. Geralmente, isso requer extensa documentação de projeto, no lugar do conhecimento informal, usado em processos ágeis.
9. Quão bons são os projetistas e programadores na equipe de desenvolvimento? Às vezes, argumenta-se que os métodos ágeis exigem níveis mais altos de habilidade do que as abordagens dirigidas a planos, em que os programadores simplesmente traduzem um projeto detalhado em um código. Se você tem uma equipe com níveis de habilidade relativamente baixos, pode precisar usar as melhores pessoas para desenvolver o projeto, juntamente com outros, responsáveis pela programação.
10. O sistema é sujeito à regulamentação externa? Se um sistema tem de ser aprovado por um regulador externo (por exemplo, a FAA [Autoridade Federal de Aviação, do inglês *Federal Aviation Authority*] aprova os softwares

críticos para a operação de uma aeronave), então, provavelmente, será obrigatória a produção de uma documentação detalhada como parte da documentação de segurança do sistema.

Na realidade, a questão sobre rotular o projeto como dirigido a planos ou ágil não é muito importante. Em última análise, a principal preocupação dos compradores de um sistema de software é se estão comprando um sistema de software executável que atenda às suas necessidades e faça coisas úteis para o usuário individual ou para a organização. Na prática, muitas empresas que afirmam ter usado métodos ágeis adotaram algumas práticas ágeis e integraram-nas em seus processos dirigidos a planos.

3.3 Extreme Programming

Extreme Programming (XP) é talvez o mais conhecido e mais utilizado dos métodos ágeis. O nome foi cunhado por Beck (2000), pois a abordagem foi desenvolvida para impulsionar práticas reconhecidamente boas, como o desenvolvimento iterativo, a níveis 'extremos'. Por exemplo, em XP, várias novas versões de um sistema podem ser desenvolvidas, integradas e testadas em um único dia por programadores diferentes.

Em Extreme Programming, os requisitos são expressos como cenários (chamados de estórias do usuário), que são implementados diretamente como uma série de tarefas. Os programadores trabalham em pares e desenvolvem testes para cada tarefa antes de escreverem o código. Quando o novo código é integrado ao sistema, todos os testes devem ser executados com sucesso. Há um curto intervalo entre os *releases* do sistema. A Figura 3.2 ilustra o processo XP para a produção de um incremento do sistema que está sendo desenvolvido.

Extreme Programming envolve uma série de práticas que refletem os princípios dos métodos ágeis (elas estão resumidas na Tabela 3.2):

1. O desenvolvimento incremental é sustentado por meio de pequenos e frequentes releases do sistema. Os requisitos são baseados em cenários ou em simples estórias de clientes, usadas como base para decidir a funcionalidade que deve ser incluída em um incremento do sistema.
2. O envolvimento do cliente é sustentado por meio do engajamento contínuo do cliente com a equipe de desenvolvimento. O representante do cliente participa do desenvolvimento e é responsável por definir os testes de aceitação para o sistema.
3. Pessoas — não processos — são sustentadas por meio de programação em pares, propriedade coletiva do código do sistema e um processo de desenvolvimento sustentável que não envolve horas de trabalho excessivamente longas.
4. As mudanças são aceitas por meio de *releases* contínuos para os clientes, do desenvolvimento *test-first*, da refatoração para evitar a degeneração do código e integração contínua de nova funcionalidade.
5. A manutenção da simplicidade é feita por meio da refatoração constante que melhora a qualidade do código, bem como por meio de projetos simples que não antecipam desnecessariamente futuras mudanças no sistema.

Em um processo XP, os clientes estão intimamente envolvidos na especificação e priorização dos requisitos do sistema. Os requisitos não estão especificados como uma lista de funções requeridas do sistema. Pelo contrário, o cliente do sistema é parte da equipe de desenvolvimento e discute cenários com outros membros da equipe. Juntos, eles desenvolvem um 'cartão de estórias', englobando as necessidades do cliente. A equipe de

Figura 3.2 O ciclo de um *release* em Extreme Programming

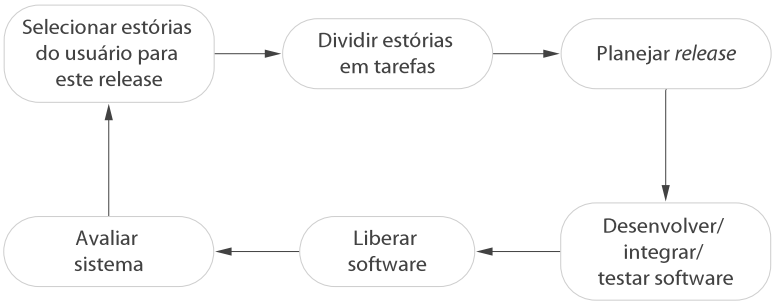


Tabela 3.2 Práticas de Extreme Programming

Princípio ou prática	Descrição
Planejamento incremental	Os requisitos são gravados em cartões de estória e as estórias que serão incluídas em um release são determinadas pelo tempo disponível e sua relativa prioridade. Os desenvolvedores dividem essas estórias em 'Tarefas'. Veja os quadros 3.1 e 3.2.
Pequenos *releases*	Em primeiro lugar, desenvolve-se um conjunto mínimo de funcionalidades útil, que fornece o valor do negócio. *Releases* do sistema são frequentes e gradualmente adicionam funcionalidade ao primeiro *release*.
Projeto simples	Cada projeto é realizado para atender às necessidades atuais, e nada mais.
Desenvolvimento *test-first*	Um *framework* de testes iniciais automatizados é usado para escrever os testes para uma nova funcionalidade antes que a funcionalidade em si seja implementada.
Refatoração	Todos os desenvolvedores devem refatorar o código continuamente assim que encontrarem melhorias de código. Isso mantém o código simples e manutenível.
Programação em pares	Os desenvolvedores trabalham em pares, verificando o trabalho dos outros e prestando apoio para um bom trabalho sempre.
Propriedade coletiva	Os pares de desenvolvedores trabalham em todas as áreas do sistema, de modo que não se desenvolvam ilhas de *expertise*. Todos os conhecimentos e todos os desenvolvedores assumem responsabilidade por todo o código. Qualquer um pode mudar qualquer coisa.
Integração contínua	Assim que o trabalho em uma tarefa é concluído, ele é integrado ao sistema como um todo. Após essa integração, todos os testes de unidade do sistema devem passar.
Ritmo sustentável	Grandes quantidades de horas-extra não são consideradas aceitáveis, pois o resultado final, muitas vezes, é a redução da qualidade do código e da produtividade a médio prazo.
Cliente no local	Um representante do usuário final do sistema (o cliente) deve estar disponível todo o tempo à equipe de XP. Em um processo de Extreme Programming, o cliente é um membro da equipe de desenvolvimento e é responsável por levar a ela os requisitos de sistema para implementação.

desenvolvimento, então, tenta implementar esse cenário em um *release* futuro do software. O Quadro 3.1 mostra um exemplo de um cartão de estória para o gerenciamento do sistema de cuidado da saúde mental de pacientes. Essa é uma breve descrição de um cenário para a prescrição de medicamentos a um paciente.

