Capítulo 2

Processos Ágeis de Desenvolvimento de Software

Márcio Amorim de Medeiros[1], Milton Moura Campos Neto[2]

Este capítulo discute sobre Processos Ágeis de desenvolvimento de software, uma nova abordagem de desenvolvimento, que surgiu como uma alternativa aos Processos Tradicionais na tentativa de reduzir os problemas e custos dos projetos de software. Ao longo deste capítulo será feita uma contextualização a respeito do paradigma ágil e enfatizado alguns processos como o *Extreme Programming* (XP), o *Scrum* e o *Feature Driven Development* (FDD).

2.1 Introdução

Os processos tradicionais de desenvolvimento de software não se adéquam à realidade de algumas organizações, em especial, as pequenas e médias fábricas de software que não possuem recursos suficientes para seguirem algum processo. Neste contexto, os processos ágeis surgiram como uma nova tendência de desenvolvimento para melhorar a qualidade dos sistemas e reduzir a quantidade de projetos fracassados, eliminando gastos com documentação excessiva, enfatizando a comunicação face-a-face, sendo mais flexíveis às mudanças e privilegiando as atividades que agregam maior valor ao negócio.

Os métodos tradicionais e os ágeis possuem o mesmo objetivo: satisfazer as necessidades dos usuários construindo sistemas de qualidade. A diferença entre eles está nos princípios utilizados por cada um [SATO 2007]. Os princípios relacionados aos processos tradicionais já foram abordados no Capítulo 1, já os ágeis serão detalhados no decorrer deste capítulo.

A adaptabilidade à mudança é um fator que deve está presente em um software, a fim de atender às novas necessidades dos clientes, das instituições ou do mercado. Os processos tradicionais tendem a planejar grande parte do desenvolvimento do software por um longo período antes de iniciar a implementação. Com isso, o software demora a ser disponibilizado ao cliente. Durante este tempo podem surgir novos padrões, políticas e tecnologias que afetam os requisitos do software, o cliente pode perceber que alguma

[1] mkamorim@gmail.com
[2] miltoncampospe@gmail.com

funcionalidade não está conforme solicitado ou precisar de outras. Estes fatores implicam em mudança no sistema, que não são bem-vindas nos processos tradicionais, pois a fase de planejamento já foi concluída.

Outro fator que pode acontecer no desenvolvimento tradicional é a implementação de funcionalidades que não agregarão valor ao cliente, ou seja, o sistema disponibiliza funcionalidades aos usuários que serão de pouca ou nenhuma utilidade, enquanto outras mais prioritárias ainda não foram implementadas.

Os processos ágeis surgiram com a finalidade de desburocratizar o processo de desenvolvimento de software. Eles tentam se adaptar e se fortalecer com as mudanças. Os clientes têm, em curto espaço de tempo, versões de software executável, nas quais são priorizadas as funcionalidades que agregam mais valor ao seu negócio. Com isso, eles já podem sugerir novas funcionalidades e correções nestas versões intermediárias.

Na agilidade, outro fator determinante é o fato de “não documentar, apenas por documentar”. Só é documentado aquilo que for necessário em outro momento e que justifique o esforço e recursos gastos na documentação. Segundo [BECK 2000], nos processos ágeis, a documentação se restringe às estórias dos usuários, que são descrições simples do funcionamento do sistema, usadas para ajudarem os envolvidos no projeto a terem uma visão de seu funcionamento e entenderem os elementos básicos do projeto e seus relacionamentos.

Os Processos ágeis são orientados a pessoas ao contrário dos tradicionais que são orientados a processos. Assim, processos ágeis afirmam que a habilidade da equipe de desenvolvimento é imprescindível durante o desenvolvimento de software. O papel do processo é, por sua vez, dar suporte à equipe durante o trabalho

2.2 O Manifesto Ágil

No início de 2001, 17 especialistas em software reuniram-se para propor um conjunto de princípios e valores para agilizar o desenvolvimento dos seus sistemas tendo como base suas experiências em programação. Tais princípios foram motivados pela conclusão de que os processos de desenvolvimento estavam tornando-se cada vez mais burocráticos, dificultando a visibilidade e o entendimento das equipes de construção de softwares.

A essência deste movimento em prol da agilidade foi a definição de um novo enfoque de desenvolvimento de software, calcado no ágil, na flexibilidade, nas habilidades de comunicação e na capacidade de oferecer novos produtos e serviços de valor ao mercado, em curtos períodos de tempo. [HIGHSMITH 2004]

Como produto deste movimento, marco inicial do desenvolvimento ágil de software, foi produzido o Manifesto Ágil conforme ilustrado no Quadro 2.1

Quadro 2.1 O Manifesto Ágil. Adaptado de [AGILE MANIFESTO 2009]

Nós estamos descobrindo melhores maneiras de desenvolver software, fazendo e ajudando outros a fazê-lo. Através deste trabalho, passamos a valorizar:

Indivíduos e interação entre eles mais que processos e ferramentas

Software em funcionamento mais que documentação abrangente

Colaboração com o cliente mais que negociação de contratos

Responder a mudanças mais que seguir um plano

Mesmo havendo valor nos itens à direita, valorizamos mais os itens à esquerda.

***Assinam este manifesto*:**

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland e Dave Thomas.

Os envolvidos na concepção do Manifesto Ágil se autodenominaram de Aliança Ágil. Os quatro valores propostos no manifesto foram então traduzidos em doze princípios que estão descritos a seguir:

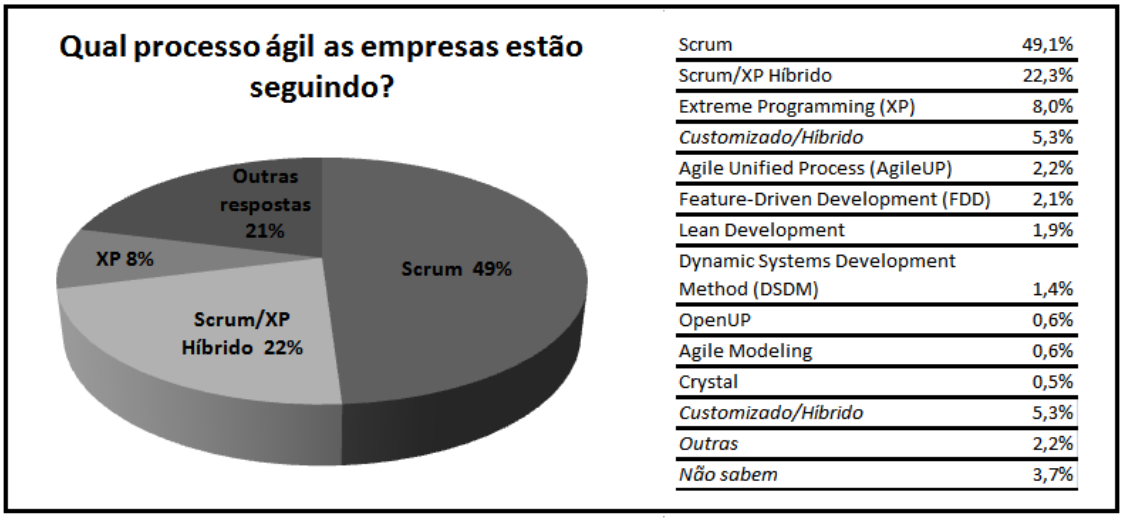
- A maior prioridade é satisfazer o cliente através de entregas antecipadas e contínuas de software de valor;
- Mudanças de requisitos são bem vindas, mesmo que tardiamente no desenvolvimento. Processos ágeis tiram proveito das mudanças visando vantagem competitiva para o cliente;
- Entrega frequente de software funcionando, em intervalos de duas semanas de trabalho até dois meses, com preferência a escala de tempo mais curta;
- Pessoas das áreas de software e de negócios devem trabalhar juntas diariamente durante o desenvolvimento do projeto;
- Construir projetos em torno de indivíduos motivados. Dê-los o ambiente e o suporte necessário e acredite neles para fazer o trabalho;
- O Método mais eficiente e efetivo de repassar a informação dentro de uma equipe de desenvolvimento é a conversa face a face;
- Software funcionando é a primeira medida de progresso;
- Processos ágeis promovem desenvolvimento sustentável. Os patrocinadores, desenvolvedores e usuários devem ser capazes de manter um passo constante indefinidamente;
- Atenção contínua à excelência técnica e um bom projeto melhoram a agilidade;
- Simplicidade: a arte de maximizar a quantidade de trabalho não realizado, é essencial;

- As melhores arquiteturas, requisitos, e projetos emergem de times auto-organizáveis;
- Em intervalos regulares, o time deve refletir sobre como se tornar mais efetivo, então melhora e ajusta seu comportamento de acordo com a reflexão.

2.3 Principais Processos Ágeis

Todos os processos de desenvolvimento de software que seguem os princípios do Manifesto Ágil, tais como: entrega frequente, flexão a mudança, foco em pessoas e simplicidade, são considerados processos ágeis.

Os processos ágeis ganham mais adeptos à medida que estão evoluindo com novas técnicas e adaptações. A 3ª pesquisa sobre o estado do Desenvolvimento Ágil promovido pela VersionOne [VERSIONONE 2008], e aplicada a mais de três mil entrevistados de 80 países, revela entre diversos indicadores, quais são os processos ágeis mais utilizados nas empresas, conforme mostra a Figura 2.1.



Scrum	49,1%
Scrum/XP Híbrido	22,3%
Extreme Programming (XP)	8,0%
Customizado/Híbrido	5,3%
Agile Unified Process (AgileUP)	2,2%
Feature-Driven Development (FDD)	2,1%
Lean Development	1,9%
Dynamic Systems Development Method (DSDM)	1,4%
OpenUP	0,6%
Agile Modeling	0,6%
Crystal	0,5%
Customizado/Híbrido	5,3%
Outras	2,2%
Não sabem	3,7%

Figura 2.1 Processos Ágeis mais utilizados nas empresas de acordo com a 3° Pesquisa Anual sobre o Estado do Desenvolvimento Ágil – Ano 2008. Adaptado de [VERSIONONE 2008].

O processo Agile Unified Process (AUP), também conhecido como AgileUP e desenvolvido por Scott Ambler [AMBLER 2002], é uma versão ágil e simplificada do Rational Unified Process (RUP). Ele descreve como desenvolver sistemas utilizando técnicas ágeis, como Test Driven Development (TDD), Agile Modeling e Refactoring, para melhorar a produtividade, porém mantendo-se fiel ao RUP.

