Capítulo

2

Processos Ágeis de Desenvolvimento de Software

Márcio Amorim de Medeiros,   
Milton Moura Campos Neto

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 contextualizaremos a respeito do paradigma ágil e utilizaremos, também, terminologias como métodos e metodologias nessa abordagem, com ênfase no *Extreme Programming* (XP), *Scrum e Feature Driven Development* (FDD).

## 2.1 Introdução a Processos Ágeis de Desenvolvimento de Software

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 processo algum. 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, mais flexível à mudança 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 decorrerdeste capítulo.

Atualmente, mudança é algo bastante comum na vida de um software, a fim de garantir adaptação do sistema às novas necessidades do cliente, das instituições ou do mercado. Os processos tradicionais tendem a tentar planejar grande parte do software por um longo período antes de iniciar a implementação. Com isso, o software demora a ser disponibilizado ao cliente. Durante esse tempo podem surgir novos padrões, políticas e tecnologias que afetam os requisitos do software, o cliente pode perceber que alguma funcionalidade não está conforme solicitado ou precisar de outras. Esses fatores implicam em mudança no sistema, que não é bem-vinda nos métodos tradicionais, pois a fase de planejamento já foi concluída.

Outro fator comum 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 nunca utilizada, enquanto outras funções mais prioritárias ainda não foram implementadas.

As metodologias ágeis surgiram com a finalidade de desburocratizar o processo de desenvolvimento. Elas tentam se adaptar e fortalecer com as mudanças, até mesmo a ponto de se auto-modificarem. Os clientes têm, em curto espaço de tempo, versões de software executável, onde são priorizadas as funcionalidades que agregam mais valor ao seu negócio. Com isso, eles já podem sugerir novas funcionalidades e correções.

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, descrições simples do funcionamento do sistema, usadas para ajudarem os envolvidos no projeto a ter uma visão de seu funcionamento e entender os elementos básicos do projeto e seus relacionamentos.

Os Processos ágeis são orientados a pessoas ao invés dos tradicionais que são orientados a processos. Metodologias ágeis afirmam que nenhum processo jamais será semelhante à habilidade da equipe de desenvolvimento. Em vista disso, o papel do processo é dar suporte à equipe e seu 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. Foram motivados pela conclusão de que os processos de desenvolvimento estavam tornando-se cada vez mais longos, atolando as equipes de construção de softwares.

A essência desse movimento é a definição de novo enfoque de desenvolvimento de software, calcado na agilidade, 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]

Esse movimento, marco inicial do desenvolvimento ágil de software, foi descrito conforme a Figura 2.1.

|  |
| --- |
| **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. |

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

Os envolvidos se denominaram de Aliança Ágil. Esta abordagem tentava manter a qualidade dos projetos de software permitindo aos mesmos que mudanças fossem inseridas em seus desenvolvimentos, mas que reduzisse seus impactos, esta flexibilidade foi traduzida nos quatro valores vistos acima e em doze princípios que estão mostrados a seguir:

* A maior prioridade é satisfazer o cliente através de entregas antecipadas e contínuas de software de valor ao cliente;
* Mudanças de requisitos são bem vindas, mesmo que tardiamente no desenvolvimento. Processos ágeis se aproveitam da mudança para vantagem competitiva do cliente;
* Entrega freqüente de software funcionando, de duas semanas de trabalho até dois meses, com preferência à escala de tempo mais curta;
* Pessoas de software e 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 sustentável indefinidamente;
* Atenção contínua a excelência técnica e um bom projeto melhoram a agilidade;
* Simplicidade: a arte de maximizar a quantidade de trabalho não feito, 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

São considerados Processos Ágeis de Desenvolvimento de Software as metodologias que seguem os princípios do Manifesto Ágil – como entrega freqüente, flexão a mudança, foco em pessoas e simplicidade.

Os métodos ágeis ganham mais adeptos à medida que estão evoluindo com as 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 as metodologias ágeis mais utilizadas nas empresas, conforme Figura 2.2. As principais metodologias ágeis serão caracterizadas a seguir:

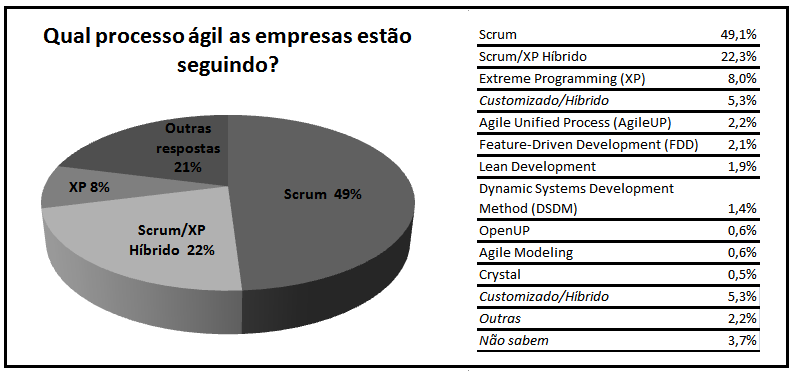


Figura 2.2 Metodologias Ágeis mais utilizadas nas empresas de acordo com a 3° Pesquisa Anual   
sobre o Estado do Desenvolvimento Ágil – Ano 2008. Adaptado de [VERSIONONE 2008].