Quadro 3.1 Uma estória de prescrição de medicamentos

Prescrição de medicamentos
Kate é uma médica que deseja prescrever medicamentos para um paciente de uma clínica. O prontuário do paciente já está sendo exibido em seu computador, assim, ela clica o campo 'medicação' e pode selecionar 'medicação atual', 'nova medicação', ou 'formulário'. Se ela selecionar 'medicação atual', o sistema pede que ela verifique a dose. Se ela quiser mudar a dose, ela altera esta e em seguida, confirma a prescrição. Se ela escolher 'nova medicação', o sistema assume que ela sabe qual medicação receitar. Ela digita as primeiras letras do nome do medicamento. O sistema exibe uma lista de possíveis fármacos que começam com essas letras. Ela escolhe a medicação requerida e o sistema responde, pedindo-lhe para verificar se o medicamento selecionado está correto. Ela insere a dose e, em seguida, confirma a prescrição. Se ela escolhe 'formulário', o sistema exibe uma caixa de busca para o formulário aprovado. Ela pode, então, procurar pelo medicamento requerido. Ela seleciona um medicamento e é solicitado que verifique se a medicação está correta. Ela insere a dose e, em seguida, confirma a prescrição. O sistema sempre verifica se a dose está dentro da faixa permitida. Caso não esteja, Kate é convidada a alterar a dose. Após Kate confirmar a prescrição, esta será exibida para verificação. Ela pode escolher 'OK' ou 'Alterar'. Se clicar em 'OK', a prescrição fica gravada nos bancos de dados da auditoria. Se ela clicar em 'Alterar', reinicia o processo de 'Prescrição de Medicamentos'.

Os cartões de estória são as principais entradas para o processo de planejamento em XP ou 'jogo de planejamento'. Uma vez que tenham sido desenvolvidos, a equipe de desenvolvimento os divide em tarefas (Quadro 3.2) e estima o esforço e os recursos necessários para a realização de cada tarefa. Esse processo geralmente envolve discussões com o cliente para refinamento dos requisitos. O cliente, então, prioriza as estórias para implementação, escolhendo aquelas que podem ser usadas imediatamente para oferecer apoio aos negócios. A intenção é identificar funcionalidade útil que possa ser implementada em cerca de duas semanas, quando o próximo *release* do sistema é disponibilizado para o cliente.

Claro que, como os requisitos mudam, as estórias não implementadas mudam ou podem ser descartadas. Se houver necessidade de mudanças em um sistema que já tenha sido entregue, novos cartões de estória são desenvolvidos e, mais uma vez, o cliente decide se essas mudanças devem ter prioridade sobre a nova funcionalidade.

Às vezes, durante o jogo de planejamento, emergem questões que não podem ser facilmente respondidas, tornando necessário algum trabalho adicional para explorar possíveis soluções. A equipe pode fazer algum protótipo ou desenvolvimento-teste para entender o problema e a solução. Em termos XP, isso é um '*spike*', um incremento em que nenhum tipo de programação é realizado. Também pode haver '*spikes*' de projeto da arquitetura do sistema ou para desenvolver a documentação do sistema.

Extreme Programming leva uma abordagem 'extrema' para o desenvolvimento incremental. Novas versões do software podem ser construídas várias vezes por dia e *releases* são entregues aos clientes a cada duas semanas, aproximadamente. Prazos de *releases* nunca são desrespeitados; se houver problemas de desenvolvimento, o cliente é consultado, e a funcionalidade é removida do *release* planejado.

Quando um programador constrói o sistema para criar uma nova versão, deve executar todos os testes automatizados existentes, bem como os testes para a nova funcionalidade. A nova construção do software só é aceita se todos os testes forem executados com êxito. Esta se torna, então, a base para a próxima iteração do sistema.

Um preceito fundamental da engenharia de software tradicional é que você deve projetar para mudar. Ou seja, você deve antecipar futuras alterações do software e projetá-lo para que essas mudanças possam ser facilmente implementadas. O Extreme Programming, no entanto, descartou esse princípio com base na concepção de que muitas vezes a mudança é um esforço desperdiçado. Não vale a pena perder tempo adicionando generalidades a um programa para lidar com mudanças. Frequentemente, mudanças previstas não se materializam e solicitações por mudanças completamente diferentes podem ser feitas. Portanto, a abordagem XP aceita que as mudanças acontecerão e reorganizarão o software quando essas mudanças realmente acontecerem.

Um problema geral com o desenvolvimento incremental é que ele tende a degradar a estrutura do software. Desse modo, as mudanças para o software tornam-se cada vez mais difíceis de serem implementadas. Essencialmente, o desenvolvimento prossegue, encontrando soluções para os problemas, mas o resultado final frequentemente é a duplicação do código; partes do software são reusadas de maneira inadequada, e a estrutura global do código degrada-se quando ele é adicionado ao sistema.

Quadro 3.2 Exemplos de cartões de tarefa para a prescrição de medicamentos

Tarefa 1: Alterar dose de medicamentos prescritos
Tarefa 2: Seleção de formulário
Tarefa 3: Verificação de dose
A verificação da dose é uma precaução de segurança para verificar se o médico não receitou uma dose perigosamente pequena ou grande. Usando o ID do formulário para o nome do medicamento genérico, procure o formulário e obtenha a dose mínima e máxima recomendada. Verifique a dose mínima e máxima prescrita. Caso esteja fora da faixa, emita uma mensagem de erro dizendo que a dose está muito alta ou muito baixa. Caso esteja dentro da faixa, habilite o botão 'Confirmar'.