O processo *Lean Development* tem suas raízes na indústria automotiva, ele é uma adaptação para software do *Lean Manufacturing* do revolucionário Sistema Toyota de Produção. Inicialmente proposto por Bob Charette [CHARETTE 2002], tem como principais objetivos tentar reduzir em um terço o prazo, o custo e o nível de defeito no desenvolvimento de software e para isso exige um grande comprometimento da alta administração com predisposição inclusive para mudanças radicais.

Originado no Desenvolvimento Rápido de Aplicações (RAD) e no modelo iterativo e incremental [STAPLETON 2003], o processo *Dynamic Systems Development Method* (DSDM), baseia-se no seguinte: ao invés de fixar o escopo de funcionalidades do produto e a partir daí estimar tempo e recursos para alcançar o

escopo definido, é preferível fixar tempo e recursos e ajustar o escopo de acordo com estas limitações [COHEN et al. 2003].

O *Crystal* não é apenas um único processo, e sim, uma família de métodos denominada como Família *Crystal*, proposto por Alistair Cockburn [COCKBURN 2002]. Cada um dos processos da família é denominado por cores de acordo com a quantidade de pessoas envolvidas no projeto. A versão mais ágil e mais documentada é a *Crystal Clear* que pode ser usada em projetos de até oito pessoas, seguida pela *Crystal Yellow* (para times de 8 a 20 pessoas), *Crystal Orange* (para times de 20 até 50 pessoas), *Crystal Red* (para times de 50 até 100 pessoas).

Já o processo *Agile Modeling* (AM) tem foco apenas na modelagem ágil, não contemplando outras etapas do ciclo de vida do software. Isto é, AM visa construir e manter modelos de sistemas de maneira eficaz e eficiente e pode ser utilizado tanto em conjunto com processos ágeis como em conjunto com processos tradicionais.

Os processos ágeis que se destacam no mercado são o *Extreme Programming* (XP), que se atém ao desenvolvimento propondo um conjunto de técnicas, e o *Scrum*, que enfatiza o planejamento e o gerenciamento dos projetos. Ambos serão tratados com maior detalhe nas próximas seções, além do processo FDD que é uma boa opção para grandes projetos e para empresas que possuem dificuldades para migrar para um ambiente completamente ágil.

2.4 Extreme Programming

O *Extreme Programming* (XP) é um processo de desenvolvimento de software ágil, criado oficialmente em 1999 por Kent Beck e Ward Cunningham, quando do lançamento do livro *Extreme Programming Explained* que formalizou e difundiu o referido processo. As experiências dos dois, que culminaram no XP, vêm desde o início da década de 90 com o desenvolvimento na linguagem *Smalltalk* [SMALLTALK 2009].

O processo XP é voltado ao desenvolvimento de produtos com requisitos não totalmente definidos e em constante mudança [BECK 2004]. O processo é apoiado em valores, princípios e práticas, cujo foco é o desenvolvimento de um produto que venha a atender os objetivos do cliente. Alguns adeptos do XP o definem como sendo a prática e a perseguição da mais clara simplicidade, aplicado ao desenvolvimento de *software* [TELLES 2004].

O termo "*Extreme*" referencia o emprego de forma extrema das boas práticas da engenharia de software, além de suas práticas peculiares que diferenciam XP de outros processos ágeis, como a programação em pares, forte cultura de testes, propriedade coletiva de código entre outras que serão abordadas adiante.

Segundo Kent Beck (2004), o XP inclui em seu arcabouço: uma filosofia de desenvolvimento de *software* baseada em valores (comunicação, *feedback*, simplicidade, coragem e respeito); um conjunto de práticas, que expressam os valores, comprovadamente úteis para melhorar o desenvolvimento de software; um conjunto complementar de princípios e técnicas que auxiliam a tradução dos valores em práticas; e uma comunidade que compartilha dos mesmos princípios e muitas das mesmas práticas.

Nas seções a seguir serão detalhados estes valores, princípios e práticas do XP, com base na abordagem descrita na segunda edição do livro *Extreme Programming Explained*, publicado em 2004.

2.4.1 Valores de XP

XP apresenta os seguintes valores que descrevem os objetivos e definem critérios para se obter sucesso no desenvolvimento de um projeto de *software*:

- **Comunicação:** o fator de sucesso e insucesso de projetos de *software* é atribuído em grande parte à qualidade da comunicação. XP inova e ousa ao priorizar a comunicação pessoal e oral ao invés da escrita. O contato presencial, por meio de sinais sutis e linguagem corporal, possibilita um enriquecimento da comunicação, além permitir que dúvidas sejam discutidas e resolvidas logo que surgem. Já a documentação escrita sempre tende a desatualizar-se rapidamente.
- **Simplicidade:** XP recomenda manter o sistema simples de forma a não gerar mais trabalho desnecessário. Os desenvolvedores devem pensar no presente, não generalizar sem necessidade e nem supor necessidades, bem como estarem alertas a oportunidades de refatorar[3] o *software* com o objetivo de simplificá-lo.
- ***Feedback*:** para poder mudar o plano e se adaptar, precisa-se saber rapidamente e com exatidão o que está acontecendo. Ao longo do desenvolvimento, é muito importante que exista *feedback* dentro do time de desenvolvimento e com o cliente e/ou demais envolvidos, para avaliar se as necessidades acordadas estão sendo atendidas. As entregas frequentes do *software* funcionando, por exemplo, possibilitam que o cliente entenda efetivamente o que precisa, mude de ideia, ou descubra requisitos dos quais não estava ciente.
- **Coragem:** quando se quer iniciar o desenvolvimento de um *software*, muitos são os medos envolvidos. XP combate estes medos ao fornecer o suporte necessário para que os envolvidos possam sentir coragem para agir e tomar decisões. O time pode ter coragem de refatorar, pois sabe que os testes irão detectar erros imediatamente. O cliente pode decidir o escopo com mais coragem, quando avalia o *software* funcional após cada *release*, sabendo que pode priorizar as funcionalidades que lhe são mais importantes.
- **Respeito:** valorizar a relação entre os membros da equipe e, também, a de cada membro com o projeto e sua instituição. O valor respeito atua como um alicerce para todos os outros valores, por isso ele sempre deve estar presente.

2.4.2 Princípios de XP

Os princípios agem como guias, específicos ao domínio da programação, para identificar práticas concretas em harmonia com os valores abstratos. Deste modo, Kent Beck elenca os seguintes princípios de XP:

- **Humanidade:** reconhece que são pessoas com necessidades humanas que desenvolvem *software*. Esse princípio configura-se através da promoção, ao time, de aspectos como: segurança, crescimento, identidade com o grupo,

[3] Refatorar é melhorar o projeto do código após este ter sido escrito, sem alterar o comportamento do software.

realização, intimidade e privacidade entre outras. Este princípio é concretizado, na prática de trabalho energizado descrita na seção a seguir. Um dos desafios propostos por este princípio é balancear as necessidades individuais com as da equipe. Espera-se que os envolvidos no processo de desenvolvimento estejam dispostos a perceber e atuar nesse sentido.

- **Economia:** busca garantir valor para o negócio. Ao economizar pensamos sobre o valor do dinheiro ao longo do tempo e como melhor empregá-lo. É importante receber o mais cedo possível e gastar o mais tarde possível. É importante também refletir sobre o valor de opções que podemos tomar pela equipe e pelo sistema, percebendo que a habilidade de poder mudar de idéia no futuro deve guiar nossas decisões. Este princípio é evidenciado nas práticas de *design* incremental, pague pelo uso e também na priorização e estimativa de estórias.
- **Benefício mútuo:** a partir da ênfase que o processo XP emprega em suas práticas e em procedimentos como testes, refatorações e metáfora, é possível uma redução substancial da documentação escrita, a eliminação do retrabalho e facilitação do entendimento do problema pela equipe e pelo cliente.
- **Auto-semelhança:** permite que uma solução seja aplicada a diferentes contextos e até mesmo em projetos de diferentes escalas. Isto é permitido por meio do ritmo similar que se observa nas práticas do ciclo de estação, ciclo semanal e nas atividades diárias de programação e das observações feitas pela equipe
- **Melhoria:** indica que não devemos esperar a perfeição, mas sim fazer o melhor que podemos hoje, para poder fazer o melhor amanhã. A prática de ciclo de estação evidencia este princípio dando à equipe a oportunidade de melhorar o plano de um *release*. A prática de *design* incremental também segue o princípio da melhoria.
- **Diversidade:** lembra que um time com pessoas diferentes apresenta mais habilidades, conhecimentos e oportunidades. A diversidade é causa de conflitos, sendo importante lidar com essa possibilidade valorizando o respeito. O princípio é concretizado nas práticas do time completo e nos diferentes ciclos de planejamento. Pessoas com perspectivas diversas têm igual oportunidade de colaborar: aquelas que pensam em longo prazo contribuindo com o ciclo de estação e as que têm perspectivas de curto prazo contribuindo com o ciclo semanal.
- **Reflexão:** implica em pensar sobre como e por que trabalhamos. Este princípio pode guiar uma equipe a adotar práticas como a retrospectiva e a realizar análises frequentes do seu processo de adoção do processo. Para isso, é preciso tempo para pensar. É importante socializar-se com a equipe em contextos de diversão ou até em refeições. A reflexão é evidenciada nas práticas do ciclo de estação e ciclo semanal, nas conversas de pares programando e na prática de integração contínua.
- **Fluxo:** determina que exista uma corrente contínua de atividades e que o processo deve explicitá-la. Desta maneira permite-se que as etapas do desenvolvimento aconteçam em paralelo e não sequencialmente, como proposto em processos mais tradicionais. As práticas de *release* frequentes e integração contínua evidenciam isto.