A metodologia *Agile Unified Process* (AUP), também conhecida como AgileUP e desenvolvida por Ambler [AMBLER 2002], é uma versão simplificada do *Rational Unified Process* (RUP), que está detalhada no Capítulo 1. Ela descreve como desenvolver sistemas utilizando técnicas ágeis, como *Test Driven Development* (TDD), *Agile Modeling* e *Refactoring* no banco de dados para melhorar a produtividade, porém mantendo-se fiel ao RUP.

A *Lean Development* tem suas raízes na indústria automotiva, ela é uma adaptação para software do *Lean Manufacturing*do revolucionário Sistema Toyota de Produção. Inicialmente proposto por Bob Charette, 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.

Baseado no Desenvolvimento Rápido de Aplicações (RAD) e no modelo iterativo e incremental, a metodologia *Dynamic Systems Development Method* (DSDM) se tornou o *framework* para RAD [STAPLETON 2003]. A idéia central do 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 uma única metodologia, e sim, uma família de métodos denominada, portanto, de Família *Crystal*, proposto por Alistair Cockburn [COCKBURN 2002]. É categorizada por cores de acordo com a quantidade de pessoas envolvidas. 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), e assim por diante.

Já a *Agile Modeling* (AM), apesar de ser citada na pesquisa, não é uma metodologia de desenvolvimento de software. É uma metodologia de modelagem ágil, isto é, AM visa construir e manter modelos de sistemas de maneira eficaz e eficiente, que pode ser utilizada dentro de metodologias ágeis ou nos processos tradicionais, como o RUP.

Os métodos 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 gerenciamento dos projetos. Ambos serão tratados com maior detalhe nas próximas seções, além da metodologia FDD, pois 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 Cunninhgam, quando do lançamento do livro *Extreme Programming Explained* que formalizou e difundiu o referido processo. As experiências dos dois, que culminou no XP, vêm desde o início da mesma década com o desenvolvimento na linguagem *Smalltalk*.

A metodologia XP é definida como leve e apoiada em valores, princípios e práticas, cujo foco é o desenvolvimento de um produto que venha atender aos objetivos do cliente e forte adaptabilidade a requisitos não totalmente definidos e em constante mudança [BECK 2004]. 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 diferencia XP de outras 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: a) uma filosofia de desenvolvimento de *software* baseada nos valores (comunicação, *feedback*, simplicidade, coragem e respeito); b) um conjunto de práticas, que expressam os valores, comprovadamente úteis para melhorar o desenvolvimento de software; um conjunto complementar de princípios, técnicas intelectuais que auxiliam a tradução dos valores em práticas, úteis quando as práticas existentes não resolvem se problema particular; e uma comunidade que compartilha dos mesmos princípios e muitas das mesmas práticas.

Nas seções a seguir serão detalhados esses valores, princípios e práticas do XP, com base na nova abordagem feita por Kent Beck, quando da publicação da segunda edição do seu livro em 2004, citado anteriormente.

### 2.4.1 Valores do XP

A metodologia XP apresenta os seguintes valores que descrevem os objetivos e definem critérios para se obter sucesso:

* **Comunicação:** o fator de sucesso e insucesso de projetos de *software* é atribuído em grande parte à qualidade na comunicação. A grande maioria dos processos existentes valoriza a comunicação, porém frente à dificuldade de comunicar-se aposta na documentação extensa e complexa. 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:** a partir do Princípio de Pareto – 80% das consequências são frutos de 20% das causas –, os desenvolvedores adotam que se produza o mais simples possível que seja funcional. XP recomenda que manter o sistema simples é essencial para não gerar mais trabalho. Os desenvolvedores devem pensar no presente, não generalizar sem necessidade nem supor necessidades, bem como alertas a oportunidades de refatorar 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 idéia, ou descubra requisitos dos quais não estava ciente.
* **Coragem:** quando se quer iniciar o desenvolvimento de um *software* muitos são os medos envolvidos. O XP combate esses 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 com mais coragem, quando avalia o *software* funcional após cada *release*, sabendo que pode priorizar as funcionalidades que lhe são mais importantes. No jogo do planejamento as estimativas das funcionalidades poderão ser feitas com mais coragem e confiança quando do tempo de entrega.
* **Respeito:** valorizar a relação entre membros da equipe e, também, a de cada membro com o projeto e sua instituição. Esse valor também é corroborado com o princípio da diversidade e visto através da prática do código compartilhado.