Extreme Programming aborda esse problema, sugerindo que o software deve ser constantemente refatorado. Isso significa que a equipe de programação deve estar focada em possíveis melhorias para o software e em implementação imediata destas. Quando um membro da equipe percebe que o código pode ser melhorado, essas melhorias são feitas, mesmo quando não existe necessidade imediata destas. Exemplos de refatoração incluem a reorganização da hierarquia de classes para eliminação de código duplicado, a arrumação e renomeação de atributos e métodos, bem como a substituição do código com as chamadas para métodos definidos em uma biblioteca de programas. Ambientes de desenvolvimento de programas, como o Eclipse (CARLSON, 2005), incluem ferramentas de refatoração que simplificam o processo de encontrar dependências entre as seções do código e fazer as modificações no código global.

Em princípio, portanto, o software deve ser sempre fácil de compreender e mudar à medida que novas estórias sejam implementadas. Na prática, isso nem sempre ocorre. Em alguns casos, a pressão pelo desenvolvimento significa que a refatoração será postergada, porque a maior parte do tempo é dedicada à implementação de nova funcionalidade. Algumas novas características e mudanças não podem ser facilmente acomodadas por refatoração do nível do código, e exigem modificações da arquitetura do sistema.

Na prática, muitas empresas que adotaram XP não usam todas as práticas da Extreme Programming listadas na Tabela 3.2. Elas escolhem de acordo com sua organização. Por exemplo, algumas empresas consideram útil a programação em pares, outras preferem a programação individual e revisões. Para acomodar os diferentes níveis de habilidade, alguns programadores não fazem refatoração em partes do sistema que não desenvolveram, e podem ser usados os requisitos convencionais em vez de estórias de usuários. No entanto, a maioria das empresas que adotaram uma variante de XP usa *releases* de pequeno porte, desenvolvimento do *test-first* e integração contínua.

3.3.1 Teste em XP

Como discutido na introdução deste capítulo, uma das diferenças importantes entre o desenvolvimento incremental e o desenvolvimento dirigido a planos está na forma como o sistema é testado. Com o desenvolvimento incremental, não há especificação do sistema que possa ser usada por uma equipe de teste externa para desenvolvimento de testes do sistema. Como consequência, algumas abordagens para o desenvolvimento incremental têm um processo de testes muito informal em comparação com os testes dirigidos a planos.

Para evitar alguns dos problemas de teste e validação do sistema, a abordagem XP enfatiza a importância dos testes do programa. Extreme Programming inclui uma abordagem de testes que reduz as chances de erros desconhecidos na versão atual do sistema.

As principais características dos testes em XP são:

1. desenvolvimento *test-first*;
2. desenvolvimento de teste incremental a partir de cenários;
3. envolvimento dos usuários no desenvolvimento de testes e validação;
4. uso de *frameworks* de testes automatizados.

O desenvolvimento *test-first* é uma das mais importantes inovações no XP. Em vez de escrever algum código e, em seguida, escrever testes para esse código, você escreve os testes antes de escrever o código. Isso significa que você pode executar o teste enquanto o código está sendo escrito e pode encontrar problemas durante o desenvolvimento.

Ao escrever os testes, implicitamente se definem uma interface e uma especificação de comportamento para a funcionalidade a ser desenvolvida. Problemas de requisitos e mal-entendidos de interface são reduzidos. Essa abordagem pode ser adotada em qualquer processo em que haja uma relação clara entre um requisito do sistema e o código que implementa esse requisito. Em XP, você sempre pode ver esse link, porque os cartões de estórias que representam os requisitos são divididos em tarefas, e essas são a principal unidade de implementação. A adoção do desenvolvimento *test-first* em XP gerou o desenvolvimento de abordagens mais gerais dirigidas a testes (ASTELS, 2003). No Capítulo 8, discuto essas abordagens.

No desenvolvimento *test-first*, os implementadores de tarefas precisam entender completamente a especificação para que possam escrever testes para o sistema. Isso significa que as ambiguidades e omissões da lista de especificações devem ser esclarecidas antes do início da implementação. Além disso, também evita o problema de '*test-lag*'. Isso pode acontecer quando o desenvolvedor do sistema trabalha em um ritmo mais rápido que o

testador. A implementação fica mais e mais à frente dos testes e desenvolve-se uma tendência a ignorar os testes, a fim de que o cronograma de desenvolvimento possa ser mantido.

Em XP, os requisitos do usuário são expressos como cenários ou estórias, e o usuário os prioriza para o desenvolvimento. A equipe de desenvolvimento avalia cada cenário e divide-o em tarefas. Por exemplo, alguns dos cartões de tarefas desenvolvidos a partir do cartão de estória para a prescrição de medicamentos (Quadro 3.1) são mostrados no Quadro 3.2. Cada tarefa gera um ou mais testes de unidade que verificam a implementação descrita naquela tarefa. O Quadro 3.3 é uma descrição resumida de um caso de teste desenvolvido para verificar se a dose prescrita de uma medicação não fica fora dos limites de segurança conhecidos.

No processo de testes, o papel do cliente é ajudar a desenvolver testes de aceitação para as estórias que serão implementadas no próximo *release* do sistema. Como discuto no Capítulo 8, o teste de aceitação é o processo em que o sistema é testado com os dados do cliente para verificar se o sistema atende às reais necessidades do cliente.

Em XP, o teste de aceitação, assim como o desenvolvimento, é incremental. O cliente, que faz parte da equipe, escreve os testes enquanto o desenvolvimento avança. Portanto, todos os novos códigos são validados para garantir que realmente é o que o cliente necessita. Para a estória no Quadro 3.1, o teste de aceitação implicaria cenários nos quais (a) a dose de um medicamento foi alterada, (b) um novo medicamento foi selecionado e (c) o formulário foi usado para encontrar um medicamento. Na prática, em vez de um único teste, uma série de testes de aceitação é normalmente necessária.

Uma grande dificuldade no processo de teste em XP é contar com o apoio do cliente no desenvolvimento de testes de aceitação. Clientes têm muito pouco tempo disponível e podem não conseguir trabalhar com a equipe de desenvolvimento em tempo integral. O cliente pode sentir que fornecer os requisitos seja uma contribuição suficiente e, dessa forma, pode estar relutante em se envolver no processo de testes.

Automação de testes é essencial para o desenvolvimento *test-first*. Os testes são escritos como componentes executáveis antes que a tarefa seja implementada. Esses componentes de teste devem ser autônomos, devem simular a submissão de entrada a ser testada e devem verificar se o resultado atende à especificação de saída. Um *framework* de testes automatizados é um sistema que torna mais fácil escrever os testes executáveis e submeter um conjunto de testes para execução. Junit (MASSOL e HUSTED, 2003) é um exemplo amplamente usado de *framework* de testes automatizados.

