UNIVERSIDADE FUMEC
FACULDADE DE CIÊNCIAS EMPRESARIAIS - FACE

MATHEUS RAMOS FERNANDES

SCRUM E XP:

Um comparativo no processo de desenvolvimento de software

BELO HORIZONTE
2011

MATHEUS RAMOS FERNANDES

SCRUM E XP:

Um comparativo no processo de desenvolvimento de software

Trabalho de Conclusão de Curso apresentado ao curso de Ciência da Computação da Universidade FUMEC, como exigência parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientadores:
Professor Osvaldo Manoel Corrêa
Professor Ricardo Terra

BELO HORIZONTE
2011

Agradeço aos orientadores Osvaldo Corrêa (normas técnicas)
e Ricardo Terra (conteúdo) por todo apoio e passagem de
conhecimentos durante a elaboração do trabalho.

RESUMO

Ao longo do tempo, as empresas estão se tornando cada vez mais dependentes da indústria de software, logo, ficam mais evidentes os problemas relacionados ao processo de desenvolvimento de sistemas como: alto custo, alta complexidade, dificuldade de manutenção e uma grande diferença entre os requisitos requisitados pelos usuários e o produto final entregue. Desenvolver um software não é uma tarefa trivial e uma possível solução é utilizar um método de desenvolvimento de software adequado ao negócio, para ganhar em qualidade e poder mensurar de forma mais precisa o tamanho e o custo do projeto, propiciando uma maior satisfação por parte do cliente e ainda ter a possibilidade de maximizar o lucro da empresa. Assim, este estudo apresenta diversos métodos tradicionais e duas metodologias ágeis, explicando seus principais conceitos e processos adotados, e realizando uma análise comparativa do *Scrum* e do *Extreme Programming* na fase de implementação. Como um resultado preliminar, estudos comprovam que é possível combinar práticas dessas duas metodologias ágeis e obter sucesso. Ainda assim, não se pode dizer qual é a melhor metodologia, mas sim, é possível analisar qual é a mais adequada.

Palavras chave: Engenharia de Software, *Scrum*, *Extreme Programming* (XP), Metodologias Tradicionais, Metodologias Ágeis, Fase de Implementação.

LISTA DE FIGURAS

SUMÁRIO

INTRODUÇÃO

Conforme relatado por Pressman (2006), em meados da década de 70, a atividade de desenvolvimento de software era executada de forma desorganizada, sem planejamento e desestruturada, resultando normalmente em um produto final de má qualidade e que não correspondia com as reais expectativas do contratante. A partir dessa realidade, algumas empresas começaram a perceber a necessidade de tornar a atividade de desenvolvimento de software um processo estruturado, padronizado e planejado.

Diante disso, ainda de acordo com Pressman (2006), surgiram muitas metodologias voltadas a esse tipo de desenvolvimento, além da criação de novas linguagens de modelagem com o intuito de facilitar o entendimento do produto tanto pela empresa desenvolvedora, como pelo cliente contratante.

O método de desenvolvimento de software tradicional mais utilizado é o modelo em Cascata, em que cada ciclo tem uma documentação padrão associada e que deve ser aprovada pelos envolvidos para que se possa avançar para as próximas etapas do projeto. Cada fase finalizada gera um marco no projeto, que geralmente é alguma versão do software, um protótipo ou mesmo documentação (NETO, 2004).

Como alternativa aos métodos tradicionais, surgiram as metodologias ágeis que segundo o manifesto ágil de Beedle (2001), sintetiza o consenso da comunidade de desenvolvedores de software sobre como trabalhar melhor em desenvolvimento de projetos. Ele privilegia indivíduos e interações sobre processos e ferramentas, colaboração com cliente sobre determinação de prazos e contratos, além de respostas à mudanças. Os principais métodos que se aplicam ao manifesto ágil são *Lean*, *Scrum* e *Extreme Programming*, sendo os dois últimos abordados neste estudo.

Uma das grandes vantagens das metodologias ágeis, segundo Cockburn (2001), é que elas são adaptativas ao invés de serem preditivas, isto é, elas se adaptam a novos fatores decorrentes do desenvolvimento de um projeto, ao invés de procurar analisar previamente tudo o que pode acontecer no decorrer do desenvolvimento.

Conforme apontado por Highsmith (2004), as metodologias ágeis variam em termos de prática e ênfases, mas, por outro lado, possuem características comuns como desenvolvimento incremental e interativo, comunicação e redução de produtos intermediários,

tal como documentação extensiva. Dessa maneira, existem várias possibilidades para atender os requisitos do contratante, que muitas vezes são mutáveis.

Apoiado por essas características comuns nas metodologias ágeis, ainda de acordo com Highsmith (2004), existem estudos que relatam a combinação do *Scrum* com o *Extreme Programming* trabalhando em conjunto e com sucesso. O autor também enfatiza que como o Scrum atua principalmente em gerência de projeto, isso favorece que o *Scrum* integre alguns métodos ou práticas do *Extreme Programming* em seu processo.