### 2.4.2 Princípios do XP

Os princípios em XP compõem uma das bases (valores, princípios e práticas) de sua sustentação. Após alguns anos de *feedback* Kent Beck (2004) evidencia essa pilastra como um instrumento importante, também, na adaptação de práticas ao contexto local e até mesmo na criação de novas práticas. Os princípios agem como guias, específicos ao domínio da programação, para localizar práticas concretas em harmonia com os valores abstratos. Desse modo, Kent Beck (2004) elenca os seguintes princípios do XP:

* **Humanidade:** reconhece que são pessoas com necessidades humanas que desenvolvem *software*. Destas necessidades, algumas podem ser satisfeitas no trabalho. São elas segurança, crescimento, identidade com o grupo, realização, intimidade e privacidade. Este princípio se concretiza na prática de trabalho energizado. Com tempo para se dedicar a satisfazer suas outras necessidades fora do trabalho, os seres humanos podem voltar com energia para se dedicar ao trabalho. Um dos desafios propostos por este princípio é balancear as necessidades individuais com as da equipe.
* **Economia:** busca garantir valor para o negócio. Ao economizar pensamos sobre o valor do dinheiro o 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 e pague pelo uso e também na priorização e estimativa de histórias, garantindo que uma equipe XP não invista em flexibilidade especulativa.
* **Benefício mútuo:** é o princípio mais importante e difícil de seguir. Um exemplo de como ele se concretiza é a ênfase dada em XP a testes, refatorações e a metáfora no lugar de extensa documentação escrita. A documentação não traz benefício para os programadores no ato de sua criação, só organização em um possível futuro. Enquanto investir em testes, refatorações e na construção de uma metáfora trazem benefícios imediatos aos programadores, ao sistema, ao cliente e à organização.
* **Auto-semelhança:** diz que se deve copiar estruturas e processos existentes para resolver problemas em diferentes contextos ou escalas. É o caso do ritmo similar que se observa nas práticas do ciclo de estação, ciclo semanal e nas atividades diárias de programação. Observamos que primeiro criamos testes e depois trabalhamos para que eles funcionem. No ciclo mensal, escrevemos testes de aceitação, que no final devem todos passar para validarmos um *release*. Nas atividades diárias escrevemos testes de unidade para validar tarefas que devem ser codificadas.
* **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 evidente nas práticas do time completo e nos diferentes ciclos de planejamento. Pessoas com perspectivas diversas têm igual oportunidade de colaborar nestes: 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 da metodologia. Para isso, é preciso tempo para pensar. É importante socializar com a equipe em contextos de diversão ou até em refeições. A reflexão é evidenciada nas práticas do ciclo de estação e o 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 permiti-se que as etapas do desenvolvimento aconteçam em paralelo e não sequencialmente como proposto em metodologias mais tradicionais. As práticas de *releases* frequentes e integração contínua evidenciam isto.
* **Oportunidade:** nos leva a encarar problemas como oportunidades para mudança. Estar aberto a oportunidades de aprender e melhorar durante todo o processo é importante.
* **Redundância:** aumenta as nossas chances de sucesso, promovendo várias oportunidades de fazer a coisa certa. A redundância está presente na complementação das práticas, e nos testes de unidade automatizados: escrevemos o código fonte e, de maneira redundante, escrevemos mais código para verificar se o primeiro funciona, diminuindo a nossa 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 do XP

Extreme Programming define também práticas que tornam o processo viável e possível de seguir seus valores e princípios. As práticas são simples, porém o poder da metodologia 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 programação extrema 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 serão detalhadas as novas 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 seus sentidos ao conversar. Ressalta-se 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 seguir somente dois limites naturais: equipes de 12 pessoas (número de pessoas com as quais um indivíduo pode interagir em um dia) e equipes de 150 pessoas (quantidade de pessoas que permite que um indivíduo se lembre de todos outros no time).
* **Espaço de trabalho informativo:** era uma prática implícita que ganhou destaque na segunda edição. Ela 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 pareada e disponibilidade de água e comida garantem que o espaço informativo complemente a prática de trabalho energizado.
* **Histórias:** que antes faziam parte do jogo do planejamento passaram a ser uma prática independente na segunda versão. Elas continuam iguais: textos simples que expressam necessidades do cliente e que são estimadas, o quanto antes, pelos programadores.
* **Ciclo semanal:** que antes era parte do jogo do planejamento, 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 histórias, 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:** que antes era parte do jogo do planejamento, 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 e reparos. Durante o planejamento deste ciclo, temas são escolhidos para a estação. Estes temas servem para agrupar histórias relacionadas, que também são criadas neste momento. O uso de temas garante que não há exagero de detalhes e que o planejamento acontece com a perspectiva do longo prazo. 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 o planejamento, por melhor que tenha sido, sempre falha. Para evitar atrasos ou a necessidade de renegociação de escopo, o planejamento deve conter explicitamente espaços de folga. Isto pode acontecer tanto com a introdução de tarefas menores, que podem ser descartadas em caso de atraso, quanto com a distribuição de histórias de trabalho livre para os programadores. 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 descartada e mais trabalho pode ser realizado para que a equipe consiga entregar as histórias que prometeu ao cliente.
* ***Build* veloz:** exige que o sistema deve ser compilado por completo e todos os testes devem ser executados, de maneira automática, em no máximo 10 minutos. Esta prática provê agilidade 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, porém não é essencial. 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 rode 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 rendeu criticas a XP que diziam que a metodologia não investia em *design* da aplicação. Para responder às críticas, Beck (2004) a redeﬁne como *design* incremental. Esta prática implica em investimento diário no *design* da aplicação justiﬁcando seu ﬂuxo contínuo ao mostrar que, se realizado próximo de ser utilizado, ele tende a ser mais eﬁciente e valioso.