Como o teste é automatizado, há sempre um conjunto de testes que podem ser executados rapidamente e com facilidade. Sempre que qualquer funcionalidade é adicionada ao sistema, os testes podem ser executados e os problemas que o novo código introduziu podem ser detectados imediatamente.

Desenvolvimento *test-first* e teste automatizado, geralmente, resultam em um grande número de testes sendo escritos e executados. No entanto, essa abordagem não leva, necessariamente, a testes completos do programa. Existem três razões para isso:

1. Programadores preferem programar para testes e, por vezes, tomam atalhos ao escrevê-los. Por exemplo, eles talvez escrevam testes incompletos que não verificam todas as possíveis exceções que podem ocorrer.
2. Alguns testes podem ser muito difíceis de escrever de forma incremental. Por exemplo, em uma interface complexa de usuário, muitas vezes é difícil escrever testes unitários para o código que implementa a 'lógica de exibição' e o *workflow* entre as telas.

Quadro 3.3 Descrição do caso de teste para verificação de dose

Teste 4: Verificação de dose
Entrada: 1. Um número em mg representando uma única dose da medicação. 2. Um número que representa o número de doses únicas por dia. **Testes:** 1. Teste para entradas em que a dose única é correta, mas a frequência é muito alta. 2. Teste para entradas em que a única dose é muito alta e muito baixa. 3. Teste para entradas em que a dose única x frequência é muito alta e muito baixa. 4. Teste para entradas em que a dose única x frequência é permitida. **Saída:** Mensagem de OK ou erro indicando que a dose está fora da faixa de segurança.

3. É difícil julgar a completude de um conjunto de testes. Embora você possa ter vários testes do sistema, o conjunto pode não fornecer uma cobertura completa. Partes essenciais do sistema podem não ser executadas e, assim, permanecer não testadas.

Portanto, apesar de um grande conjunto de testes executados frequentemente dar a impressão de que o sistema está completo e correto, esse pode não ser o caso. Se os testes não são revistos e outros testes não são escritos após o desenvolvimento, defeitos não detectados podem ser entregues no *release* do sistema.

3.3.2 A programação em pares

Outra prática inovadora introduzida no XP é que, para desenvolver o software, os programadores trabalhem em pares. Na verdade, para desenvolver o software eles se sentam juntos, na mesma estação de trabalho. No entanto, os mesmos pares nem sempre programam juntos. Pelo contrário, os pares são criados de maneira dinâmica, de modo que todos os membros da equipe trabalhem uns com os outros durante o processo de desenvolvimento.

O uso da programação em pares tem uma série de vantagens:

1. Dá suporte à ideia de propriedade e responsabilidade coletiva para o sistema, o que reflete a ideia de programação sem ego, de Weinberg (1971), segundo a qual o software é de propriedade da equipe como um todo e os indivíduos não são responsabilizados por problemas com o código. Em vez disso, a equipe tem responsabilidade coletiva para resolver esses problemas.
2. Atua como um processo de revisão informal, porque cada linha de código é observada por, pelo menos, duas pessoas. Inspeções e revisões do código (abordadas no Capítulo 24) são muito bem-sucedidas em descobrir uma elevada porcentagem de erros de softwares. No entanto, são demoradas para organizar e costumam apresentar atrasos no processo de desenvolvimento. Embora a programação em pares seja um processo menos formal que provavelmente não encontra tantos erros como as inspeções de código, é um processo de inspeção muito mais barato do que inspeções formais de programa.
3. Dá suporte à refatoração, que é um processo de melhoria de software. A dificuldade de implementar isso em um ambiente de desenvolvimento normal é que o esforço despendido na refatoração é para um benefício a longo prazo. Um indivíduo que pratica a refatoração pode ser considerado menos eficiente do que aquele que simplesmente atua no desenvolvimento de código. Sempre que a programação em pares e a propriedade coletiva são usadas, outros se beneficiam imediatamente da refatoração para que eles possam apoiar o processo.

Você pode pensar que a programação em pares é menos eficiente do que a individual. Em um período determinado, um par de desenvolvedores produziria metade da quantidade de código que dois indivíduos trabalhando sozinhos. Houve vários estudos a respeito da produtividade de programadores pagos, com resultados diversos. Usando estudantes voluntários, Williams e seus colaboradores (COCKBURN e WILLIAMS, 2001; WILLIAMS et al., 2000) concluíram que a produtividade na programação em pares parece ser comparável com a de duas pessoas que trabalham de forma independente. As razões sugeridas para tanto são as de que os pares discutem o software antes do desenvolvimento, de modo que provavelmente têm menos falsos começos e menos retrabalho. Além disso, o número de erros evitados pela inspeção informal reduz o tempo gasto consertando defeitos descobertos durante o processo de teste.

No entanto, estudos com programadores mais experientes (ARISHOLM et al., 2007; PARRISH et al., 2004) não replicaram esses resultados. Os pesquisadores descobriram uma perda significativa de produtividade em comparação com dois programadores trabalhando sozinhos. Havia alguns benefícios na qualidade, mas estes não compensaram o *overhead* da programação em pares. No entanto, o compartilhamento de conhecimento que acontece durante a programação em pares é muito importante, pois reduz os riscos globais para um projeto quando da saída de membros da equipe. Por si só, esse aspecto pode fazer a programação em pares valer a pena.

3.4 Gerenciamento ágil de projetos

A principal responsabilidade dos gerentes de projeto de software é gerenciar o projeto para que o software seja entregue no prazo e dentro do orçamento previsto. Eles supervisionam o trabalho dos engenheiros de software e acompanham quão bem o desenvolvimento de software está progredindo.

A abordagem-padrão para gerenciamento de projetos é a dirigida a planos. Como discuto no Capítulo 23, os gerentes devem elaborar um plano para o projeto mostrando o que deve ser entregue, quando deve ser en-

tregue e quem vai trabalhar no desenvolvimento das entregas do projeto. Uma abordagem baseada em planos necessita de um gerente que tenha uma visão estável de tudo o que tem de ser desenvolvido e os processos de desenvolvimento. Contudo, essa abordagem não funciona bem com os métodos ágeis, nos quais os requisitos são desenvolvidos de forma incremental, o software é entregue em incrementos curtos e rápidos, e as mudanças nos requisitos e no software são a norma.

Como todos os outros processos profissionais de desenvolvimento de software, o desenvolvimento ágil tem de ser gerenciado de modo que se faça o melhor uso com o tempo e os recursos disponíveis para a equipe. Isso requer do gerenciamento de projeto uma abordagem diferente, adaptada para o desenvolvimento incremental e para os pontos fortes dos métodos ágeis.