Assim, este estudo está organizado como a seguir: No Capítulo 1, são descritos diversos métodos tradicionais como modelo em Cascata (o pioneiro no conceito de engenharia de software), Prototipação, modelo Incremental, *Rapid Application Development (*RAD) e modelo Espiral. Em seguida no Capítulo 2 são apontadas duas metodologias ágeis, *Scrum* e *Extreme Programming*, apresentando suas principais características, assim como, métodos e práticas utilizadas. Por último, no Capítulo 3, é apresentada uma análise comparativa entre tais metodologias ágeis na fase de implementação, em que são destacados pontos em comum, contrastando algumas diferenças e semelhanças, além de abordar também as atividades típicas de um processo de desenvolvimento de software e o processo adotado na fase de implementação dessas duas metodologias ágeis.

1. METODOLOGIAS TRADICIONAIS

Antigamente, em meados da década de 70, os projetos de desenvolvimento de software eram conduzidos de forma desorganizada, sem planejamento e desestruturados, gerando, grande parte das vezes, um produto final de má qualidade. Segundo Pressman (2006), as crises que ocorreram na época, forçaram as empresas a desenvolverem seus projetos de uma forma padronizada, gerando uma documentação que acompanhe o produto que está sendo desenvolvido.

Diante disso, ainda de acordo com Pressman (2006), surgiram muitas metodologias voltadas a esse tipo de desenvolvimento, além da criação de novas linguagens de modelagem com o intuito de facilitar o entendimento do produto tanto pela empresa desenvolvedora, como pelo cliente contratante.

Com essa realidade, surgiu a necessidade do projeto de desenvolvimento de software ser executado de forma estruturada, planejada e padronizada, fazendo com que as necessidades fossem atendidas e o investimento realizado fosse retornado. Assim, surgiram metodologias que dividem o processo em fases pré-definidas com foco sempre na qualidade final do produto.

Segundo Sommerville (2007), os processos eram sempre iniciados com uma fase de análise, em que se estabelecia os requisitos de acordo com as necessidades do cliente. Em seguida, era realizada a modelagem, que gerava a documentação necessária para a próxima etapa. Por último, tem-se as etapas de desenvolvimento e testes.

O sucesso de um projeto depende de diversos fatores, tais como a grande complexidade e desorganização em seu desenvolvimento. A negligência em alguns desses aspectos podem ressaltar no fracasso do projeto.

Como podemos observar na FIG. 1, segundo Chaos (2009), apesar da tendência crescente de PMOs em empresas, o resultado não foi animador. Algumas pessoas argumentam que estão apenas medindo orçamento, tempo e espaço (deixando de fora qualidade, risco e satisfação do cliente).

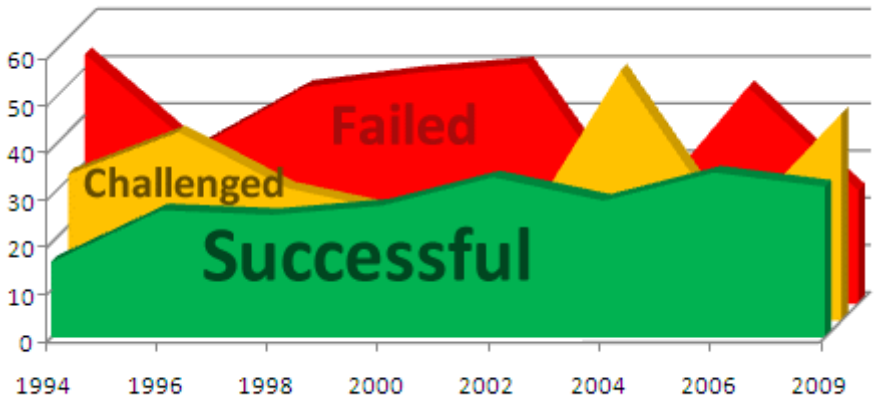

Figura 1 – Gráfico CHAOS
Fonte: Chaos (2009)

Segundo Soares (2004), as metodologias tradicionais são conhecidas como orientadas a documentação ou pesadas. Possuem como característica principal dividir os processos de desenvolvimento em fases ou etapas definidas. De acordo com Sommerville (2007), muitas metodologias orientadas a documentação são desenvolvidas em cima de modelos lineares, como o cascata.

No modelo linear, cada ciclo tem uma documentação padrão associada e que deve ser aprovada pelos envolvidos para que se possa avançar para as próximas etapas do projeto. Cada fase finalizada gera um marco no projeto, que geralmente é alguma versão do software, um protótipo ou mesmo documentação (NETO, 2004).

Neste capítulo são apontadas as diversas metodologias tradicionais. Assim, na seção 1.1, é apresentado o modelo Cascata que é conhecido como modelo Linear e também o modelo pioneiro e mais utilizado no conceito de engenharia de software segundo Sommerville (2007). Na seção 1.2, é introduzido a Prototipação que é mais utilizado quando os requisitos não estão definidos de forma clara pelo contratante e propõe-se a gerar protótipos do software com definições de requisitos fornecidas pelo próprio cliente.