Dentre as práticas do XP, duas não sofreram modificações e compõem as primárias: a **programação pareada,** na qual todo o código é produzido por pares de programadores e cada par trabalhando na mesma máquina e a **integração contínua,** cujo 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 antes serem dominadas as práticas primárias. O conselho de progredir incrementalmente em direção às práticas se mantém. A prática do cliente sempre presente foi renomeada para envolvimento real com o cliente e assume que muitas vezes um cliente direto não poderá ser colocado com a equipe. A prática de propriedade coletiva do código foi renomeada para código compartilhado e se mantém como prática corolário. A seguir iremos detalhar as novas práticas.

* **Implantação incremental:** lida com uma equipe que deve substituir um sistema legado e diz que essa substituição deve ser feita incrementalmente. O importante é manter o sistema funcionando e ter segurança na migração. É possível que, durante algum tempo, ambos o legado e o novo sistema devam funcionar em conjunto, adicionando um pouco de trabalho de comunicação extra tanto no sistema quanto com usuários, mas garantindo harmonia na migração.
* **Continuidade da equipe:** propõe que equipes eﬁcientes continuem trabalhando juntas. Deve-se incentivar um rodízio razoável entre equipes, mas ao se concentrar na eﬁciência da organização como um todo, o valor de equipes que trabalhem bem juntas se torna evidente.
* **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 deixar alguém ocioso; esta pessoa pode ser liberada para formar novos times. Esta prática tende a eliminar o desperdício e ajudar a organização a resolver novos problemas.
* **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 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 que ele recorra.
* **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 histórias e cartazes devem ser descartados, pois o histórico do projeto se mantém por mecanismos sociais. A cerimônia envolvida em documentação interrompe o ﬂuxo de valor.
* **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 que não deve ter branches permanentes. O problema com múltiplos repositórios é que eles não escalam. 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 histórias 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 implantação incremental 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:** também aborda o lado de negócios de um projeto XP, sugerindo que o cliente pague por toda vez que for usar o sistema (trazendo à tona o debate sobre “arquiteturas orientadas a serviços” que está em voga na indústria atualmente). Esta prática valoriza o dinheiro como *feedback* mais importante e provê ao cliente possibilidades de prever custos.

O quadro abaixo ilustra e converge o entendimento das práticas da primeira versão do XP com a segunda.

Quadro 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 | Histórias  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 todos 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. Os principais papéis em XP são [BECK 2004]:

* **Programadores:** este é o coração de XP. Responsável por quebrar histórias 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 história. Estes treinam os programadores a utilizarem as ferramentas de teste.
* **Analista de Negócios:** trabalham com clientes para definir as histórias 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 histórias 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 a equipe.
* **Gerente do Produto:** escrevem e priorizam histórias 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 utilizar o sistema diariamente, ajudam o time a escrever e escolher de forma melhor as histórias do sistema. Por fornecerem as necessidades do sistema é ideal que tenham experiência com sistemas similares.

É importante notar que os papéis não são ﬁxos e que cada um deve 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. De acordo com o tipo de projeto ou a característica da organização, por exemplo, essas fases podem sofrer modificações, porém manterá forte semelhança estrutural. Em cada fase várias atividades são realizadas. Um projeto XP passa basicamente pelas seguintes fases: exploração, planejamento, iterações e aprovação. Abaixo descreveremos algumas das principais fases de um projeto e consequente visão de como ele acontece.

A fase de exploração é anterior à construção propriamente dita do sistema. 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. Ao tempo em que o cliente prepara as histórias. Conseguinte, os programadores e os clientes vão ganhando confiança, e quando eles possuírem histórias suficientes, passam a formatar o primeiro *release* do sistema.