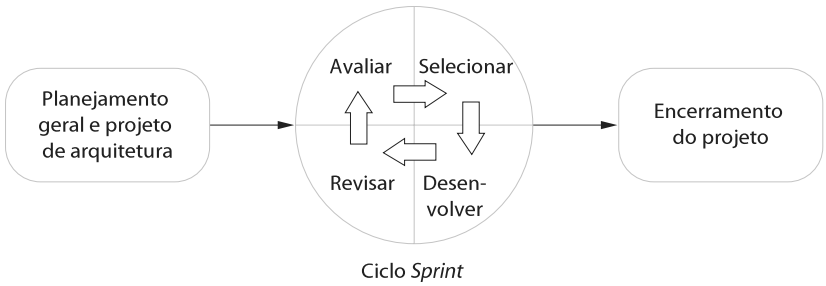
A abordagem Scrum (SCHWABER, 2004; SCHWABER e BEEDLE, 2001) é um método ágil geral, mas seu foco está no gerenciamento do desenvolvimento iterativo, ao invés das abordagens técnicas específicas da engenharia de software ágil. A Figura 3.3 é um diagrama do processo Scrum de gerenciamento. Scrum não prescreve o uso de práticas de programação, como programação em pares e desenvolvimento *test-first*. Portanto, pode ser usado com abordagens ágeis mais técnicas, como XP, para fornecer um *framework* de gerenciamento do projeto.

No Scrum, existem três fases. A primeira é uma fase de planejamento geral, em que se estabelecem os objetivos gerais do projeto e da arquitetura do software. Em seguida, ocorre uma série de ciclos de *sprint*, sendo que cada ciclo desenvolve um incremento do sistema. Finalmente, a última fase do projeto encerra o projeto, completa a documentação exigida, como quadros de ajuda do sistema e manuais do usuário, e avalia as lições aprendidas com o projeto.

A característica inovadora do Scrum é sua fase central, chamada ciclos de *sprint*. Um *sprint* do Scrum é uma unidade de planejamento na qual o trabalho a ser feito é avaliado, os recursos para o desenvolvimento são selecionados e o software é implementado. No fim de um *sprint*, a funcionalidade completa é entregue aos *stakeholders*. As principais características desse processo são:

1. *Sprints* são de comprimento fixo, normalmente duas a quatro semanas. Eles correspondem ao desenvolvimento de um *release* do sistema em XP.
2. O ponto de partida para o planejamento é o *backlog* do produto, que é a lista do trabalho a ser feito no projeto. Durante a fase de avaliação do *sprint*, este é revisto, e as prioridades e os riscos são identificados. O cliente está intimamente envolvido nesse processo e, no início de cada *sprint*, pode introduzir novos requisitos ou tarefas.
3. A fase de seleção envolve todos da equipe do projeto que trabalham com o cliente para selecionar os recursos e a funcionalidade a ser desenvolvida durante o *sprint*.
4. Uma vez que todos estejam de acordo, a equipe se organiza para desenvolver o software. Reuniões diárias rápidas, envolvendo todos os membros da equipe, são realizadas para analisar os progressos e, se necessário, repriorizar o trabalho. Nessa etapa, a equipe está isolada do cliente e da organização, com todas as comunicações canalizadas por meio do chamado 'Scrum Master'. O papel do Scrum Master é proteger a equipe de desenvolvimento de distrações externas. A maneira como o trabalho é desenvolvido depende do problema e da equipe. Diferentemente do XP, a abordagem Scrum não faz sugestões específicas sobre como escrever os requisitos ou sobre o desenvolvimento *test-first* etc. No entanto, essas práticas de XP podem ser usadas se a equipe achar que são adequadas.
5. No fim do *sprint*, o trabalho é revisto e apresentado aos *stakeholders*. O próximo ciclo *sprint* começa em seguida.

Figura 3.3 O processo Scrum



A ideia por trás do Scrum é que toda a equipe deve ter poderes para tomar decisões, de modo que o termo 'gerente de projeto' tem sido deliberadamente evitado. Pelo contrário, o 'Scrum Master' é um facilitador, que organiza reuniões diárias, controla o *backlog* de trabalho, registra decisões, mede o progresso comparado ao *backlog* e se comunica com os clientes e a gerência externa à equipe.

Toda a equipe participa das reuniões diárias; às vezes, estas são feitas com os participantes em pé ('*stand-up*'), muito rápidas, para a manutenção do foco da equipe. Durante a reunião, todos os membros da equipe compartilham informações, descrevem seu progresso desde a última reunião, os problemas que têm surgido e o que está planejado para o dia seguinte. Isso garante que todos na equipe saibam o que está acontecendo e, se surgirem problemas, poderão replanejar o trabalho de curto prazo para lidar com eles. Todos participam desse planejamento de curto prazo; não existe uma hierarquia *top-down* a partir do Scrum Master.

Existem, na Internet, muitos relatos de sucesso do uso de Scrum. Rising e Janoff (2000) discutem seu uso bem-sucedido em um ambiente de desenvolvimento de software de telecomunicações, e listam suas vantagens conforme a seguir:

1. O produto é decomposto em um conjunto de partes gerenciáveis e compreensíveis.
2. Requisitos instáveis não atrasam o progresso.
3. Toda a equipe tem visão de tudo, e, consequentemente, a comunicação da equipe é melhorada.
4. Os clientes veem a entrega de incrementos dentro do prazo e recebem *feedback* sobre como o produto funciona.
5. Estabelece-se confiança entre clientes e desenvolvedores e cria-se uma cultura positiva, na qual todo mundo espera que o projeto tenha êxito.

O Scrum, como originalmente concebido, foi projetado para uso de equipes colocalizadas, em que todos os membros poderiam se encontrar todos os dias em reuniões rápidas. No entanto, muito do desenvolvimento de software atual envolve equipes distribuídas, ou seja, com membros da equipe situados em diferentes lugares ao redor do mundo. Consequentemente, estão em curso várias experiências para desenvolvimento Scrum para ambientes de desenvolvimento distribuído (SMITS e PSHIGODA, 2007; SUTHERLAND et al., 2007).

3.5 Escalamento de métodos ágeis

Os métodos ágeis foram desenvolvidos para serem usados por equipes de programação de pequeno porte que podiam trabalhar juntas na mesma sala e se comunicar de maneira informal. Os métodos ágeis foram, portanto, usados principalmente para o desenvolvimento de sistemas de pequeno e médio porte. Naturalmente, a necessidade de acelerar a entrega de software, adequada às necessidades do cliente, também se aplica a sistemas maiores. Consequentemente, tem havido um grande interesse em escalamento dos métodos ágeis para lidar com sistemas maiores, desenvolvidos por grandes organizações.