Na seção 1.3, é apresentado o modelo Incremental, que é uma versão mais evoluída do modelo Cascata, segundo Pressman (2006). A principal diferença em relação ao modelo Cascata, é que o Incremental define que o sistema desenvolvido pode sempre ganhar novas funcionalidades e crescer gradativamente. Na seção 1.4, é descrito o modelo RAD, que segundo Sommerville (2007), é considerado um modelo Incremental, porém adaptado para tipos de projetos mais curtos.

A próxima seção é a 1.5, que fala sobre o modelo Espiral, que segundo Pressman (2006), é parecido com o modelo de Prototipagem e também com a metodologia linear, pois, além de ser iterativo, também é sistemático e permite o acompanhamento da evolução do software. Na última seção, a 1.6, é apresentada as considerações finais relacionadas ao primeiro capítulo como um todo.

1.1. Cascata

Esse modelo foi o primeiro largamente utilizado no conceito da engenharia de software. Por volta da década de 70, quando as empresas que desenvolviam software começaram a perceber a sua falta de eficiência na construção de sistemas, Royce indicou uma abordagem linear e pragmática que as empresas usavam incessantemente no desenvolvimento de todos os tipos de sistemas, segundo Sommerville (2007).

No modelo Cascata, o produto final é obtido através da execução de ciclos sistematicamente definidos, como pode ser observado na FIG. 2. O padrão segue, linearmente, as seguintes etapas: engenharia do sistema, análise de requisitos, geração de código, testes e manutenção, segundo Pressman (2006). Em cada uma dessas etapas, uma série de atividades pré estabelecidas são realizadas de forma que os artefatos produzidos de cada fase, sejam a entrada para a próxima fase. De acordo com o mesmo autor, cada etapa possui as seguintes características:

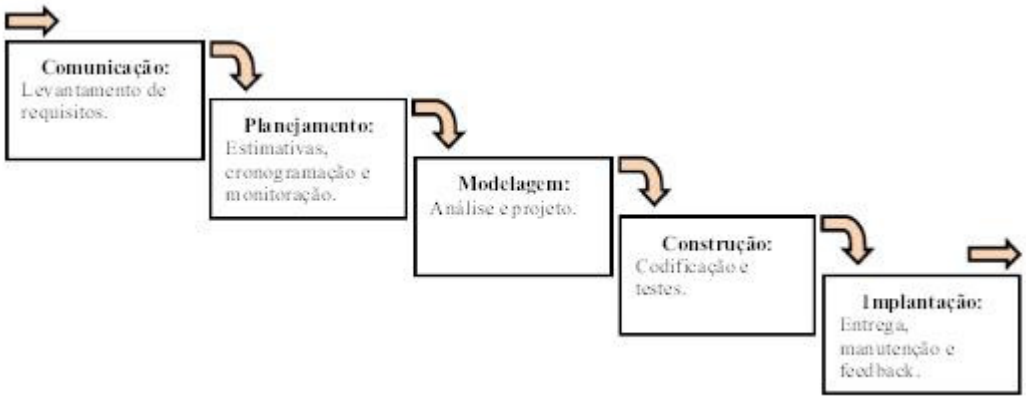

Figura 2 – Modelo em Cascata
Fonte: Pressman (2006)

a) Comunicação: Consiste na captura e na análise dos requisitos de alto nível e de ferramentas para definir as possíveis limitações e os requisitos não funcionais;

b) Planejamento: Constitui a identificação das funcionalidades requeridas para o software. As necessidades de interface, layout, os tipos de interação e todos os outros fatores que determinam a maneira como outros sistemas e usuários irão utilizar o novo programa devem ser identificados, revistos pelo contratante e documentados. Nesta etapa, é obtido conhecimento detalhado sobre a regra de negócio que o sistema propõe-se a tratar;

c) Modelagem: Define-se propostas de soluções para arquitetura, estruturas de dados, modelagem e os dispositivos lógicos entre os elementos do sistema. Essas características de implementação, devem atender aos níveis mínimos de qualidade pré estabelecidos, bem como às suas respectivas restrições, além de cobrir todos os requisitos e funcionalidades identificados nas fases anteriores;

d) Construção: Corresponde a implementação do código conforme a especificação, além dos testes das funcionalidades prontas. Todo requisito é testado de acordo com a sua especificação. Esses testes podem ser de dois tipos: caixa branca, que testa o software detendo o conhecimento de sua implementação e, caixa preta, onde o teste é baseado apenas nas interfaces oferecidas pelo software;

e) Implantação: Consiste na alteração da versão do software, que irá substituir à atual. Após a versão homologada ir para produção, o sistema retrocede para alguma das etapas anteriores para a criação de novas funcionalidades ou ajuste de algum requisito existente.

Um dos problemas do modelo clássico é a divisão do projeto em ciclos distintos, que dificulta futuras alterações dos requisitos por parte do contratante. Segundo Sommerville (2007), esse modelo é mais apropriado somente quando os requisitos forem bem compreendidos. Embora seja o mais usado pelas empresas no mercado, ele não é o mais adequado, justamente porque raros projetos de sistemas seguem um fluxo linear.