A fase de planejamento atende o momento em que será acordado uma data lançamento do primeiro *release*. Nessa etapa os programadores de posse das histórias elaboradas pelo cliente assinalam certa dificuldade para cada uma e, baseados na sua velocidade de implementação, dizem quantas podem implementar em uma iteração, denominado de planejamento da iteração. Depois, os clientes escolhem as histórias 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 entre outras) e, ainda, submetido aos testes de aceitação. Os testes de aceitação já foram escritos a partir das histórias do cliente. À medida que esse 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 histórias acordadas para o *release* já em funcionamento. Nesse 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.

A figura 2.3 a seguir corrobora com o entendimento do ciclo de vida do produto XP.

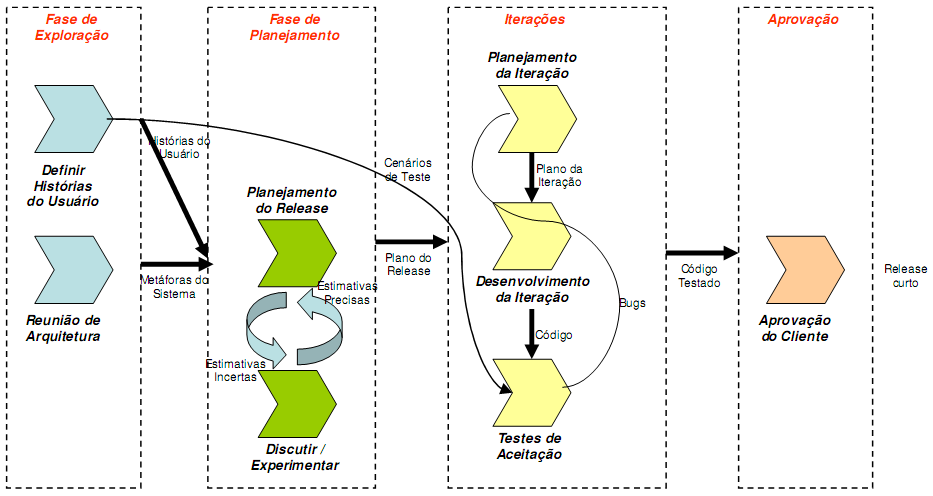
****

Figura 2.3 Ciclo de Vida do produto em XP [JÚNIOR 2008].

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, a medida que novas funcionalidades surgem, tecnologias e pessoas passam a incorporadas e o melhorando o código.

A fase de mortecorresponde ao término de um projeto XP. Um projeto XP 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

A metodologia ágil *Scrum* foi criada em 1996 por Ken Schwaber e Jeff Sutherland e destaca-se das demais metodologias 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 utilizando em organizações ao redor do mundo. Ele permite manter o foco na entrega do maior valor de negócio, no menor tempo possível permitindo à 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 de lidar com o caos, em detrimento a um processo bem definido. Focado em pessoas para ambientes em que há requisitos voláteis, resultando em uma abordagem que reintroduz as idéias de flexibilidade, adaptabilidade e produtividade. O foco da metodologia é encontrar uma forma de trabalho dos membros da equipe para produzir o software de forma flexível e em um ambiente em constante mudança.

A metodologia baseia-se em princípios como: equipes pequenas e multi-disciplinares, no máximo 7 pessoas, requisitos instáveis ou desconhecidos e iterações curtas. Cada ciclo do *Scrum* é denominado *Sprint*, que possui intervalos de tempo reduzido de 15 a 30 dias. Esta metodologia não requer ou fornece qualquer técnica ou método específico para o desenvolvimento do software, ela enfatiza o planejamento e gerenciamento dos projetos, através de um conjuntos de regras e práticas gerenciais que são estabelecidas.

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

### 2.5.2 Papéis no *Scrum*

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

**a) *Scrum Master***

O *Scrum Master* (SM) gerencia o processo, dissimulando o *Scrum* a todos os envolvidos no projeto e adequando a metodologia 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 Product Owner, responsável por: resolver barreiras entre o time e o PO; ensinar o cliente a aumentar o retorno sobre o investimento; 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 intra-time as informações sobre os progressos da equipe.

**b) Product Owner**

O *Product Owner* (PO) representa o cliente no projeto. Seu foco é na parte comercial do produto. Ele é o representante de todos os *stakeholders.* 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 e

Segundo Schwaber (2009), o *Product Owner* 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, com diferentes habilidades, necessárias para implementar as funcionalidades, envolve analistas, desenvolvedores, designers, gerente de qualidade, entre outros. Quando necessário, a equipe 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 responsáveis 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-se de antigas. Não há títulos em Times, e não há exceções a essa regra. Os Times também não contém sub-times dedicados a áreas particulares como testes ou análise de negócios.