Denning e colegas. (2008) argumentam que a única maneira de evitar problemas comuns da engenharia de software, como os sistemas que não atendem às necessidades dos clientes e estouros de orçamento, é encontrar maneiras de fazer os métodos ágeis trabalharem para grandes sistemas. Leffingwell (2007) discute quais práticas de desenvolvimento ágil escalam para sistemas de grande porte. Moore e Spens (2008) relatam sua experiência em usar uma abordagem ágil para desenvolver um grande sistema médico com 300 desenvolvedores trabalhando em equipes distribuídas geograficamente.

O desenvolvimento de sistemas de software de grande porte é diferente do de sistemas pequenos, em vários pontos, como vemos a seguir.

1. Sistemas de grande porte geralmente são coleções de sistemas separados que se comunicam, nos quais equipes separadas desenvolvem cada um dos sistemas. Frequentemente, essas equipes estão trabalhando em lugares diferentes e, por vezes, em diferentes fusos horários. É praticamente impossível que cada equipe tenha uma visão de todo o sistema. Consequentemente, suas prioridades costumam ser voltadas para completar sua parte do sistema, sem levar em conta questões mais amplas do sistema como um todo.
2. Sistemas de grande porte são '*brownfield systems*' (HOPKINS e JENKINS, 2008), isto é, incluem e interagem com inúmeros sistemas existentes. Muitos dos requisitos do sistema estão preocupados com essa interação; assim, realmente não se prestam à flexibilidade e desenvolvimento incremental. Questões políticas também podem ser importantes aqui. Muitas vezes, a solução mais fácil para um problema é mudar um sistema em vigor. No

entanto, isso requer uma negociação com os gerentes do sistema para convencê-los de que as mudanças podem ser implementadas sem risco para a operação do sistema.

3. Sempre que vários sistemas estão integrados para criar um único, uma fração significativa do desenvolvimento preocupa-se com a configuração do sistema e não com o desenvolvimento do código original. Isso não é necessariamente compatível com o desenvolvimento incremental e com a integração frequente de sistemas.
4. Sistemas de grande porte e seus processos de desenvolvimento são frequentemente restringidos pelas regras externas e regulamentos que limitam o desenvolvimento, que exigem certos tipos de documentação a ser produzida etc.
5. Sistemas de grande porte têm um longo tempo de aquisição e desenvolvimento. É difícil manter equipes coerentes que saibam sobre o sistema durante esse período, pois as pessoas, inevitavelmente, deslocam-se para outros trabalhos e projetos.
6. Sistemas de grande porte geralmente têm um conjunto diverso de *stakeholders*. Por exemplo, enfermeiros e administradores podem ser os usuários finais de um sistema médico, mas o pessoal médico sênior, gerentes de hospital etc. também são *stakeholders* do sistema. É praticamente impossível envolver, no processo de desenvolvimento, todos esses diferentes *stakeholders*.

Há duas perspectivas no escalamento de métodos ágeis:

1. Perspectiva '*scaling up*', relacionada ao uso desses métodos para desenvolver sistemas de software de grande porte que não podem ser desenvolvidos por uma equipe pequena.
2. Perspectiva '*scaling out*', relacionada com a forma como os métodos ágeis podem ser introduzidos em uma grande organização com muitos anos de experiência em desenvolvimento de software.

Métodos ágeis precisam ser adaptados para lidar com sistemas de engenharia de grande porte. Leffingwell (2007) argumenta que é essencial manter os fundamentos dos métodos ágeis — planejamento flexível, frequentes *releases* do sistema, integração contínua, desenvolvimento dirigido a testes e boa comunicação entre os membros da equipe. Eu acredito que as adaptações críticas que necessitam ser introduzidas são as seguintes:

1. Para o desenvolvimento de sistemas de grande porte, não é possível focar apenas no código do sistema. Você precisa fazer mais projeto adiantado e documentação do sistema. A arquitetura de software precisa ser projetada, e é necessário haver documentos produzidos para descrever os aspectos críticos do sistema, como esquemas de banco de dados, a divisão de trabalho entre as equipes etc.
2. Mecanismos de comunicação entre equipes precisam ser projetados e usados. Isso deve envolver telefonemas e videoconferências regulares entre os membros da equipe, bem como reuniões eletrônicas frequentes e curtas nas quais os membros das diversas equipes se atualizam sobre o progresso uns dos outros. Uma série de canais de comunicação, como e-mail, mensagens instantâneas, *wikis* e sistemas de redes sociais, deve ser fornecida para facilitar as comunicações.
3. A integração contínua, em que todo o sistema é construído toda vez que um desenvolvedor verifica uma mudança, é praticamente impossível quando vários programas distintos precisam ser integrados para criar o sistema. No entanto, é essencial manter construções frequentes e *releases* regulares. Isso pode significar a necessidade da introdução de novas ferramentas de gerenciamento de configuração para suporte ao desenvolvimento de software com multiequipes.

Pequenas empresas que desenvolvem produtos de software estão entre os adeptos mais entusiastas dos métodos ágeis. Essas empresas não são limitadas pelas burocracias organizacionais ou padrões de processos e podem mudar rapidamente para adotar novas ideias. Naturalmente, as grandes empresas também têm feito experiências com métodos ágeis em projetos específicos, mas é muito mais difícil para eles introduzirem ('*scale out*') esses métodos em toda a organização. Lindvall e colegas. (2004) discutem alguns dos problemas em introduzir métodos ágeis em quatro grandes empresas de tecnologia.

A introdução de métodos ágeis em grandes empresas é difícil por diversas razões:

1. Os gerentes de projeto que não têm experiência em métodos ágeis podem ser relutantes em aceitar o risco de uma nova abordagem, uma vez que não sabem como isso vai afetar seus projetos particulares.
2. Nas grandes organizações existem procedimentos e padrões de qualidade que todos os projetos devem seguir e, por causa de sua natureza burocrática, geralmente são incompatíveis com os métodos ágeis. Às vezes, recebem suporte de ferramentas de software (por exemplo, ferramentas de gerenciamento de requisitos), e o uso dessas ferramentas é obrigatório a todos os projetos.