1.2. Prototipação

Segundo Sommervile (2007), o modelo de prototipagem ilustrado na FIG. 3, é mais utilizado quando os requisitos de sistema não estão definidos de maneira clara pelo contratante, mas sim, somente quando os objetivos do sistema de software são descritos, porém não a forma com que a informação será processada e como será mostrada a saída.

De acordo com Pressman (2006), a característica marcante do modelo de prototipagem é a geração de protótipos sistemas de software com definições de requisitos fornecidas pelo próprio cliente. Feito isso, o protótipo é revisado para ser repassado ao cliente, que irá testar e validar as suas funcionalidades. Após aprovado, o ciclo de desenvolvimento do protótipo vai se repetindo, com a inclusão das novas funcionalidades.

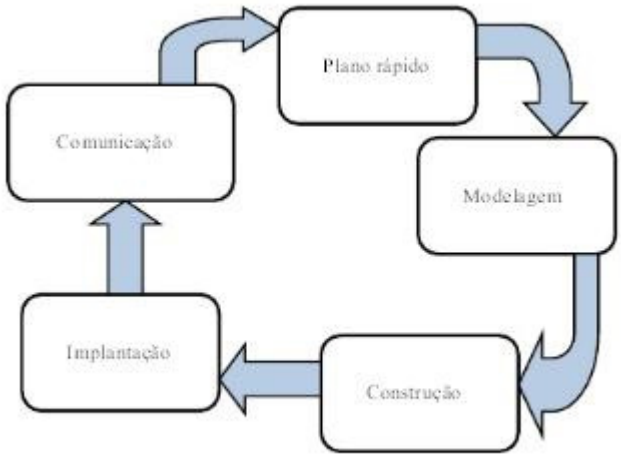

Figura 3 – Prototipação
Fonte: Pressman (2006)

Nesse modelo, o engenheiro de software, elemento que é responsável por realizar todo o levantamento das necessidades informadas pelo cliente, faz a coleta das novas funcionalidades acompanhado do contratante, geralmente por meio de reuniões presenciais ou questionários preenchidos pelo cliente, para então serem desenvolvidas e acrescentadas ao protótipo existente. Os resultados são apresentados ao cliente o mais rápido possível e o cliente pode ainda vislumbrar a evolução do sistema ao final de cada iteração.

Conforme relatado por Soares (2004), quando as maiorias das funcionalidades que o software deveria dispor estiverem desenvolvidas e aprovadas, o protótipo deve ser descartado para que o projeto verdadeiro de fato seja implementado com fundamento no que foi especificado na criação dos protótipos. É relevante, nesse momento, também dar ênfase nos requisitos não funcionais, tais como: manutenabilidade, segurança e qualidade. Encontrando uma solução ideal para os possíveis problemas.

Segundo Pressman (2006), por se tratarem de protótipos, muitas vezes a solução escolhida para resolver determinado problema não é a melhor, porque não levam em consideração as variáveis de cada ambiente, como a linguagem de programação adequada, ou o sistema operacional onde o sistema será executado. Os desenvolvedores podem desenvolver

o mal hábito da prática de utilizar o protótipo criado como parte integral do software, deixando possíveis erros que serão encontrados mais tarde.

1.3. Incremental

De acordo com Pressman (2006), esse modelo é uma versão mais evoluída do modelo Cascata. A diferença com o modulo anterior é que o modelo Incremental define que o sistema desenvolvido pode sempre ganhar novas funcionalidades e crescer, sendo que o conjunto desses novos requisitos será desenvolvido por um incremento que segue todos os ciclos descritos no modelo Cascata, como é exibido na ilustração da FIG. 4 a seguir.

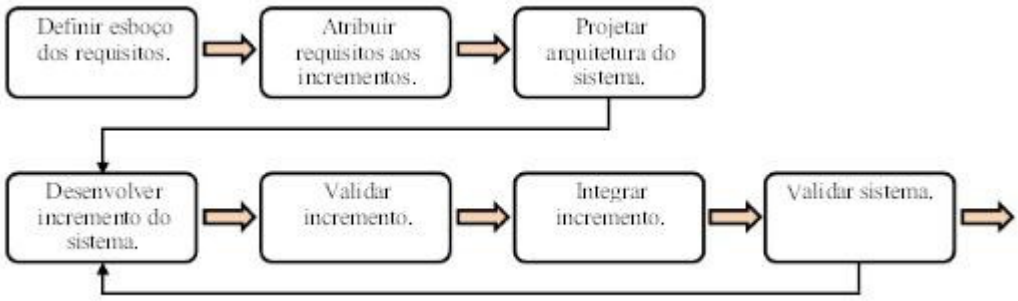

Figura 4 – Modelo Incremental
Fonte: Sommerville (2005)

O incremento inicial desse modelo leva o nome de n*úcleo do produto*, porque detêm das principais funcionalidades do software, as que são consideradas vitais para o correto funcionamento. Os incrementos seguintes, vão ser constituídas pelas funções do *núcleo do produto* e dos incrementos subseqüentes desenvolvidos. O modelo Incremental é análogo a uma bola de neve que vai crescendo até atingir a sua totalidade quando o projeto está totalmente desenvolvido, segundo Pressman (2006).