### 2.5.3 Artefatos do *Scrum*

A seguir serão abordados os principais conceitos e artefatos utilizados na metodologia *Scrum*, que colaboram com o gerenciamento do projeto:

***a) Product Backlog***

1. 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 pelo cliente.
2. O *Backlog* do produto define tudo o que se tem conhecimento inicialmente, que seja necessário para o produto final. Ele é dinâmico, evolui 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.
3. 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.
4. ***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.

1. ***c) Burndown* da *Sprint***

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

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

Através do gráfico 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.

1. **d) Impedimentos**

Impedimentos é qualquer fator que impede 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 esses obstáculos. Quando o impedimento não pode ser resolvido durante a reunião diária, marca-se uma reunião 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 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 – 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 30 dias. É durante a *Sprint* que são executados os 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 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 tamanho 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 têm condições de avaliar a sua capacidade.

Já na segunda parte é debatido “como” será implementado as IBLs. O time lista as atividades necessárias para tornar funcional os itens de backlog, as quais devem ser decompostas para 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, elimina outras reuniões, 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 para 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. Deve ser listado, nesse momento, os pontos fortes da *Sprint* atual e o que podia ter sido melhor, e ao final identificado as 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*.* Éestabelecida 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 que está ilustrado na figura a seguir:

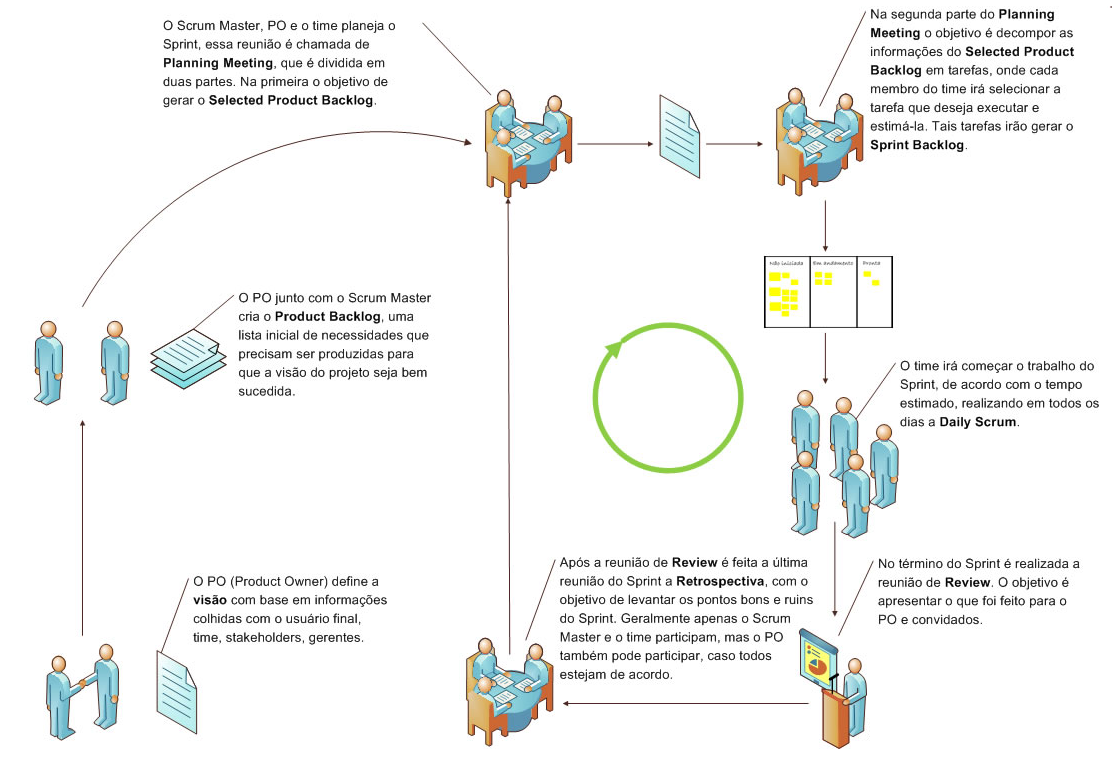


Figura 2.4 Ciclo de trabalho do *Scrum* [QUALIDADEBR 2009].

## 2.6 FEATURE DRIVEN DEVELOPMENT

**O *Feature Driven Development*** (FDD) é um processo ágil para gerenciamento e desenvolvimento de software, criado em 1977 em um grande projeto em Java para o *Unided Overseas Bank*, em Singapura. Nasceu a partir da experiência de análise e modelagem orientadas por objetos de Peter Coad e de gerenciamento de projetos de Jeff De Luca, frente a uma necessidade da referida instituição [SLIGER 2008].

A FDD é uma metodologia voltada para o cliente e orientada a modelagem, combinando algumas das melhores práticas do gerenciamento ágil de projetos com uma abordagem completa para Engenharia de Software orientada por objetos [COAD 1999]. Compõe de um arcabouço particular, através de seus princípios e práticas, que proporciona 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. Stephen Palmer (2002) coloca a FDD como sendo algo que será incorporado a sua empresa com fácil adaptação e que possibilitará resultados frequentes, tangíveis e funcionais.