- **Oportunidade:** leva a encarar problemas como oportunidades para mudança. Estar aberto a oportunidades de aprender e melhorar durante todo o processo é importante.
- **Redundância:** quando usada de forma positiva, aumenta as chances de sucesso, promovendo várias oportunidades de fazer a coisa certa. A redundância está presente, por exemplo, nos testes de unidade automatizados: escreve-se o código fonte e, de maneira redundante, escrevemos mais código para verificar se o primeiro funciona, diminuindo a probabilidade de errar.
- **Falha:** indica que pode ser bom falhar, desde que se aprenda com a experiência. Quando uma equipe não sabe para onde ir, arriscar-se a falhar pode ser o caminho mais curto para obter o sucesso. Este princípio é complementar ao valor de coragem e se evidencia na prática de abandonar código e começar de novo quando percebemos que determinado plano não poderá ser realizado.
- **Qualidade:** sempre presente e em alta. Este princípio diz que quanto maior a qualidade, mais fácil será realizar o trabalho. Ele complementa o princípio da humanidade ao satisfazer a necessidade de se orgulhar do trabalho feito e é evidenciado na prática de controle do escopo no planejamento.
- **Passos pequenos:** garante que iremos fazer sempre o caminho mais curto na direção correta, pois a execução de tarefas complexas em passos pequenos diminui o risco. A prática de *design* incremental, integração contínua, implantação incremental e implantação diária refletem este princípio.
- **Aceitação de Responsabilidade:** evidencia que só o próprio indivíduo pode se responsabilizar por suas ações. Este princípio está claro na prática de estimativas feitas pelos próprios programadores no jogo do planejamento.

2.4.3 Práticas de XP

XP define também práticas que tornam o processo viável e possível de seguir seus valores e princípios. As práticas são simples, porém o poder deste processo provém da combinação delas. As práticas abordadas a seguir representam uma evolução em relação à primeira versão do XP. Nessa versão, Kent Beck (2004) as categoriza em dois tipos – primária e corolário, a saber:

As práticas primárias são guias para se começar a adoção de XP em uma organização. Elas podem ser implantadas facilmente, pois são seguras e devem ser introduzidas em pequenos passos, para evitar uma mudança muito rápida na cultura da organização. É importante ter tempo para que os novos hábitos sejam incorporados pela equipe. A seguir são apresentados detalhes de cada uma dessas práticas:

- **Sentar Juntos:** deixa explícita a necessidade de se ter um espaço onde toda a equipe possa trabalhar junta, valorizando a comunicação e possibilitando que as pessoas possam beneficiar-se de todos os seus sentidos ao conversar. Ressalta também a necessidade de pequenos espaços privativos para respeitar o princípio de humanidade.
- **Time Completo:** a equipe precisa de pessoas com todas as habilidades e perspectivas necessárias para o sucesso do projeto. As equipes em sua composição devem ser pequenas (cerca de 12 pessoas), podendo haver um grupo

com maior número de membros, desde que organizados em uma estrutura de pequenos grupos, cujo processo atuaria em hierarquias. Pode ser formado um grupo de cerca de 120 pessoas divido em grupos de 12 (atendendo o processo), no qual o processo XP seria aplicado e numa esfera superior (10 membros, sendo 01 de cada grupo) e nos grupos individualmente.

- **Espaço de trabalho informativo:** essa prática diz que qualquer observador interessado deve ser capaz de olhar para o espaço de trabalho e ter idéia do andamento do projeto em pouco tempo. O espaço informativo deve conter cartazes grandes e visíveis, que comunicam medidas coletadas pelo acompanhador. Limpeza, ordem, espaço para programação em par e disponibilidade de água e comida garantem que o espaço informativo complemente a prática de trabalho energizado.
- **Trabalho Energizado:** Com tempo para se dedicar a satisfazer suas outras necessidades fora do trabalho, os seres humanos podem voltar com mais energia e mais dedicação. Por outro lado, a equipe deve ter mantida a sua motivação e dispor de um ambiente que propicie os trabalhos atendendo e equilibrando aspectos como: segurança, crescimento pessoal, interação na equipe e o tempo factível para a realização das atividades.
- **Estórias:** São textos simples que expressam necessidades do cliente e que têm suas estimativas de tempo para desenvolvimento feitas, o quanto antes, pelos programadores.
- **Ciclo semanal:** explicita as iterações que fazem parte de um *release*. O ciclo semanal inclui um jogo de planejamento do trabalho que deve ser efetuado em aproximadamente uma semana, ou seja, planeja as iterações de menor granularidade. Uma reunião no começo da semana revê o progresso de um *release*, comparando o que foi feito com o que tinha sido planejado na semana anterior. O cliente prioriza uma semana de estórias, que são divididas em tarefas, que por sua vez são aceitas e estimadas pelos programadores. A semana começa com a escrita de testes de aceitação da iteração e acaba com a implantação do sistema codificado.
- **Ciclo de estação:** explicita o planejamento de um *release*. A avaliação de um ciclo de estação pode identificar gargalos e dependências da equipe com outras partes da organização, apresentando uma oportunidade para melhorias. Durante o planejamento deste ciclo, temas são escolhidos para a estação. Estes temas servem para agrupar estórias relacionadas, que também são criadas neste momento. O enfoque do planejamento da estação é também perceber como o projeto se encaixa com o resto da organização. A estação pode ter duração variável e depende do contexto do negócio. Em muitas organizações um trimestre é uma boa medida para avaliar o progresso.
- **Folga:** reconhece que por melhor que tenha sido o planejamento, o mesmo sempre falha. Para evitar atrasos ou a necessidade de renegociação de escopo, o planejamento deve conter explicitamente espaços de folga. Se o progresso de um ciclo for bom, este tempo pode ser usado para pesquisa e para manter o trabalho energizado. Caso atrasos ocorram, a folga pode ser utilizada para que a equipe consiga entregar as estórias que prometeu ao cliente.

- ***Build* veloz:** exige que o sistema seja compilado por completo e todos os testes sejam executados, de maneira automática, em no máximo 10 minutos. Esta prática provê agilidade à equipe e complementa a habilidade de entregar *releases* pequenos. O limite de 10 minutos é somente o tempo razoável para que o par possa tomar um café durante o *build*. Se o build demora menos que 10 minutos, excelente, o café pode ficar para depois. Se demora mais existem duas possibilidades. A primeira é de que o *build* pode ser refatorado para que não execute todos os testes de aceitação, por exemplo, pois estes podem demorar muito tempo para serem executados. A segunda é de que o build é lento, pois o sistema é muito complexo, e a demora pode ser um indício da necessidade de simplificá-lo.
- ***Design* incremental:** a prática de *design* simples, presente na primeira versão de XP, rendeu criticas por parte dos que diziam que o processo não investia em *design* da aplicação. Para responder às críticas, Beck (2004) a redefiniu como *design* incremental. Esta prática implica em investimento diário no *design* da aplicação, observando o seu fluxo contínuamente para possíveis ajustes ou para o entendimento do mesmo.
- **Programação em pares:** todo o código deve ser produzido por pares de programadores, cada par trabalhando na mesma máquina.
- **Integração contínua:** o código é integrado, o *build* do *software* é gerado e os testes são executados várias vezes ao dia.

As práticas corolário são difíceis ou perigosas de serem implementadas sem que antes sejam dominadas as práticas primárias. A prática do **cliente sempre presente**, criada na primeira versão de XP, foi renomeada para **envolvimento real com o cliente** e assume que muitas vezes um cliente direto não poderá ser colocado junto com a equipe. A prática de **propriedade coletiva do código**, presente na primeira versão de XP, foi renomeada para **código compartilhado** e se mantém como prática corolário. Além dessas duas práticas, já presentes na primeira versão de XP, foram adicionadas novas práticas corolário.

- **Implantação incremental:** caracterizada pelo desenvolvimento de software feito de maneira gradativa e, também, pela susbtituição de sistemas existentes gradualmente, de maneira que a implantação gere pouco ou nenhum impacto para o cliente e os usuários.
- **Continuidade da equipe:** propõe que equipes eficientes continuem trabalhando juntas. Deve-se incentivar um rodízio razoável entre equipes, mas que não coloque em risco a integração, harmonia e sua eficiência.
- **Redução da equipe:** com o passar do tempo, a necessidade de uma equipe grande pode diminuir, principalmente se o sistema entra em um ciclo de manutenção. Se isto acontecer, mantenha a carga de trabalho constante e distribua tarefas de modo a não permitir ociosidade. Alguns membros podem ser direcionados a compor novos times.
- **Análise da causa inicial:** sempre que encontrar um defeito, elimine o defeito e sua causa, para que o mesmo tipo de erro não ocorra novamente. Esta prática define uma nova atividade: ao encontrar um defeito, um par deve primeiro

escrever um teste de aceitação automatizado que evidencie o erro no nível do sistema e então escrever um teste de unidade no menor escopo possível. O par deve proceder para resolver o problema e o código reparado deve passar nos testes. O último passo é descobrir por que o defeito surgiu e, principalmente, como ele passou despercebido. A técnica de análise da causa inicial propõe que se pergunte 5 vezes o motivo do defeito ter surgido e sugere que a causa inicial, na maioria das vezes, é um problema relacionado à equipe e que pode ser resolvido utilizando- se práticas que evitem a sua recorrência.

- **Código e testes:** explicita que só o código fonte e os testes automatizados devem ser artefatos permanentes gerados por uma equipe XP. Até mesmo as estórias e cartazes devem ser descartados, pois o histórico do projeto se mantém pelo código e testes e respectivos comentários.
- **Repositório único de código:** complementa a prática de **código compartilhado** e vai além, dizendo que todo o código deve estar contido em um único repositório e que não deve haver *branches* permanentes para atualização do código. Se o seu sistema é tão complexo que precisa de repositórios separados, isso é evidência de um problema no *design*.
- **Implantação diária:** é a evolução da prática de ***releases* pequenos** e é ainda mais extrema. Ela determina que o seu sistema deve ser implantado diariamente, de preferência com auxílio do ***build* veloz**. O objetivo é colocar estórias implementadas em produção toda noite, para que os usuários possam usufruir de benefícios o quanto antes. Esta prática depende de uma baixa taxa de defeitos e de um processo automático de implantação, com habilidade para implantações incrementais e eventuais *rollbacks* em caso de falhas.
- **Contrato de escopo negociável:** determina como devem ser feitos contratos em um projeto que adota XP, fixando o tempo, custos e a qualidade, mas mantendo o escopo negociável. Desta maneira, o escopo é negociado constantemente com o cliente, possivelmente no planejamento do ciclo de estação. Assim a equipe poderá celebrar uma sequência de contratos curtos.
- **Pague pelo uso:** sugere que o cliente possa optar por pagar pelo sistema à medida que os releases são entregues. Nesse caso, o cliente só paga pelo que foi entregue e está em funcionamento.

A Tabela 2.1 apresenta um mapeamento e vinculação entre práticas da primeira e da segunda versão de XP, não havendo substituição de uma pela outra, e sim uma atualização com fins de melhorar o entendimento e a aplicação do processo.