Conforme Soares (2004), essa metodologia é mais utilizada em projetos que demandam muito tempo para serem desenvolvidos e que possuem uma tendência de crescer durante o passar do tempo, além do prazo de entrega ser encurtado. É possível mostrar para o cliente como está o andamento do projeto, pois no fim de cada incremento, é criada uma versão do que será o produto final, o que ajuda a atingir as metas pré-estabelecidas no início do projeto, possibilitando maior controle dos prazos e recursos.

1.4. RAD

A metodologia RAD, segundo Sommervile (2007), também é considerado um modelo incremental, porém adaptado para tipos de projetos mais curtos, geralmente com prazo girando em torno de três meses. A principal característica desse modelo é que o produto final seja desenvolvido em componentes, para que a equipe possa desenvolver um sistema de software completamente funcional em um curto espaço de tempo, através da reutilização de códigos dos componentes previamente desenvolvidos.

Semelhante aos outros modelos de desenvolvimento de software, o modelo RAD ilustrado na FIG. 5, também é subdividido em fases:

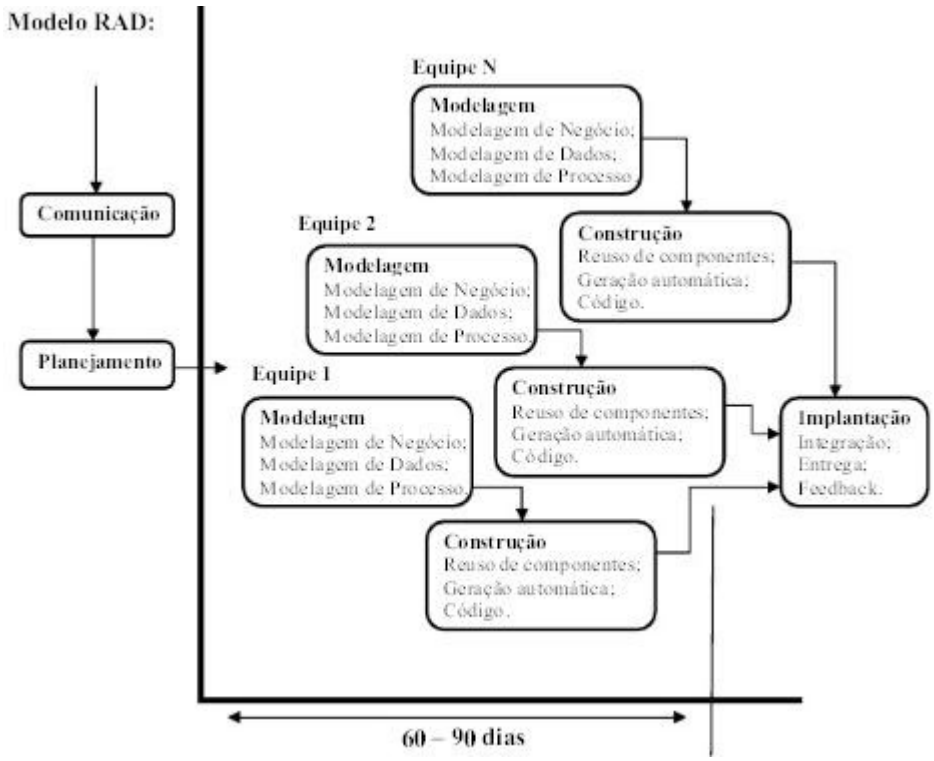

Figura 5 – RAD
Fonte: Pressman (2006)

a) Comunicação e Planejamento: A comunicação trabalha para entender os problemas do negócio, e o planejamento para gerenciar o trabalho paralelo das equipes;
b) Modelagem de negócios: Nessa fase, definem-se as arquiteturas que permitem ao software utilizar informações coletadas de maneira efetiva;

c) Modelagem de dados: Corresponde a todas as questões que estão relacionadas a objetos de dados que utilizam diagramas de entidade e relacionamento DER;
d) Modelo de processo: Esse ciclo é responsável por descrever todos os processos básicos para manipular as informações pertencentes aos objetos de dados que foram gerados na fase anterior, gerando dessa forma, um fluxo de informações;
e) Construção: Utilização de ferramentas para reutilização dos componentes desenvolvidos e para criação de novos componentes;
f) Implantação: Na fase final do modelo RAD, a integração entre todos os componentes utilizados no sistema, tanto os que foram desenvolvidos para esse projeto, quanto os componentes criados em sistemas anteriores e utilizados no projeto atual, são testados juntamente com as novas interfaces.