3. Métodos ágeis parecem funcionar melhor quando os membros da equipe têm um nível relativamente alto de habilidade. No entanto, dentro das grandes organizações é possível encontrar uma ampla gama de habilidades e competências; além disso, pessoas com níveis menores de habilidade podem não se tornar membros efetivos da equipe em processos ágeis.
4. Pode haver resistência cultural aos métodos ágeis, principalmente em organizações com longa história de uso dos processos convencionais de engenharia de sistemas.

Os procedimentos de gerenciamento de mudanças e testes são exemplos de procedimentos normais das empresas que podem não ser compatíveis com os métodos ágeis. O gerenciamento de mudanças é o processo de controle das mudanças em um sistema, de modo que o impacto das mudanças seja previsível, e os custos, controlados. Todas as mudanças necessitam de aprovação prévia antes de serem feitas, o que entra em conflito com a noção de refatoração. No XP, qualquer desenvolvedor pode melhorar qualquer código sem a aprovação externa. Para sistemas de grande porte, existem também padrões para os testes, em que uma construção de sistema é entregue a uma equipe externa de teste. Esse procedimento pode entrar em conflito com as abordagens *test-first* e testes frequentes, usadas em XP.

Apresentar e sustentar o uso de métodos ágeis em uma grande organização é um processo de mudança cultural. Mudanças culturais levam muito tempo para serem implementadas e, em muitos casos, demandam uma mudança de gerenciamento para serem efetivadas. As empresas que desejam usar métodos ágeis precisam de evangelizadores para promoverem as mudanças. Elas devem dedicar recursos significativos para o processo de mudança. Quando este livro foi escrito, poucas grandes empresas conseguiram fazer uma transição bem-sucedida para o desenvolvimento ágil em toda a organização.

PONTOS IMPORTANTES

- Métodos ágeis são métodos de desenvolvimento incremental que se concentram em desenvolvimento rápido, *releases* frequentes do software, redução de *overheads* dos processos e produção de códigos de alta qualidade. Eles envolvem o cliente diretamente no processo de desenvolvimento.
- A decisão de usar uma abordagem ágil ou uma abordagem dirigida a planos para o desenvolvimento deve depender do tipo de software a ser desenvolvido, das habilidades da equipe de desenvolvimento e da cultura da empresa que desenvolve o sistema.
- Extreme Programming é um método ágil, bem conhecido, que integra um conjunto de boas práticas de programação, como *releases* frequentes do software, melhorias contínuas do software e participação do cliente na equipe de desenvolvimento.
- Um ponto forte da Extreme Programming é o desenvolvimento de testes automatizados antes da criação de um recurso do programa. Quando um incremento é integrado ao sistema, todos os testes devem ser executados com sucesso.
- O método Scrum é uma metodologia ágil que fornece um *framework* de gerenciamento de projetos. É centralizado em torno de um conjunto de *sprints*, que são períodos determinados de tempo, quando um incremento de sistema é desenvolvido. O planejamento é baseado na priorização de um *backlog* de trabalho e na seleção das tarefas mais importantes para um *sprint*.
- O escalamento de métodos ágeis para sistemas de grande porte é difícil. Tais sistemas necessitam de projeto adiantado e alguma documentação. A integração contínua é praticamente impossível quando existem várias equipes de desenvolvimento separadas trabalhando em um projeto.

LEITURA COMPLEMENTAR

Extreme Programming Explained. Esse foi o primeiro livro sobre XP, e talvez ainda seja o mais lido. Ele explica a abordagem a partir da perspectiva de um de seus inventores, e seu entusiasmo fica explícito no livro. (BECK, K. *Extreme Programming Explained*. Addison-Wesley, 2000.)

Get Ready for Agile Methods, With Care. Uma crítica cuidadosa aos métodos ágeis, que discute seus pontos fortes e fracos, escrita por um engenheiro de software muito experiente. (BOEHM, B. *IEEE Computer*, jan. 2002.)

Scaling Software Agility: Best Practices for Large Enterprises. Embora com foco em questões de escalamento de desenvolvimento ágil, esse livro inclui também um resumo dos principais métodos ágeis, como XP, Scrum e Crystal. (LEFFINGWELL, D. *Scaling Software Agility: Best Practices for Large Enterprises*. Addison-Wesley, 2007.)

Running an Agile Software Development Project. A maioria dos livros sobre métodos ágeis se concentra em um método específico, mas esse livro tem uma abordagem diferente e discute como colocar XP em prática em um projeto. A obra apresenta conselhos bons e práticos. (HOLCOMBE, M. *Running an Agile Software Development Project*. John Wiley and Sons, 2008.)

EXERCÍCIOS

3.1 Explique por que, para as empresas, a entrega rápida e implantação de novos sistemas frequentemente é mais importante do que a funcionalidade detalhada desses sistemas.

3.2 Explique como os princípios básicos dos métodos ágeis levam ao desenvolvimento e implantação de software acelerados.

3.3 Quando você não recomendaria o uso de um método ágil para o desenvolvimento de um sistema de software?

3.4 Extreme Programming expressa os requisitos dos usuários como histórias, com cada história escrita em um cartão. Discuta as vantagens e desvantagens dessa abordagem para a descrição de requisitos.

3.5 Explique por que o desenvolvimento *test-first* ajuda o programador a desenvolver um melhor entendimento dos requisitos do sistema. Quais são as potenciais dificuldades com o desenvolvimento *test-first*?

3.6 Sugira quatro razões pelas quais a taxa de produtividade de programadores que trabalham em pares pode ser mais que a metade da taxa de produtividade de dois programadores que trabalham individualmente.

3.7 Compare e contraste a abordagem Scrum para o gerenciamento de projetos com abordagens convencionais dirigida a planos, como discutido no Capítulo 23. As comparações devem ser baseadas na eficácia de cada abordagem para o planejamento da alocação das pessoas nos projetos, estimativa de custos de projetos, manutenção da coesão da equipe e gerenciamento de mudanças no quadro da equipe do projeto.

3.8 Você é um gerente de software em uma empresa que desenvolve softwares críticos de controles para aeronaves. Você é responsável pelo desenvolvimento de um sistema de apoio ao projeto de software que dá suporte para a tradução de requisitos de software em uma especificação formal de software (discutido no Capítulo 13). Comente sobre as vantagens e desvantagens das estratégias de desenvolvimento a seguir:

a) Coletar dos engenheiros de software e *stakeholders* externos (como a autoridade regulatória de certificação) os requisitos para um sistema desse tipo e desenvolver o sistema usando uma abordagem dirigida a planos.

b) Desenvolver um protótipo usando uma linguagem de *script*, como Ruby ou Python, avaliar esse protótipo com os engenheiros de software e outros *stakeholders* e, em seguida, revisar os requisitos do sistema. Desenvolver novamente o sistema final, usando Java.

c) Desenvolver o sistema em Java, usando uma abordagem ágil com um usuário envolvido na equipe de desenvolvimento.