Tabela 2.1 Práticas do XP 1ª versão vs 2ª versão

Práticas do Extreme Programming		
1ª VERSÃO 1999	2ª VERSÃO 2004	Categoria 2ªv
O jogo do planejamento	Estórias Ciclo semanal Ciclo de estação	Primária Primária Primária
Releases pequenos	Implementação incremental Implementação diária	Corolário Corolário

Metáforas Projeto de *software* simples Refatoração	*Design* incremental	Primária
Teste	Desenvolvimento orientado por testes Código e testes	Primária Corolário
Programação em pares	Programação Pareada	Primária
Propriedade coletiva Padrão de codificação Integração contínua	Código compartilhado Repositório único de código Integração contínua	Corolário Corolário Primária
-	Área de trabalho informativa	Primária
-	Análise de causa inicial	Corolário
40 horas semanais	Folga Trabalho energizado	Primária Primária
Cliente no local	Sentar junto Time completo Envolvimento real com o cliente	Primária Primária Corolário
-	Continuidade do time	Corolário
-	Redução do time	Corolário
-	Build de 10 minutos	Primária
-	Contrato de escopo variável	Corolário
-	Pague pelo uso	Corolário

2.4.4 Papéis no XP

Nesta nova versão, Beck (2004) inclui todas as funções que tipicamente se encontram em uma organização que desenvolve *software* e explicita quais são os papéis que cada uma pode assumir para colaborar com uma equipe XP. A diversidade de papéis traz benefício à prática de **time completo**. Cada parte no grupo deve entender seu papel no todo e a interação entre as pessoas deve seguir os princípios de **fluxo** e **benefício mútuo**. Uma pessoa pode assumir mais de um papel e os papéis podem ser revezados entre as pessoas. O XP sugere que o time seja multidisciplinar com habilidades necessárias para realizar o projeto. A primeira versão de XP era mais voltada aos programadores enquanto na segunda versão é dado maior valor a todos os outros papéis dentro da equipe. Assim, os principais papéis em XP são [BECK 2004]:

- **Programadores:** este é o coração de XP. Responsável por quebrar estórias e tarefas, escrever testes e código e automatizar processos manuais. Existem dois papéis especiais para programadores, aqueles com mais experiência atuam com *Coach* ou líderes de equipe que auxiliam os menos experientes da equipe, enquanto o *Tracker* atua coletando e compartilhando dados sobre o projeto e do processo.
- **Arquitetos:** executam refatoração de larga escala, escrevem testes de carga automatizados para definir cenários de estresse e auxiliam os programadores no particionamento do sistema mantendo a ênfase no projeto de alto-nível.
- **Analista de Testes:** trabalham junto com clientes e analistas de negócios para escrever testes de aceitação automatizados definindo cenários de sucesso e falha

em cada estória. Também treinam os programadores a utilizarem as ferramentas de teste.

- **Analista de Negócios:** trabalham com clientes para definir as estórias do sistema e auxiliam os programadores a definir o valor da importância das mesmas.
- **Projetistas de Interação:** avaliam o modo como o sistema está sendo utilizado pelos usuários finais, assim podem ser levantadas novas estórias bem como propostas de melhorias na interface gráfica.
- **Gerente de Projetos:** facilitam a comunicação dentro da equipe, removendo impedimentos e coordenando a comunicação com as pessoas externas à equipe.
- **Gerente do Produto:** escrevem e priorizam estórias do ciclo semanal e fecham o tema para o ciclo trimestral. Encorajam a comunicação entre a equipe de desenvolvimento e o cliente para que suas necessidades mais urgentes sejam atendidas de imediato.
- **Usuários:** Por utilizarem o sistema diariamente, ajudam o time a escrever e escolher de forma melhor as estórias do sistema. Por fornecerem as necessidades do sistema, é ideal que tenham experiência com sistemas similares.

É importante notar que os papéis não sejam fixos e que cada um possa contribuir com tudo que pode para a equipe.

2.4.5 Ciclo de vida do projeto XP

Um software desenvolvido a partir do XP terá que percorrer algumas fases durante o seu ciclo de vida [ver Figura 2.2]. De acordo com o tipo de projeto ou a característica da organização, por exemplo, essas fases podem sofrer modificações, porém manterão forte semelhança estrutural.

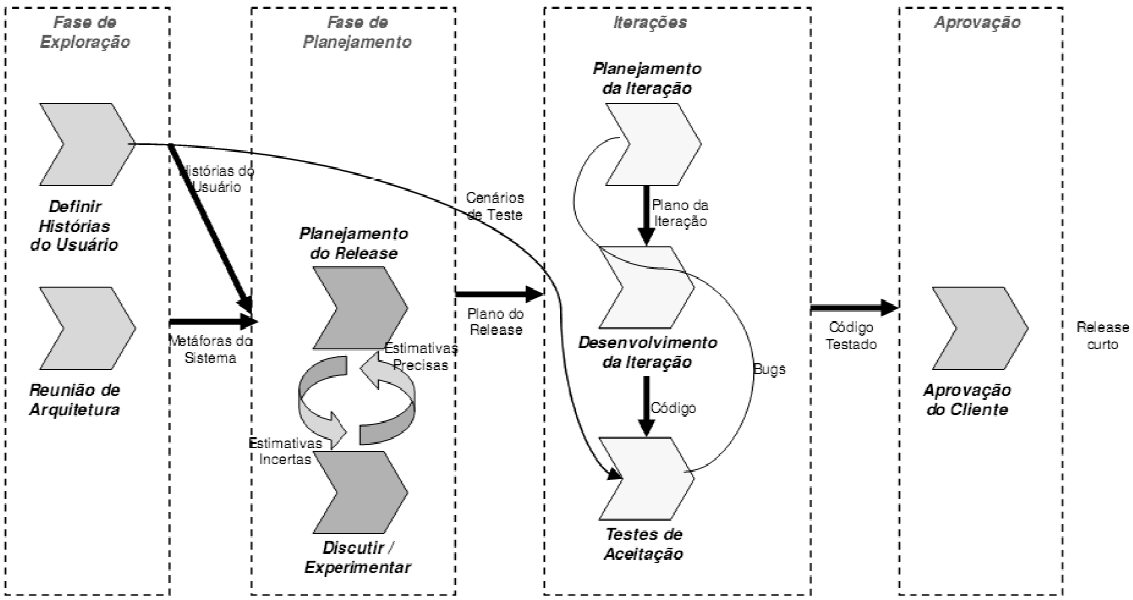


Figura 2.2 Ciclo de Vida do produto em XP [JÚNIOR 2008].

Em cada fase várias atividades são realizadas. Um projeto XP passa basicamente pelas seguintes fases: exploração, planejamento, iterações e aprovação, as quais são descritas a seguir:

- A fase de exploração é anterior à construção do sistema propriamente dita. Nela, investigações são feitas e é verificada a viabilidade de possíveis conclusões/soluções serem implementadas. Os programadores elaboram possíveis arquiteturas e tentam visualizar como o sistema funcionará, considerando os mais diversos aspectos, enquanto o cliente prepara as estórias. Quando existirem estórias suficientes, passa-se a formatar o primeiro *release* do sistema.
- A fase de planejamento inicia no momento em que será acordada uma data lançamento do primeiro *release*. Nessa etapa os programadores, de posse das estórias elaboradas pelo cliente, assinalam certa dificuldade para cada uma e, baseados na sua velocidade de implementação, dizem quantas podem implementar em uma iteração. Depois, os clientes escolhem as estórias de maior valor de negócio para serem implementadas na iteração. O processo então se repete até terminar as iterações do *release*. O tempo para cada iteração deve ser de uma a três semanas e para cada *release* de dois a quatro meses.
- Na fase das iterações do ***release*****,** de posse do planejamento da iteração, o plano de iteração é posto em desenvolvimento, momento em que os programadores seguem um fluxo de atividades (casos de testes funcionais/unidade, projeto e refatoramento, codificação, realização dos testes e integração e testes de aceitação). Os testes de aceitação já devem ter sido escritos a partir das estórias do cliente. À medida que este fluxo vai sendo seguido, o sistema vai sendo construído segundo os princípios, valores e práticas apresentados nas seções anteriores.
- Na fase de aprovação o cliente recebe algumas das estórias acordadas para o *release* já em funcionamento. Neste momento o cliente analisa o produto entregue e aprova ou desaprova. Qualquer que seja o posicionamento do cliente, essas informações serão muito úteis às demais *releases* e ao referido ciclo de vida XP.

Além destas fases, em algumas abordagens são citadas mais duas fases: manutenção e morte.

- A fase de manutenção pode ser vista como uma característica intrínseca a um projeto XP. O fluxo iterativo e incremental caracteriza o produto em constante manutenção, à medida que novas funcionalidades surgem, tecnologias e pessoas são incorporadas e o código é melhorado.
- A fase de morte corresponde ao término de um projeto XP. Um projeto chega ao seu fim de duas maneiras: a primeira é atendendo as necessidades do cliente e com um produto de qualidade, e a segunda por motivos negativos de inviabilidade econômica, dificuldade de adicionar funcionalidades a baixo custo e/ou alta presença de erros entre outras.

2.5 SCRUM

O processo ágil *Scrum* foi criado em 1996 por Ken Schwaber e Jeff Sutherland e destaca-se dos demais processos ágeis pela maior ênfase dada ao gerenciamento do projeto. Reúne atividades de monitoramento e *feedback*, em geral, reuniões rápidas e diárias com toda a equipe, visando a identificação e correção de quaisquer deficiências e/ou impedimentos no processo de desenvolvimento [SCHWABER 2008].

Scrum vem sendo largamente utilizado em organizações ao redor do mundo. Ele permite manter o foco na entrega do maior valor de negócio, no menor tempo possível, permitindo a rápida e contínua inspeção do software em produção. As necessidades do negócio é que determinam as prioridades do desenvolvimento de um sistema. As equipes se auto-organizam para definir a melhor maneira de entregar as funcionalidades de maior prioridade. Portanto, entre cada duas a quatro semanas todos podem ver o software real em produção, decidindo se o mesmo deve ser liberado ou continuar a ser aprimorado.

2.5.1 Características do *Scrum*

De acordo com [SCHAWABER & BEEDLE 2002], *Scrum* trata-se de uma abordagem empírica para lidar com o caos. É focado em pessoas e em ambientes onde há requisitos voláteis, resultando em uma abordagem que reintroduz as ideias de flexibilidade, adaptabilidade e produtividade. O foco do processo é encontrar a melhor forma de trabalho dos membros da equipe para produzir o software de forma flexível num ambiente em constante mudança.