Como a principal característica da metodologia RAD é do produto ser desenvolvido de forma modularizada, é necessário que todos os requisitos estejam bem definidos e sejam o máximo possível, independentes entre eles. Se umas dessas premissas não forem respeitadas, o modelo pode não ser adequado segundo Neto (2004).

Além disso, segundo Soares (2004), para trabalhar com RAD de forma eficiente, é preciso que exista uma porção de recursos necessários para criar as equipes RAD de desenvolvimento. Esse impedimento pode ser interpretado como um grande impedimento em projetos maiores, que demandam mais tempo para serem entregues ao contratante, porque são mais complexos de gerenciar.

1.5. Espiral

De acordo com Pressman (2006), o modelo Espiral é parecido com o modelo de prototipagem, pois também é iterativo e não deixa de ser semelhante ao modelo Cascata, por também ser sistemático. Essas características facilitam com que versões amigáveis do que será o produto final sejam lançadas ao fim de cada iteração desse modelo, semelhante ao modelo Incremental. Essa metodologia é bastante utilizada no desenvolvimento de sistemas em união com o paradigma de orientação de objetos.

Ainda segundo o mesmo autor, esse método se divide em partes denominadas regiões de tarefa, que abrigam uma soma de tarefas e crescem de acordo com o tamanho do projeto e o risco. Nem todas as regiões de tarefas podem, ou devem ser necessariamente

utilizadas em um projeto. Constituem-se, sucintamente, por seis regiões que estão listadas na FIG. 6:

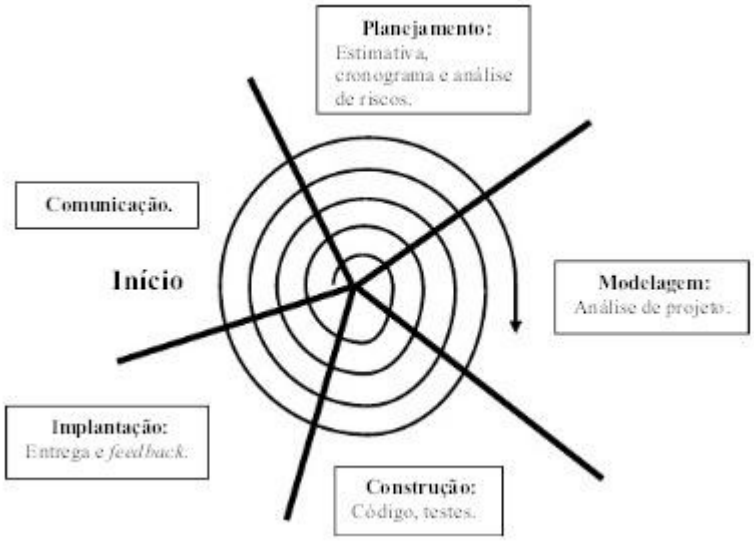

Figura 6 – Espiral
Fonte: Pressman (2006)

a) Comunicação com o cliente: Obter informações que podem ser úteis ao software, através do contato com o cliente;
b) Planejamento: Definição de custo e tempo de desenvolvimento;
c) Análise de risco: Corresponde ao gerenciamento e a recuperação dos riscos;
d) Engenharia: Nessa região acontece a criação da representação da aplicação;
e) Construção e Liberação: Responsável por implementar, testar, instalar e prover suporte a aplicação, além da avaliação realizada pelo contratante. É nessa região que obtém-se o feedback do incremento desenvolvido.

A principal vantagem desse modelo em contrapartida a outros modelos e que também faz com que seja bastante utilizado pelas empresas é que pode-se acompanhar a evolução do sistema de software. Isso é importante, pois segundo Sommerville (2007), muitas empresas focam na etapa de manutenção. O modelo Espiral também é considerado o mais realístico dentre os outros modelos, porque admite que analistas, desenvolvedores e usuários ganham maior conhecimento sobre o sistema de software ao longo de seu desenvolvimento.

1.6. Considerações finais

Quando o software era desenvolvido de forma totalmente desorganizada, sem planejamento e acompanhamento, o resultado era um produto final com péssima qualidade e que não correspondia com as expectativas do contratante. Diante disso surgiram as metodologias tradicionais de desenvolvimento de software, que aproveitaram alguns conceitos de Engenharia Civil como ferramentas de comunicação, aprendizado e principalmente como um meio de organizar seus pensamentos que ajudaram a sistematizar o processo de desenvolvimento de software.

Com o passar do tempo, as empresas passaram a se tornar cada vez mais dependentes da indústria de software. Com isso, cada vez mais ficaram evidentes os problemas relacionados ao processo de desenvolvimento de software como: maior complexidade, alto custo, dificuldade de manutenção e uma diferença entre as necessidades dos usuários e o produto final entregue.

Muitas metodologias tradicionais são desenvolvidas encima do modelo Cascata, que é um modelo linear em que cada fase tem uma documentação padrão associada e que deve ser aprovada pelos envolvidos para que se possa avançar para as próximas etapas do projeto. O fim de cada fase gera um marco do projeto, que pode ser alguma versão do software, um protótipo ou mesmo documentação.