### 2.6.1 Características do FDD

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

Algumas características intrínsecas nos processos ágeis são levantadas 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 as mudanças; f) flexibilidade; e g) simplicidade.

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

* Resultados úteis a cada duas semanas ou menos;
* Blocos bem pequenos de funcionalidade valorizada pelo cliente, chamados "*features*";
* Planejamento detalhado e guia para medição;
* Rastreabilidade e relatórios com incrível 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 poderão ser observadas e, também, outras novas identificadas ao longo das seções que abordam sobre o FDD.

### 2.6.2 Papéis no FDD

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

* **Gestor do Projeto:** trata das questões financeiras e administrativas do projeto. É o membro que decide sobre o escopo, objetivos, o time e prazos, no que se refere à decisão final. 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 das sessões de design, nas quais apresenta seus entendimentos ao time;
* **Gestor de Desenvolvimento:** acompanha as atividades de desenvolvimento do código diariamente, bem como a incumbência de fazer com que problemas não cheguem ao time ou que o mesmo seja resolvido o mais rapidamente. Desempenha suas funções afinado com o gestor de projeto;
* **Programador Chefe:** é responsável por uma equipe pequena no que se refere a 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 e fará parte das equipes cujas funcionalidades sejam envolvidas 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 necessários a construção da fature.

Por se tratar de uma metodologia ágil, na 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 a partir das características de cada projeto.

Como a proposta do FDD foi utilizada inicialmente em um time com aproximadamente 50 pessoas, pode fazer necessário o surgimento de outros papéis para compor o time dentro da característica do projeto proposto.

O FDD, inicialmente, recomenda uma composição de equipe de até 20 membros, mas existem casos conhecidos na literatura e na prática de indústrias de software, o processo atendendo a times bem maiores.

Alguns métodos ágeis, inclusive o FDD, afirmam se aplicar a qualquer projeto de desenvolvimento ágil, sem importar suas características [ABRAHAMSONN 2003].

### 

### 2.6.3 Práticas do FDD

O FDD possui em seu arcabouço um conjunto de boas práticas baseadas nas que identificamos e/ou vivemos na engenharia de software. As práticas do FDD focam em atender as necessidades do cliente e a 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 suas funcionalidades, dentro da perspectiva da orientação a objetos. Essa modelagem possibilita um maior entendimento/visibilidade do problema a ser resolvido;
* **Desenvolvimento por funcionalidade:** as atividades a serem desenvolvidas devem ser analisadas com a perspectiva de verificar a possibilidade de serem recompostas em atividades menores. Essa prática possibilita mais segurança, maior flexibilidade e escalabilidade ao código;
* **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 visão/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 um melhoramento do código no que se refere à redução de erros, melhor modelagem e fatores como legibilidade e alta coesão;
* **Gerência de configuração:** atividade que busca manter o controle sobre o código fonte, permite 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 a equipe o funcionamento de pelo menos uma versão do sistema. Essa prática garante que existirá uma versão do sistema que pode ser utilizada, a parte, a qualquer momento sem interferir no desenvolvimento; e
* **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 e demais entidades envolvidas.

### 

### 2.6.4 Ciclo de Vida do FDD

O FDD possui uma estrutura muito objetiva. A sua composição apresenta-se em duas fazes (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. A Figura 2.5 ilustra facilita o entendimento da descrição dos processos que será posteriormente feita.

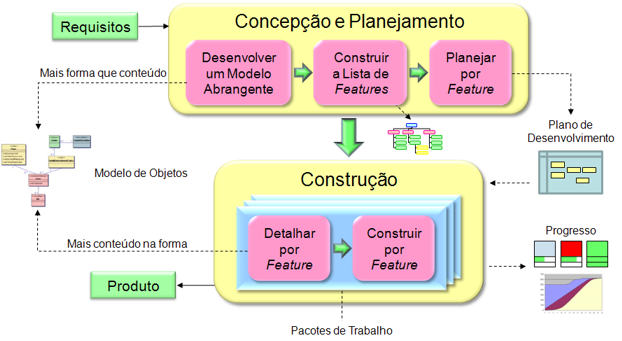


Figura 2.5 Ciclo de Vida do produto FDD [HEPTAGON 2009]

**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 a 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 colocadas e discutidas. Finalmente, os modelos são fundidos para gerar um modelo geral do domínio do sistema. Dentro desse processo existem algumas sub-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 gerado; e
* Escrever notas explicativas sobre o modelo final.