O processo baseia-se em princípios como: equipes pequenas e multidisciplinares (no máximo 7 pessoas), requisitos instáveis ou desconhecidos e iterações curtas. Cada ciclo do *Scrum* é denominado *Sprint*, possuindo intervalos de tempo reduzido de 15 a 30 dias. Este processo não requer ou fornece qualquer técnica ou método específico para o desenvolvimento do software, ele enfatiza o planejamento e gerenciamento dos projetos, através de um conjunto de regras e práticas gerenciais que são estabelecidas.

No *Scrum* o cliente toma parte da equipe de desenvolvimento, através da figura do *Product Owner*, e existem reuniões frequentes com todos os envolvidos no projeto.

2.5.2 Papéis no *Scrum*

O *Scrum* define três papéis principais para as diferentes tarefas, propósitos e práticas do processo: *Scrum Master, Product Owner* (PO) e Time [SCHWABER 2008].

a) *Scrum Master*

O *Scrum Master* (SM) gerencia o processo, disseminando o *Scrum* a todos os envolvidos no projeto e adequando o mesmo à cultura da organização. Seu papel é remover os impedimentos do projeto e garantir que todos do time sigam as regras e práticas do *Scrum*.

Ele é o líder e facilitador para o time e para o *Product Owner*, sendo responsável por: resolver barreiras entre o time e o PO;; garantir que o processo seja seguido; motivar e incentivar a equipe de desenvolvimento, facilitando a criatividade e a capacitação; melhorar a produtividade da equipe; melhorar as práticas de engenharia e prover ferramentas de modo que cada nova funcionalidade seja potencialmente realizada; manter e divulgar as informações sobre o progresso da equipe.

b) Product Owner

O *Product Owner* (PO) representa o cliente no projeto. Seu foco é na parte comercial do produto. O PO define os objetivos do projeto criando requisitos iniciais e gerais (*Product Backlog*), planeja as entregas e prioriza o *Product Backlog* a cada *Sprint*, garantindo que as funcionalidades mais importantes sejam construídas prioritariamente [SZALVAY 2007].

O PO não deve gerenciar a equipe, deve procurar equilibrar os interesses dos *stakeholders* e evitar acrescentar mais funcionalidades após iniciar uma iteração. Segundo Schwaber (2009), o PO pode ser alguém do Time, trabalhando também em desenvolvimento. Mas essa missão adicional pode reduzir a sua habilidade de lidar com as partes interessadas. No entanto, o *Product Owner* nunca pode ser o *Scrum Master*.

c) Time

O time é um grupo de pessoas, envolvendo analistas, desenvolvedores, designers, gerente de qualidade, etc. com diferentes habilidades necessárias para implementar as funcionalidades. Quando necessário, o time tem a autoridade de decidir as ações que serão realizadas e priorizá-las organizando-as nas *Sprints*. O time deve gerenciar seu próprio trabalho, sendo responsável coletivamente pelo sucesso do projeto.

De acordo com Schwaber (2009), as pessoas que se recusam a programar porque são arquitetas ou designers não se adaptam bem a times. Todos contribuem, mesmo que isso exija aprender novas habilidades ou lembrar de antigas. Não há títulos em times, e não há exceções a essa regra. Os times também não contêm sub-times dedicados a áreas particulares como testes ou análise de negócios.

2.5.3 Artefatos do *Scrum*

A seguir serão descritos os principais artefatos utilizados no processo *Scrum*:

a) Product Backlog

O *product backlog* desempenha um papel bastante importante no *Scrum*. Ele contém a lista de todas as estórias de usuário, priorizadas pelo *Product Owner* [COHN 2004]. Estórias de usuário são as funcionalidades necessárias para desenvolver um produto de sucesso, ou seja, os requisitos funcionais e não-funcionais necessários para o cliente.

O *Backlog* do produto define tudo o que se tem conhecimento inicialmente que seja necessário para o produto final. Ele é dinâmico, evoluindo à medida que o produto e o ambiente em que ele será usado evoluem. Nele são definidas, por qualquer pessoa envolvida no projeto, as funcionalidades, as prioridades, a tecnologia e as estratégias de implementação.

Os itens do *Product Backlog* são documentados com as seguintes informações: descrição, estimativa em horas, um responsável e uma prioridade (baseada no risco, valor e necessidade). Para facilitar a visualização é sugerido que os itens sejam agrupados por prioridade.

b) Sprint Backlog

A *Sprint Backlog* é um conjunto de itens selecionados do *Product Backlog* em uma *Sprint*. É responsabilidade do time, do PO e do SM selecionar quais itens serão implementados na *Sprint*, de acordo com as prioridades dos itens.

c) Burndown* da *Sprint

Burndowns são gráficos utilizados para acompanhar o andamento do produto (*Release Burndown*) ou da *Sprint* (*Sprint Burndown*).

A *Sprint Burndown* indica a velocidade da equipe e o progresso da conclusão de tarefas na *Sprint* atual. Em um eixo do gráfico está a quantidade de tarefas do *Sprint Backlog* e no outro a quantidade de dias da *Sprint*.

Através do *Sprint Burndown* é possível analisar se a *Sprint* está atrasada (quando a linha real está acima da linha estimada) ou adiantada (quando a linha real está abaixo da estimada). A partir desta análise é necessário tomar algumas ações, como retirar ou adicionar novas tarefas. Como o *burndown* é atualizado diariamente, é possível ter um melhor acompanhamento da situação, evitando o atraso na entrega do software.

d) Impedimentos

Impedimentos são quaisquer fatores que impedem algum membro da equipe de realizar as suas atividades eficientemente. Eles devem ser citados durante a reunião diária e o SM é o responsável por desobstruir estes obstáculos. Quando um impedimento não pode ser resolvido durante a reunião diária, marca-se uma reunião específica com os envolvidos.

2.5.4 Práticas do Scrum

A seguir, serão apresentadas as práticas do *Scrum* de acordo com o Guia do *Scrum*, proposto por Ken Schwaber (2009):

a) Reunião de Planejamento da Versão para Entrega (*Realease Planning Meeting*)

A *Realease Planning Meeting* tem o objetivo de estabelecer um plano e metas que o time e o resto da organização possam entender e comunicar. O plano da versão para entrega estabelece a meta da versão, as maiores prioridades do *Product Backlog*, os principais riscos e as características gerais e funcionalidades que estarão contidas na versão. Ele estabelece também uma data de entrega e prováveis custos que devem manter-se em não havendo mudanças [SCHWABER & BEEDLE 2002].

b) *Sprint*

A *Sprint* é a principal prática do *Scrum*. Ela é uma iteração e segue o ciclo PDCA [4]– Plan (Planejar), Do (Fazer), Check (Verificar), Act (Agir) – que envolve as fases tradicionais de desenvolvimento: requisitos, análise, projeto e entrega. As *Sprints* ocorrem uma após a outra, sem intervalo entre elas, e cada uma deve durar no máximo

[4] O ciclo PDCA será detalhado no capítulo: Controle da Qualidade Total

30 dias. É durante uma *Sprint* que são executadas as atividades relacionadas aos itens definidos na *Sprint* Backlog. Segundo Beedle et. al. (1998), ao final de cada *Sprint* é criado um incremento funcional do produto, com o objetivo de mostrar ao cliente o que foi desenvolvido. A integração das novas funcionalidades com as outras partes já implementadas acontece na *Sprint*, além dos testes do software. Durante a *Sprint*, é recomendado que a equipe não seja interrompida com pedidos de novas implementações, para evitar mudanças no cronograma e consequentes atrasos.

c) Reunião de Planejamento da *Sprint* (*Sprint Planning Meeting*)

Cada *Sprint* começa com uma reunião chamada de Reunião de Planejamento da *Sprint*, geralmente dividida em 2 partes. É alocado para essa reunião aproximadamente 5% do tempo total da *Sprint*, dividido em intervalos iguais para as duas partes.

Na primeira parte é definido "o quê" será implementado. Os itens do *Product Backlog* são analisados a fim de priorizá-los para o desenvolvimento na próxima *Sprint*. Não deve haver influências do *Scrum Master* e do PO na decisão de quais Itens de Backlog (IBL) serão implementados, apenas o Time tem condições de avaliar a sua capacidade.

Já na segunda parte é debatido "como" serão implementados os IBLs. O time lista as atividades necessárias para tornar funcional os itens de backlog, as quais devem ser decompostas em tarefas a serem feitas em menos de um dia. A listagem com as tarefas da *Sprint* é chamada de *Sprint Backlog*. O *Sprint Backlog* poderá ser alterado durante o andamento da *Sprint*, a fim de adicionar novas tarefas ou aumentar o detalhamento das já existentes. A maturidade do time *Scrum* torna o *Sprint Backlog* cada vez mais estável.

d) Reuniões Diárias do *Scrum* (*Daily Scrum Meeting*)

O *Scrum* recomenda uma reunião diária com a participação de todos do time. A reunião não deve durar mais de 15 minutos, e deve acontecer sempre no mesmo local e horário. Cada membro deve falar brevemente na reunião, basicamente respondendo as seguintes perguntas:

- O que fez ontem?
- O que vai fazer hoje?
- Há algum impedimento no seu caminho?

A *Daily Scrum* melhora a comunicação, identifica e remove impedimentos, promove a tomada rápida de decisões e melhora o nível de conhecimento de todos sobre o projeto. Segundo o Guia *Scrum*, a Reunião Diária é utilizada para inspecionar o progresso em direção à meta da *Sprint* e para realizar adaptações que otimizem o valor do próximo dia de trabalho.

De acordo com Linda e Norman (2000), discussões para resolução de obstáculos são feitas mais tarde, onde somente os envolvidos no problema participam.

e) Revisão da *Sprint* (*Sprint Review*)

Na reunião de revisão da *Sprint*, o time apresenta as funcionalidades desenvolvidas ao cliente. Essa é uma reunião informal, que ocorre no último dia de cada *Sprint* e não deve durar mais de 5% do tempo total da *Sprint*. Os participantes avaliam as novas funcionalidades e decidem sobre as próximas atividades. A revisão da *Sprint* fornece dados valiosos para as reuniões de planejamento das próximas *Sprints*.