O grande empecilho dos modelos tradicionais é que eles são divididos em fases distintas, que acaba deixando mais complexo as possíveis alterações de requisitos que o cliente pode necessitar. Mesmo ainda sendo as metodologias mais utilizadas no mercado, como o Cascata, não se pode afirmar que são as mais adequadas, até porque raros projetos seguem um fluxo linear. Com isso, as alterações nas documentações se tornam ainda mais complexas, pois é necessário alterar toda a documentação desenvolvida, demandando mais tempo e implicando em retrabalho.

Devido a necessidade constante dos clientes em alterar os requisitos e a dificuldade em alterar a documentação previamente desenvolvida, começaram a surgir metodologias mais simplificadas de desenvolvimento de software, que visam retirar uma parte da burocracia associada às atividades de desenvolvimento. Essas novas metodologias vêm ganhando força na Engenharia de Software e são apresentadas no próximo capítulo.

2. METODOLOGIAS ÁGEIS

As técnicas de metodologias ágeis para desenvolvimento de sistemas são uma resposta as metodologias tradicionais, também conhecidas como metodologias pesadas ou orientadas a documentação, segundo Beedle (2001). Mesmo com a evolução das técnicas, ferramentas, e inclusive dos computadores nos últimos anos, ainda era muito difícil entregar um sistema de software de qualidade dentro dos prazos e custos estimados.

Segundo Campos e Fonseca (2008), uma das grandes dificuldades das empresas é manter a equipe inteira motivada e comprometida, utilizando metodologias tradicionais de desenvolvimento de software. A utilização das metodologias tradicionais tem resultado no não cumprimento do prazo de entrega do produto final, não atendimento à expectativa do contratante, baixa qualidade do produto entregue e construção de várias funcionalidades desnecessárias.

De acordo com Fowler (2001), o processo utilizado no desenvolvimento do software é um dos motivos para a ocorrência desses problemas. Um segmento crescente na Engenharia de Software vem defendendo a adoção de processos mais simplificados conhecidos como metodologias ágeis, que visam retirar uma parte da burocracia associada às atividades de desenvolvimento.

O manifesto ágil de Beedle (2001) sintetiza o consenso da comunidade de desenvolvedores de software sobre como trabalhar melhor em desenvolvimento de projetos. Ele privilegia indivíduos e interações sobre processos e ferramentas, colaboração com cliente sobre determinação de prazos e contratos, além de respostas e mudanças sobre como seguir um plano. Os principais métodos que se aplicam ao manifesto ágil são Lean, Scrum e XP.

O estabelecimento do manifesto ágil não recusa as ferramentas, processos, planejamento, documentação ou negociação de contratos, mas os colocam em segundo plano de importância, atrás dos indivíduos, software, interações, colaboração do cliente e respostas eficientes a alterações e mudanças. Foram estabelecidos quatro conceitos chaves no manifesto ágil, são eles:

a) Indivíduos e interações ao invés de processos e ferramentas;
b) Software executável ao invés de documentação;
c) Colaboração do cliente ao invés de negociação de contratos;
d) Respostas rápidas a mudanças ao invés de seguir planos.

Uma das grandes vantagens das metodologias ágeis, segundo Cockburn (2001), é que elas são adaptativas ao invés de serem preditivas, isto é, elas se adaptam a novos fatores decorrentes do desenvolvimento de um projeto, ao invés de procurar analisar previamente tudo o que pode acontecer no decorrer do desenvolvimento.

Conforme apontado por Highsmith (2004), as metodologias ágeis variam em termos de prática e ênfases, mas, por outro lado, possuem características comuns como desenvolvimento incremental e interativo, comunicação e redução de produtos intermediários, tal como documentação extensiva. Dessa maneira, existem várias possibilidades para atender os requisitos do contratante, que muitas vezes são mutáveis.

O Scrum foi nomeado por Takeuchi no artigo *The New New Product Development Game* (1986). Após pesquisar diversas empresas privadas de sucesso no mercado, Takeuchi notou um padrão em projetos que tratam mudanças de requisitos e uma definição atrasada de requisitos como parte do processo de desenvolvimento de produtos, executado por equipes multidisciplinares. Um estudo sobre o estado do desenvolvimento ágil indicou que 70% dos entrevistados utilizam a metodologia Scrum combinada com outras metodologias e 40% utilizam apenas o Scrum de acordo com Versionone (2007).

Segundo Beck (2001), o XP é uma técnica de desenvolvimento de software que prioriza o código fonte executável. Esse modelo tem como uma característica marcante o nivelamento da equipe e a redução de uma única pessoa detendo conhecimento dentro de um determinado projeto. As práticas do XP podem ser aplicadas uma de cada vez para que sejam evitadas confusões, pressões e desentendimentos, pois uma equipe sobre pressão pode tênder a voltar a aplicar as metodologias do passado.