**Gerar uma lista de funcionalidades** recebe as práticas e o conhecimento adquirido no processo anterior e que são essenciais para esta fase. Nesta, será elaborada uma lista de funcionalidades do sistema decompondo as áreas de domínio obtidas. Cada funcionalidade é uma pequena tarefa a ser implementada que agregue valor ao cliente. Devem seguir o formato <ação> <resultado> <objeto>. 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** pregaque seja feito o planejamento de desenvolvimento de cada funcionalidade da lista obtida da fase anterior. Os programadores-chefe recebem classes ou trechos de código e serão responsáveis pelos mesmos. As sub-atividades desse 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 os diagramas de sequência e a modelagem completa das funcionalidades. Devemos observar que não se trata da mesma modelagem do primeiro processo, nesse momento a modelagem é feita para a funcionalidade em questão. Nessa fase 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 particulares a esse processo são:

* Formar uma equipe para a funcionalidade em questão;
* Estudar a funcionalidade como parte inserida no modelo de domínio;
* Estudar documentos relacionados á funcionalidade;
* Desenvolver diagrama de sequência;
* Refinar objeto modelo;
* Escrever as classes e as assinaturas dos métodos (tipo de retorno, parâmetros e exceções lançadas); e
* Realizar inspeção da modelagem.

**Construir por funcionalidade** vem consubstanciado pelo processo anterior e o programador-chefe designa um programador para desenvolver o código e finalmente ele passa a ser criado, os testes escritos e a funcionalidade ganha vida. Nesse momento temos as sub-atividades em execução, a saber:

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

Os três primeiros processos citados, que compõem a primeira fase, acontecem de forma sequenciada, lembrando os métodos tradicionais, porém os processos estão em uma mesma fase. Já os dois últimos, que estão na segunda fase do FDD, possuem uma dinâmica interativa e incremental. Isso reforça a afirmação feita por Michele Sliger no início da abordagem sobre FDD.

## 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 fez se pensar uma nova forma de desenvolver software, na qual se pudesse atender as principais demandas dos clientes – atender suas necessidades e com rapidez, redução de custos e qualidade merecida.

Este capítulo apresentou uma contextualização sobre o paradigma ágil, abrangendo desde a problemática e perspectivas que fomentaram o manifesto ágil até a descrição detalhada dos principais processos ágeis de desenvolvimento de software. Ao longo do conteúdo se faz evidente as principais características dos referidos processos, para que se possa analisar e escolher, adaptar e/ou fundir processos e práticas que possam atender uma demanda em particular. Ainda nessa linha de expor ao leitor características das metodologias ágeis, é disposta no quadro abaixo uma comparação proposta por Fagundes *et. al*. [2008], que vem a contribuir com o fechamento do entendimento, a saber:

Quadro 2.2 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 as 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, sem sugestões de 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**  **seqüência da UML.** |
| **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 no *Product Backlog* desenvolvidos.** | **O sistema é entregue após todos os conjuntos de características implementados.** |

No processo de construção da mensagem é oferecido ao leitor o norte que se configura o processo de desenvolvimento de software. Nesse sentido, é recomendada uma análise dos processos em confrontação com o problema e características da empresa e/ou do time de desenvolvimento e escolher a metodologia que melhor se adéqua. Outra configuração pode ser adotada que é a de coletar práticas de algumas e segui-las para se atender o objetivo final da prática de desenvolvimento – entrega de um produto de qualidade que atenda as expectativas do cliente.

Contudo, um estudo deve ser feito, uma pesquisa elaborada/executada e ao final evoluída a mensagem do manifesto ágil.

## 2.8 Tópicos de Pesquisa

* **Combinação 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 PADS. Com isso passam a combinar práticas de mais de um processo ágil ou juntá-los em sua completude para que o mesmo aconteça.
* **Desenvolvimento Distribuído de Software (DDS) com Metodologias Ágeis:** O DDS 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 combatidos/vencidos, os quais são categorizá-los em: pessoas, processos, tecnologia, gestão e, principalmente, comunicação. Ao tempo em que o estudo/uso de metodologias ágeis é crescente, seus resultados são muito positivos e estão preenchendo as lacunas deixadas pelos processos tradicionais. Frente a isso, demanda-se um casamento entre as duas abordagens de forma a se ter processos adaptados e/ou a criação de um único processo para atender a problemática do DDS.
* **Aproximação dos Processos Ágeis de Desenvolvimento de Software e dos Modelos de Qualidade de Software (MQS):** 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).

## 2.9 Sugestões de Leitura

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. Nesse endereço é possível encontrar outros artigos e recursos 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. E para se expandir no assunto a leitura do livro *A practical Guide to Feature Driven Development*. 2002, Stephen Palmer.

A base inicial dos processos ágeis poderá ser encontrada no endereço http://agilemanifesto.org. Nele são encontrados os princípios e valores propostos no Manifesto Ágil, além de disponibilizar detalhes dos autores e *links* para algumas metodologias 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. A *Agile Modeling* é uma metodologia 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 importante.
7. Caracterize a metodologia 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.

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

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*/](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.