De acordo com o Guia do *Scrum*, a *Sprint Review* e a *Sprint Planning Meeting* funcionam como ponto de inspeção e adaptação do *Scrum*, com a finalidade de acompanhar o andamento do projeto e de fazer adaptações que otimizem o processo na próxima *Sprint*.

f) Retrospectiva da *Sprint* (*Sprint Retrospective*)

A Retrospectiva da *Sprint* acontece após a reunião de revisão, antes de iniciar as atividades do próximo ciclo. O objetivo dessa reunião é avaliar as pessoas, a comunicação, o processo e as ferramentas envolvidas. Devem ser listados, neste momento, os pontos fortes da *Sprint* atual e o que podia ter sido melhor, e ao final identificadas ações de melhoria a serem adotadas na próxima *Sprint*.

A *Sprint Retrospective* é uma oportunidade para adaptar o *Scrum* à realidade da empresa, tornando o processo mais eficaz e gratificante na próxima *Sprint*. É indispensável, portanto, que o *feedback* recebido na retrospectiva seja utilizado. Caso contrário, a equipe identifica a reunião como um desperdício de tempo.

2.5.5 Ciclo de Vida do *Scrum*

O ciclo de vida de um produto com *Scrum* é, segundo Schwaber (2006), dividido em três fases:

- **Planejamento (*Pre-game phase*):** Nesta fase é produzido o *Product Backlog*, definido o cronograma inicial e estimado o custo. É estabelecida a visão do projeto e expectativas garantindo recursos para a sua execução, como equipe de desenvolvimento e ferramentas. Esta fase inclui também a análise dos riscos e a definição da arquitetura do sistema.
- **Desenvolvimento (*game phase*):** Nesta fase o sistema é desenvolvido em Sprints. Em cada uma dessas iterações primeiramente faz-se a análise, em seguida o projeto, implementação e testes. Toda *Sprint* tem como resultado um incremento do produto final que é potencialmente entregável.
- **Releasing (*post-game phase)*:** Nesta fase é realizada a preparação para a entrega do software ao cliente. As seguintes atividades são realizadas nesta fase: integração do sistema, testes, documentação do usuário, *marketing*, preparação de treinamento e o lançamento do produto.

O *Scrum* tem um processo simples e bem definido como ilustrado na Figura 2.3.

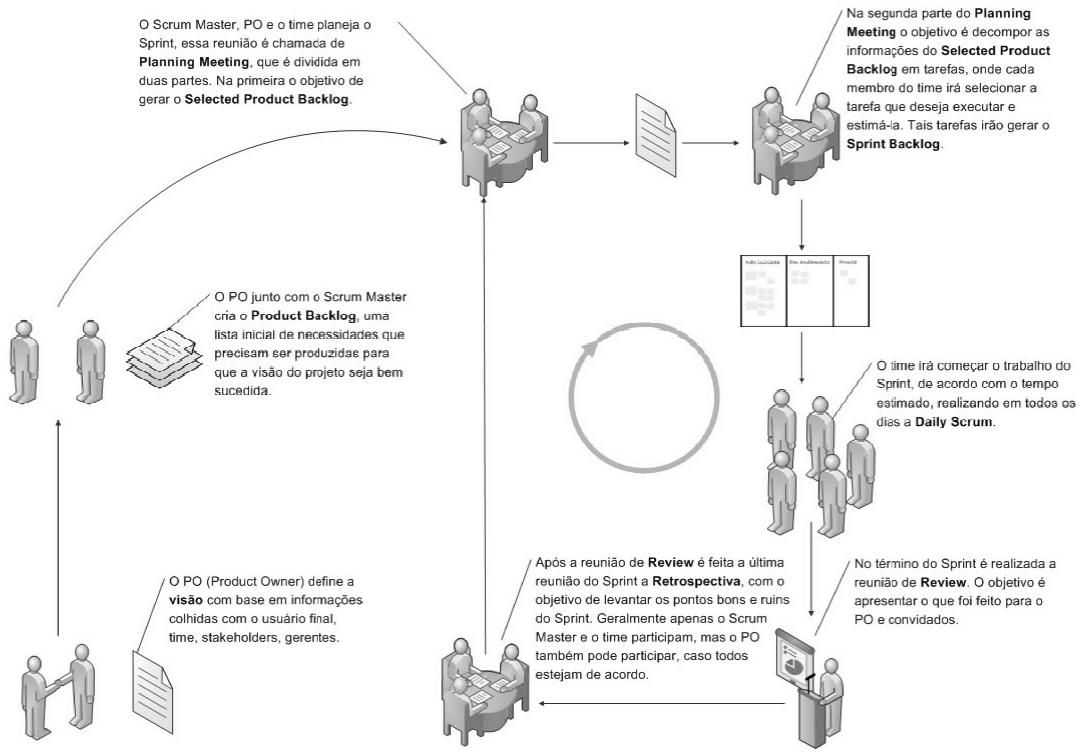


Figura 2.3 Ciclo de trabalho do *Scrum* [QUALIDADEBR 2009].

2.6 FEATURE DRIVEN DEVELOPMENT

O *Feature Driven Development* (FDD) é um processo ágil para gerenciamento e desenvolvimento de software, criado em 1997 em um grande projeto em Java para o *United Overseas Bank*, em Singapura. Nasceu a partir da experiência de análise e modelagem orientadas por objetos de Peter Coad [COAD et al. 1999] e de gerenciamento de projetos de Jeff De Luca [COAD et al. 1999], frente a uma necessidade da referida instituição [SLIGER 2008].

FDD é um processo voltado para o cliente e orientado à modelagem, combinando algumas das melhores práticas do gerenciamento ágil de projetos com uma abordagem completa para Engenharia de Software orientada por objetos [COAD et al. 1999]. Compõe-se de um arcabouço particular, através de seus princípios e práticas, que proporcionam um equilíbrio entre as filosofias tradicionais e as ágeis. O referido equilíbrio, segundo Michele Sliger (2008) proporciona uma transição mais suave para organizações mais conservadoras, e a retomada da responsabilidade para as organizações que se desiludiram com as propostas mais radicais.

2.6.1 Características do FDD

Os processos ágeis devem seguir o todo ou parte do que foi acordado no Manifesto Ágil, mesmo os que surgiram antes do referido manifesto, e por isso tendem a possuir características comuns [BECK & FOWLER 2001].

Algumas características intrínsecas nos processos ágeis são destacadas por Pekka Abrahamsson (2003), a saber: a) cliente presente; b) iterações curtas; c) equipes pequenas (menos de 12 aproximadamente); d) entregas frequentes do produto; e) adaptativos às mudanças; f) flexibilidade; e g) simplicidade.

Corroborando neste sentido, Craig Larman (2003) destaca que dentre as características comuns aos processos ágeis sempre existirá particularidades que os diferenciem. Seguindo essa linha das características ágeis próprias, Palmer (2002) destaca as seguintes em FDD:

- Resultados úteis a cada duas semanas ou menos;
- Blocos bem pequenos de funcionalidade valorizados pelo cliente, chamados "*features*";
- Planejamento detalhado e guia para medição;
- Rastreabilidade e relatórios com alta precisão;
- Monitoramento detalhado dentro do projeto, com resumos de alto nível para clientes e gerentes, tudo em termos de negócio; e
- Fornece uma forma de saber, dentro dos primeiros 10% de um projeto, se o plano e a estimativa são sólidos.

Essas características são detalhadas a seguir..

2.6.2 Papéis no FDD

O FDD apresenta em seu escopo a definição de papéis para que se possa ter uma maior organização e visão na hora de se pensar/iniciar um projeto. Neste contexto, o FDD estrutura seu time em[PALMER 2002]:

- **Gestor do Projeto:** trata das questões financeiras e administrativas do projeto. É o membro que dá a palavra final sobre o escopo, objetivos, o time e prazos. É também atribuição sua prezar por ótimas condições de trabalho e manter o time focado, com vistas a maximizar os resultados;
- **Chefe de *Design*:** responsável por toda a arquitetura do projeto, bem como pelas sessões de design, nas quais apresenta seu entendimento ao time;
- **Gestor de Desenvolvimento:** acompanha as atividades de desenvolvimento do código diariamente, bem como tem a incumbência de fazer com que problemas não cheguem ao time ou que os mesmos sejam resolvidos o mais rapidamente. Desempenha suas funções de forma afinada com o gestor de projeto;
- **Programador Chefe:** é responsável por uma equipe pequena no que se refere à divisão e atribuição de trabalho entre seus membros. Recomenda-se que seja um programador experiente, pois fará parte de suas atribuições a escolha das funcionalidades a serem implementadas em cada iteração, bem como o relatório de atividades do time. Deve permitir um canal aberto de comunicação com o chefe de design e com o programador chefe;
- **Dono de Classe:** responsável pela arquitetura, implementação, teste e documentação de uma determinada classe. Ele fará parte das equipes cujas funcionalidades sejam envolvidas com a sua classe;
- **Especialista da Área:** membro conhecedor do assunto sobre o qual a aplicação atuará. Trabalha em conjunto com o gestor de projeto em algumas questões

macro que sua área lhe habilita, bem como ao lado dos desenvolvedores com suporte de conhecimento necessário à construção da *feature*.

Por se tratar de um processo ágil, no qual a flexibilidade e adaptabilidade são presentes em sua essência, um membro pode assumir mais de um papel simultaneamente, e um mesmo papel pode ser assumido por vários membros. Isso acontece dependendo das características de cada projeto.

O FDD, inicialmente, recomenda uma composição de equipe de até 20 membros, mas há inclinações em empresas ou profissionais do desenvolvimento de software à fragmentação e hierarquização dos processos, para sempre aplicados em times bem maiores [HEPTAGON 2009]. Alguns processos ágeis, inclusive o FDD, afirmam se aplicar a qualquer tipo de projeto de desenvolvimento ágil, não importando suas características [ABRAHAMSONN 2003].