3.9 Tem-se sugerido que um dos problemas de se ter um usuário participando de uma equipe de desenvolvimento de software é que eles 'se tornam nativos', ou seja, adotam a perspectiva da equipe de desenvolvimento e perdem de vista as necessidades de seus colegas usuários. Sugira três maneiras de evitar esse problema e discuta as vantagens e desvantagens de cada abordagem.

3.10 Para reduzir os custos e o impacto ambiental das viagens, sua empresa decide fechar uma série de escritórios e dar suporte ao pessoal para trabalhar em casa. No entanto, a gerência sênior que introduz essa política não está ciente de que o software é desenvolvido por métodos ágeis, que contam com equipe trabalhando no mesmo local, e a programação em pares. Discuta as dificuldades que essa nova política pode causar e como você poderia contornar esses problemas.

REFERÊNCIAS

AMBLER, S. W.; JEFFRIES, R. *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. Nova York: John Wiley & Sons, 2002.

ARISHOLM, E.; GALLIS, H.; DYBA, T.; SJOBERG, D. I. K. Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise. *IEEE Trans*. on Software Eng., v. 33, n. 2, 2007, p. 65-86.

ASTELS, D. *Test Driven Development: A Practical Guide*. Upper Saddle River, NJ: Prentice Hall, 2003.

BECK, K. Embracing Change with Extreme Programming. *IEEE Computer*, v. 32, n. 10, 1999, p. 70-78.

_____. *Extreme Programming Explained*. Reading, Mass.: Addison-Wesley, 2000.

CARLSON, D. *Eclipse Distilled*. Boston: Addison-Wesley, 2005.

COCKBURN, A. *Agile Software Development*. Reading, Mass.: Addison-Wesley, 2001.

_____. *Crystal Clear: A Human-Powered Methodology for Small Teams*. Boston: Addison-Wesley, 2004.

COCKBURN, A.; WILLIAMS, L. The costs and benefits of pair programming. In: Extreme Programming Examined. Boston: Addison-Wesley, 2001.

COHN, M. *Succeeding with Agile: Software Development Using Scrum*. Boston: Addison-Wesley, 2009.

DEMARCO, T.; BOEHM, B. The Agile Methods Fray. *IEEE Computer*, v. 35, n. 6, 2002, p. 90-92.

DENNING, P. J.; GUNDERSON, C.; HAYES-ROTH, R. Evolutionary System Development. *Comm. ACM*, v. 51, n. 12, 2008, p. 29-31.

DROBNA, J.; NOFTZ, D.; RAGHU, R. Piloting XP on Four Mission-Critical Projects. *IEEE Software*, v. 21, n. 6, 2004, p. 70-75.

HIGHSMITH, J. A. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. Nova York: Dorset House, 2000.

HOPKINS, R.; JENKINS, K. *Eating the IT Elephant: Moving from Greenfield Development to Brownfield*. Boston, Mass.: IBM Press, 2008.

LARMAN, C. *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and the Unified Process*. Englewood Cliff, NJ: Prentice Hall, 2002.

LEFFINGWELL, D. *Scaling Software Agility: Best Practices for Large Enterprises*. Boston: Addison-Wesley, 2007.

LINDVALL, M.; MUTHIG, D.; DAGNINO, A.; WALLIN, C.; STUPPERICH, M.; KIEFER, D. et al. Agile Software Development in Large Organizations. *IEEE Computer*, v. 37, n. 12, 2004, p. 26-34.

MARTIN, J. *Application Development Without Programmers*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

MASSOL, V.; HUSTED, T. *JUnit in Action*. Greenwich, Conn.: Manning Publications Co., 2003.

MILLS, H. D.; O'NEILL, D.; LINGER, R. C.; DYER, M.; QUINNAN, R. E. The Management of Software Engineering. *IBM Systems. J.*, .v 19, n. 4, 1980, p. 414-477.

MOORE, E.; SPENS, J. Scaling Agile: Finding your Agile Tribe. Proc. Agile 2008 Conference. *IEEE Computer Society*, Toronto, 2008, p. 121-124.

PALMER, S. R.; FELSING, J. M. *A Practical Guide to Feature-Driven Development*. Englewood Cliffs, NJ: Prentice Hall, 2002.

PARRISH, A.; SMITH, R.; HALE, D.; HALE, J. A Field Study of Developer Pairs: Productivity Impacts and Implications. *IEEE Software*, v. 21, n. 5, 2004, p. 76-79.

POOLE, C.; HUISMAN, J. W. Using Extreme Programming in a Maintenance Environment. *IEEE Software*, v. 18, n. 6, 2001, p. 42-50.

RISING, L.; JANOFF, N. S. The Scrum Software Development Process for Small Teams. *IEEE Software*, v. 17, n. 4, 2000, p. 26-32.

SCHWABER, K. *Agile Project Management with Scrum*. Seattle: Microsoft Press, 2004.

SCHWABER, K.; BEEDLE, M. *Agile Software Development with Scrum*. Englewood Cliffs, NJ: Prentice Hall, 2001.

SMITS, H.; PSHIGODA, G. Implementing Scrum in a Distributed Software Development Organization. *Agile 2007*. Washington, DC: IEEE Computer Society, 2007.

STAPLETON, J. *DSDM Dynamic Systems Development Method*. Harlow, Reino Unido: Addison-Wesley, 1997.

_____. *DSDM: Business Focused Development*, 2. ed. Harlow, Reino Unido: Pearson Education, 2003.

STEPHENS, M.; ROSENBERG, D. *Extreme Programming Refactored*. Berkley, Califórnia: Apress, 2003.

SUTHERLAND, J.; VIKTOROV, A.; BLOUNT, J.; PUNTIKOV, N. Distributed Scrum: Agile Project Management with Outsourced Development Teams. 40th Hawaii Int. Conf. on System Sciences. *IEEE Computer Society*, Hawaii, 2007.

WEINBERG, G. *The Psychology of Computer Programming*. Nova York: Van Nostrand, 1971.

WILLIAMS, L.; KESSLER, R. R.; CUNNINGHAM, W.; JEFFRIES, R. Strengthening the Case for Pair Programming. *IEEE Software*, v. 17, n. 4, 2000, p. 19-25.