2.6.3 Práticas do FDD

O FDD possui em seu arcabouço um conjunto de boas práticas baseadas na engenharia de software. As práticas do FDD focam em atender as necessidades do cliente e na produção do sistema com qualidade. Abaixo serão descritas algumas dessas práticas [PALMER 2002]:

- **Modelagem de objeto do domínio**: é construída, inicialmente, uma modelagem genérica com as funcionalidades do sistema, dentro da perspectiva da orientação a objetos. Essa modelagem possibilita um maior entendimento/visibilidade do problema a ser resolvido;

- **Desenvolvimento por funcionalidade:** cada funcionalidade é analisada individualmente com a perspectiva de poder ou não decompô-la em atividades menores. O foco é na funcionalidade que está sendo trabalhada no momento. Essa prática possibilita mais segurança, maior flexibilidade e escalabilidade ao código, devido ao foco dado e à particularidade atendida na funcionalidade;

- **Posse individual do código:** uma funcionalidade ou um conjunto delas é delegado a determinado desenvolvedor e este se torna automaticamente responsável por tudo que estiver relacionado ao código, desde a performance, passando pela consistência e corretude até a integração da classe;

- **Equipes de funcionalidades:** as equipes são montadas para atender determinada funcionalidade ou um pequeno conjunto delas, conforme o tamanho, dependência e semelhança. A partir do ponto de vista/entendimento de cada membro é convergido para determinar o design da funcionalidade e sua solução final;

- **Inspeções:** inspeção de código é uma prática da engenharia de software que possibilita, através da análise e acompanhamento do código, um melhoramento no que se refere à redução de erros, promover uma melhor modelagem e manter a legibilidade e alta coesão;

- **Gerência de configuração:** atividade que busca manter o controle sobre o código fonte, permitindo a vinculação referencial (funcionalidade-código-proprietário), além de manter histórico de alterações no código;

- ***Build* constante:** deve existir sempre uma versão do sistema rodando numa máquina, garantindo à equipe o funcionamento de pelo menos uma versão do sistema. Essa prática garante que existirá uma versão do sistema que pode ser utilizada, à parte, a qualquer momento sem interferir no desenvolvimento;

- **Visibilidade do progresso:** a prática recomenda que um relatório de progresso das atividades do projeto seja mantido visível à equipe e aos demais interessados, para saberem exatamente como estão em termos de produtividade. Essa atividade requer muita atenção, precisão e constância, pois dados incorretos podem levar a decisões desastrosas ao projeto.

2.6.4 Ciclo de Vida do FDD

O FDD possui uma estrutura muito objetiva. O seu ciclo de vida apresenta duas fases (Concepção/Planejamento e Construção) e possui cinco processos (Desenvolver um modelo abrangente, Gerar uma lista de funcionalidades, Planejar por funcionalidade, Detalhar por funcionalidade e Construir por funcionalidade). Os três primeiros processos fazem parte da primeira fase (Concepção/Planejamento) e são executados de forma sequencial. Já os dois últimos, fazem parte da segunda fase (Construção) e são executados de forma interativa e incremental. A Figura 2.4 ilustra o ciclo de vida de desenvolvimento utilizando FDD, o qual será detalhado a seguir:

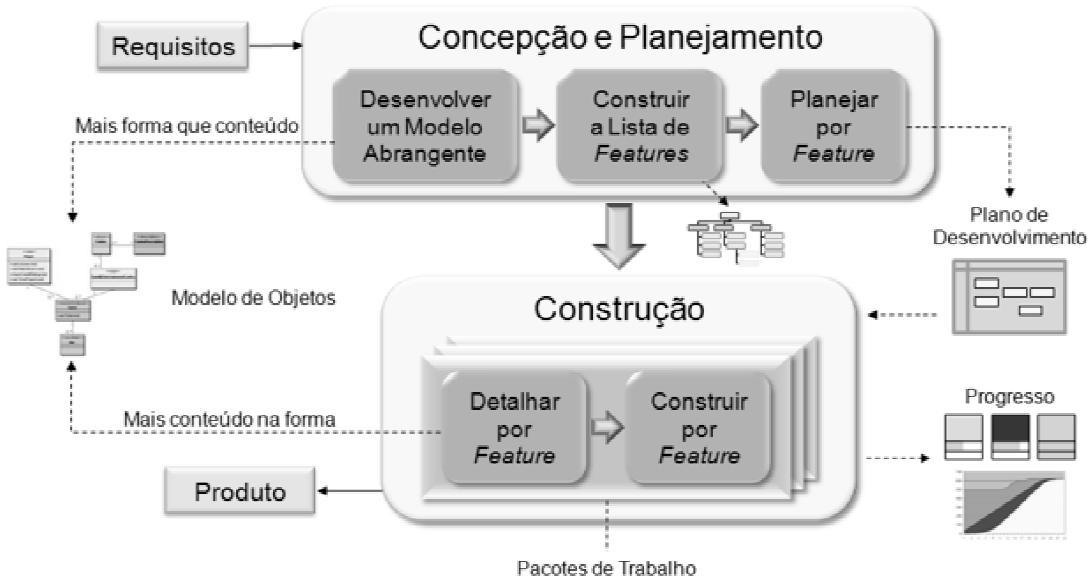


Figura 2.4 Ciclo de Vida do produto FDD [HEPTAGON 2009]

Desenvolver um modelo abrangente: começa com uma análise superficial do escopo do sistema e seu contexto, seguido de um estudo do(s) domínio(s) de negócio(s) do sistema que leva à criação do referido modelo. Depois é criada uma modelagem superficial para cada área de domínio existente. Os modelos decorrentes das referidas atividades serão revisados por um grupo de membros do projeto e melhorias são propostas e discutidas. Finalmente, os modelos são fundidos para gerar um modelo geral do domínio do sistema. Neste processo são executadas as seguintes atividades:

- Formar equipe de modelagem;

- Estudar o domínio de negócio;

- Estudar os documentos;
- Formar várias equipes pequenas para sugerir uma solução de modelo;
- Desenvolver o modelo escolhido;
- Refinar o modelo geral; e
- Escrever notas explicativas sobre o modelo final.

Gerar uma lista de funcionalidades: recebe as práticas e o conhecimento adquiridos no processo anterior e tem o objetivo de elaborar uma lista de funcionalidades do sistema decompondo as áreas de domínio obtidas. Cada funcionalidade identificada deve seguir o formato <ação> <resultado> <objeto> e é uma pequena tarefa a ser implementada que agregue valor ao cliente. Como no processo anterior, este também possui sub-atividades:

- Escolher uma equipe para gerar uma lista de funcionalidades; e
- Gerar a lista de funcionalidades.

Planejar por funcionalidade: prega que seja feito o planejamento de desenvolvimento de cada funcionalidade da lista obtida do processo anterior. Os programadores-chefe recebem classes ou trechos de código e são responsáveis pelo planejamento e programação da funcionalidade. As sub-atividades deste processo são:

- Formar uma equipe de planejamento;
- Determinar a sequência de desenvolvimento das funcionalidades;
- Designar atividades de negócio para os programadores-chefe; e
- Designar classes para os desenvolvedores.

Detalhar por funcionalidade: requer uma interação entre os programadores-chefe e os proprietários de código, quando da escolha de algumas funcionalidades para que sejam feitos diagramas de sequência e a modelagem completa das funcionalidades. Vale salientar que não se trata da mesma modelagem feita no primeiro processo, neste momento a modelagem é feita para a funcionalidade em questão. Neste processo o nível de detalhamento é bem maior, pois se deve pensar em classes, métodos e atributos que irão existir. Ao final, é realizada uma inspeção do modelo pela equipe que a fez ou por outra designada. As sub-atividades associadas a este processo são:

- Formar uma equipe para a funcionalidade em questão;
- Estudar a funcionalidade como parte do modelo de domínio;
- Estudar documentos relacionados à funcionalidade;
- Desenvolver diagrama de sequência;
- Refinar o modelo de objeto;
- Escrever as classes e as assinaturas dos métodos (tipo de retorno, parâmetros e exceções lançadas); e

- Realizar inspeção da modelagem.

Construir por funcionalidade: neste processo o programador-chefe designa um programador para desenvolver o código, os testes unitários são realizados e a funcionalidade ganha vida. As sub-atividades associadas a este processo são:

- Implementar as regras de negócio das classes;
- Inspecionar código;
- Conduzir testes unitários; e
- Release da funcionalidade.

2.7 Considerações Finais

O processo de desenvolvimento de software sofreu várias modificações ao longo dos anos. A vivência, as pesquisas e estudos e, ainda, uma demanda natural do mercado levaram a uma nova forma de desenvolver software, na qual se pudessem atender as principais demandas/necessidades dos clientes com agilidade, redução de custos e a qualidade merecida.

Neste contexto, este capítulo apresentou uma visão geral sobre o paradigma ágil, abrangendo desde a motivação para o surgimento do manifesto ágil até a descrição detalhada dos principais processos ágeis de desenvolvimento de software de forma a permitir que o leitor possa analisar, escolher, adaptar e/ou fundir processos e práticas que possam atender a uma demanda específica. Ainda com o intuito de expor ao leitor características dos processos ágeis apresentados neste capítulo, é apresentada nas Tabelas 2.2a e 2.2b uma comparação proposta por Fagundes *et. al*. [2008].

Tabela 2.2a Comparação dos processos ágeis. Adaptado de [FAGUNDES et. al. 2008]

	XP	***Scrum***	**FDD**
Definição dos Requisitos	Clientes escrevem as *user stories*.	Definição do *Product Backlog.*	Geração de artefatos para a documentação dos requisitos.
Atribuição dos Requisitos às Iterações	Equipe técnica e clientes definem as *user stories* que serão desenvolvidas nas iterações. As iterações duram de 1 a 4 semanas.	Definição do *Sprint Backlog*. As *Sprints* (iterações) duram no máximo 30 dias.	As características são agrupadas, priorizadas e distribuídas aos responsáveis pelo seu desenvolvimento. As iterações duram no máximo 2 semanas.
Projeto da Arquitetura do Sistema	Propõe que em paralelo à escrita das *user stories*, seja realizado o projeto da arquitetura do sistema, porém não sugere como o projeto é feito.	Sugere que seja feito um projeto geral do sistema baseado nos itens do *Product Backlog*, mas não cita nenhuma técnica associada a esta atividade.	Sugere que seja construído um diagrama de classes da UML para representar a arquitetura do sistema. Para complementar, também poderão ser gerados diagramas de sequência da UML.

Tabela 2.2b Comparação dos processos ágeis. Adaptado de [FAGUNDES et. al. 2008]

	XP	Scrum	FDD
Desenvolver Incremento do Sistema	Implementação das *user stories* que fazem parte da iteração corrente por duplas de programadores.	Implementação dos requisitos contemplados no *Sprint Backlog* para a *Sprint* corrente.	Análise da documentação existente, refinamento do modelo gerado nas atividades anteriores e implementação das características que serão desenvolvidas durante a iteração corrente.
Validar Incremento	Os programadores executam os testes de unidade e os clientes executam os testes de aceitação.	O *Scrum* não adota nenhum processo de validação pré-definido.	Os testes e inspeções são executados pelos próprios programadores após a implementação.
Integrar Incremento	A integração acontece paralelamente ao desenvolvimento das *user stories*.	Atividade realizada ao final de cada *Sprint*.	Atividade realizada após os testes no incremento.
Validar Sistema	O sistema é disponibilizado ao cliente para que o mesmo realize validações.	O cliente valida o sistema integrado em uma reunião no último dia da *Sprint*.	Esta atividade ocorre através das inspeções e dos testes de integração.
Entrega Final	Cliente satisfeito com o sistema.	Todos os itens do *Product Backlog* desenvolvidos.	O sistema é entregue após todos os conjuntos de características implementados.

2.8 Tópicos de Pesquisa

Os processos ágeis de desenvolvimento de software têm cumprido com o seu propósito e atendido com sucesso às demandas do mundo do software. Com o intuito de tornar estes processos cada vez mais adequados a estas demandas, alguns trabalhos de pesquisa podem ser desenvolvidos:

- **Adaptação de Processos Ágeis de Desenvolvimento de Software:** A literatura e a prática diária mostram que algumas organizações, projetos e/ou times de desenvolvimento não conseguem aplicar fielmente todas as recomendações de determinado Processos Ágeis. Com isso passam a adaptar suas práticas de modo a tornar viável o uso de tais processos. Neste contexto, guias de adaptação para este tipo de processos podem ser propostos.

- **Desenvolvimento Distribuído de Software (DDS) com Processos Ágeis:** O DDS [ver Capítulo 3] é uma necessidade inquestionável nos dias atuais quando se buscam fatores como qualidade, competitividade, redução de custos, mão-de-obra qualificada entre outros. Porém existem alguns desafios que precisam ser vencidos de modo a tornar viável em grande escala este tipo de desenvolvimento. Dentre estes desafios, um possível tópico de pesquisa é: *Como utilizar DDS em um contexto de desenvolvimento ágil?*

- **Aproximação entre os Processos Ágeis de Desenvolvimento de Software e os Modelos de Qualidade de Software:** A literatura nos oferece experiências da aproximação entre processos ágeis e modelos de qualidade de software, porém embora existam diversas adequações do uso dos primeiros nos segundos, isto não acontece de forma completa (por exemplo: nem toda a área de processo CMMI é satisfeita somente com a utilização de abordagens ágeis). Neste contexto, guias de adaptação de processos ágeis específicos a determinados modelos de qualidade podem ser propostos.

2.9 Sugestões de Leitura

O conteúdo deste capítulo foi elaborado a fim de introduzir os conceitos do paradigma ágil, bem como apresentar alguns Processos Ágeis. No entanto, a diversidade de processos ágeis existentes e as particularidades apresentadas por cada levam à necessidade de aprofundar o conhecimento nesta área.

Para ampliar o entendimento sobre Extreme Programming é recomendado a leitura dos livros:

- *Programação Extrema (XP) Explicada: Acolha as Mudanças*. Bookman, 2004, Kent Beck (versão original em inglês) e;

- *Extreme Programming. Aprenda como encantar seus usuários desenvolvendo software com agilidade e alta qualidade.* Novatec, 2004, Vinícius Malhães Teles.

O conhecimento mais detalhado sobre o *Scrum* poderá ser encontrado com a leitura do Guia do *Scrum*, disponível em vários idiomas, inclusive em português, no site http://www.scrumalliance.org. Neste endereço é possível encontrar outros artigos e materiais relacionados ao *Scrum*.

O material recomendado para agregar conhecimento em *Feature Driven Development* (FDD), a partir de um foco prático e adaptativo da mesma, está disponível no site da empresa Heptagon (http:/www.heptagon.com.br), o qual além de seu conteúdo, dispõe de inúmeros links para o referido assunto. Outro repositório de conhecimento sobre do assunto encontra-se no livro *A practical Guide to Feature Driven Development*. 2002, Stephen Palmer.

O Manifesto Ágil, base inicial dos processos ágeis, pode ser encontrado no endereço http://agilemanifesto.org. Neste endereço são encontrados os princípios e valores propostos no Manifesto Ágil, além de *links* para alguns processos ágeis.

2.10 Exercícios

1. O que motivou o surgimento de processos ágeis de desenvolvimento?
2. Quais os valores propostos pelo Movimento Ágil? Cite também alguns princípios.
3. O *Agile Modeling* é um processo ágil? Justifique.
4. Compare o XP com o *Scrum*.

5. Quais os valores propostos por Beck no *Extreme Programming*?
6. Cite e explique as práticas do XP que você considera mais importantes.
7. Caracterize o processo FDD.
8. O que é *Sprint*? Quais os papéis que estão envolvidos na *Sprint*?
9. Comente o ciclo de vida do *Scrum*.
10. Compare os processos tradicionais de desenvolvimento com os ágeis.

2.11 Referências

ABRAHAMSSON, P., WARSTA, J., SIPONEN, M.T., RONKAINEN, J. (2003) *New Directions on Agile Methods: A Comparative Analysis*. In: ICSE 2003, USA.

AGILE MANIFESTO (2001). Disponível em: http://www.agilemanifesto.org. Acesso em: 21 de setembro de 2009.

AMBLER, S. (2002) *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. New York: Wiley Computer Publishing.

AMBLER, S. (2009) *Agile Modeling*. Disponível em: http://www.agilemodeling.com. Acesso em: 05 de novembro de 2009.

ANDERSON, D. J. (2003) *Agile Management for Software Engineering: Using the Theory of Constraints for Business Results.* Trentice Hall, 2003.

BECK, Kent; FOWLER, Martin. (2001) *Planning Extreme Programming*. 1ª edição. *Boston*: Addison-Wesley.

BECK Kent. (1999) *Extreme Programming Explained: Embrace Change*. 1ª edição. Boston: Addison-Wesley.

BECK, Kent; ANDRES, Cynthia. (2004) *Extreme Programming Explained: Embrace Change*. 2ª edição. Boston: Addison-Wesley.

BEEDLE, M; DEVOS, M.; SHARON, Y; SCHWABER, K; SUTHERLAND, J. SCRUM: *An extension pattern language for hyperprodutive software development*. In: *Pattern Languages of Programs'98 Conference, Monticello,* 1998.

CHARETTE, R. N. (2002). *Foundations of Lean Development: The Lean Development manager's guide*. Volume 2. Spotsylvania: ITABHI Corporation;

COAD, Peter; LEFEBVRE, Eric; LUCA, Jeff. (1999) *Java Modeling In Color With UML: Enterprise Components and Process. Upper Saddle River*, N.J.: Prentice Hall.

COCKBURN, A. (2002). *Agile Software Development*. Boston: Addison-Wesley.

COHN, M. (2004). *User stories applied for Agile Software Development*. Boston: Addison-Wesley.

FAGUNDES, P. B.; DETERS, J. I.; SANTOS, S. S. (2008). Comparação entre os processos dos métodos ágeis: XP, *Scrum*, FDD e ASD em relação ao desenvolvimento iterativo incremental. Disponível em: http://revista.ctai.senai.br/index.php/edicao01/article/viewDownloadInterstitial/21/18 Acesso em: 28 de novembro de 2009.

FOWLER, M. (2009) *The New Methodology*. Disponível em: http://martinfowler.com/articles/newMethodology.html. Acesso em: 20 de setembro de 2009.

HEPTAGON. Disponível em: www.heptagon.com.br. Acesso em: 05 de outubro de 2009.

HIGHSMITH, J. (2002) *Agile software development ecosystems*. Boston, MA., Pearson Education.

HIGHSMITH, J. (2004) *Agile Project Management - Creating Innovative Products.* AddisonWesley.

JÚNIOR, C. A. S.. Avaliação da Utilização de Metodologias Ágeis no Contexto dos Modelos de Qualidade de Software. Dissertação de mestrado. 2008

JURAN, Joseph M.. Qualidade desde o Projeto. Ed. Pioneira, Rio de Janeiro 1992.

LARMAN, Craig. (2003) *Agile and iterative development: a manager's guide*. Addison-Wesley.

LINDA, Rising; NORMAN, Janoff. (2009) *The Scrum Software Development Process for Small Teams*. IEEE Software, vol 17, issue 4, p. 26-32, Julho de 2009.

KOSCIANSKI, A., SOARES, M. (2006) Qualidade de Software. 2ª edição. São Paulo: Novatec.

PALMER S. R., FELSING J. M. (2002) *A Practical Guide to Feature-Driven Development (The Coad Series).* Prentice Hall PTR, USA.

QUALIDADEBR (2009) *Scrum*. Disponível em: http://qualidadebr.wordpress.com/2009/07/12/*scrum*/ Acesso em: 05 de novembro de 2009.

SLIGER, Michele; BRODERICK, Stacia. (2008) *The Software Project Manager's Bridge to Agility*. Addison Wesley Professional, 2008.

SCHWABER, K. (2006) *Scrum Development Process: Advanced Development Methods*. Disponível em: http://jeffsutherland.com/oopsla/schwapub.pdf. Acesso em: 04 de novembro de 2009.

SCHWABER, K. (2008) *Agile Project Management with Scrum*. Redmond: Microsoft Press.

SCHWABER, K., BEEDLE, M. (2001) *Agile Software Development with Scrum*. City: Prentice Hall.

SCHWABER, K. (2009) Guia do *Scrum*. Disponível em: http://www.scrumalliance.org/resources. Acesso em: 05 de novembro de 2009.

SZALVAY, V. (2007) *Glossary of Scrum Terms*. Disponível em: http://www.scrumalliance.org/articles/39-glossary-of-*scrum*-terms#1117. Acesso em: 28 de outubro de 2009.

STAPLETON, J. (2003). DSDM, *Business Focused Development*. Addison-Wesley.

SATO, D. (2007) Uso eficaz de métricas em desenvolvimento de software. 155 p. Dissertação (Mestrado em Ciência da Computação). Instituto de Matemática e Estatística – Universidade de São Paulo, São Paulo, 2007.

TELES, Vinícius Manhães. Extreme Programming: Aprenda como encantar seus usuários desenvolvendo software com agilidade e alta qualidade. Novatec. 2004

VERSIONONE (2008). 3º *Annual Survey: The State of Agile Development*. Disponível em: http://www.versionone.com/pdf/3rdAnnualStateOfAgile_FullDataReport.pdf. Acesso em: 05 de novembro de 2009.