Segundo Boehm (2006), as organizações de todo o mundo têm despertado interesse em uma nova abordagem para desenvolvimento de software. O autor enfatiza também que existe uma tendência para o desenvolvimento ágil devido ao ritmo acelerado de mudanças na tecnologia da informação, concorrência acirrada, grande dinamismo no ambiente de negócio e pressões por constantes inovações.

De acordo com Cockburn (2001), apesar do uso crescente das metodologias ágeis, ainda falta uma base maior de projetos para verificar suas vantagens. No entanto, mesmo sem tal base histórica, que somente virá com o tempo e com mais estudos aprofundados, os resultados iniciais são satisfatórios e promissores em termos de custo, prazo de entrega e qualidade do produto final.

Ainda segundo o mesmo autor, além dos processos orientados a documentação possuírem fatores limitadores aos desenvolvedores, muitas organizações não têm recursos

para processos pesados de sistemas de software. Com isso, muitas organizações acabam por não utilizar algum processo, o que pode levar a efeitos desagradáveis em termos de qualidade de software.

Agilidade consiste na habilidade de criar e responder às mudanças, buscando a obtenção de lucro em um ambiente de negócio turbulento segundo Highsmith (2004). O autor também enfatiza que a falta de estabilidade ou a ausência de estrutura podem levar ao caos, mas que a estrutura em exagero gera rigidez desnecessária.

De acordo com Becker, Huselid e Ulrich (2001), vive-se um novo paradigma econômico que estabelece a necessidade da preocupação com a satisfação do cliente, atualização contínua e da geração de produtos e serviços com um maior valor agregado. A maioria das técnicas de metodologias ágeis não possui novidade alguma. Segundo Cockburn (2001), o que as diferencia das metodologias orientadas a documentação é o foco nas pessoas ao invés de nos algoritmos ou processos. Além da preocupação de gastar menos tempo com documentação e mais tempo com implementação.

Conforme Beck (2001), uma característica relevante das metodologias ágeis é que elas se adaptam às mudanças que acontecem no desenvolvimento do projeto, ao contrário de buscar analisar previamente tudo o que pode acontecer no transcorrer do desenvolvimento do sistema. De fato, o problema não é a mudança como um todo, até porque isso ocorre de qualquer forma, mas sim, como receber, avaliar e responder as mudanças. As metodologias tradicionais são mais indicadas apenas quando os requisitos do sistema são bem definidos, estáveis e com os requisitos futuros bem previsíveis.

Neste capítulo são apontadas duas metodologias ágeis, *Extreme Programming* e *Scrum*. Assim, na seção 2.1, é apresentado a metodologia *Scrum* que possui como objetivo fornecer um processo conveniente para o desenvolvimento orientado a objetos. Possui uma abordagem empírica e utiliza idéias da teoria de controle de processos da indústria para o desenvolvimento de sistemas de software.

Na seção 2.2, é apresentado a metodologia *Extreme Programming* que é aplicada com maior relevância em projetos curtos ou com equipes de trabalho co-localizadas. Ela apresenta uma visão parecida sobre as devidas práticas necessárias no desenvolvimento de um sistema e a obtenção da qualidade. Na última seção, a 2.3, é apresentada as considerações finais relacionadas ao segundo capítulo.

2.1. Scrum

O *Scrum* é um método de desenvolvimento de sistema de software ágil que apresenta uma longa comunidade de usuários de acordo com Cockburn (2001). O principal objetivo desse método é oferecer um processo conveniente para projeto e desenvolvimento de sistema de software orientado a objeto. Esse método apresenta uma abordagem empírica e utiliza ideias da teoria de controle de processos da indústria para o desenvolvimento de sistemas de software, aplicando o processo de flexibilidade, produtividade e adaptabilidade.

Segundo Schwaber (2004), o principal objetivo do *Scrum* é o desenvolvimento de sistemas de software envolvendo uma porção de variáveis técnicas e de ambiente, como requisitos, tecnologia e recursos, que podem mudar durante o processo. O foco desse método é encontrar uma maneira dos membros da equipe trabalharem para produzir o sistema de software de forma flexível e em um ambiente passível de sofrer constante mudança. O resultado desse trabalho deve ser um sistema de software que será realmente útil para o contratante.

Ainda segundo o mesmo autor, o *Scrum* é o mais perplexo e paradoxal processo para gerenciamento de projetos complexos, porém, é um método incrivelmente simples. Ele não é um processo prescritivo, também não descreve o que fazer em cada circunstância. Ele é usado para trabalhos complexos nos quais é impossível predizer tudo que irá acontecer durante o desenvolvimento do sistema de software.

Segundo Beedle (2001), o *Scrum* é um método de desenvolvimento ágil que procura uma forma empírica de lidar com o caos, em detrimento a um processo bem definido. Sua função primária é ser utilizado em gerenciamento de projetos de desenvolvimento de sistemas de software. Ele tem sido utilizado com sucesso para esse fim, assim como o *Extreme Programming (XP).* Além de tudo, esse método pode ser aplicado em qualquer situação em que um grupo de pessoas precisa trabalhar junto para atingir um objetivo comum, por exemplo, uma festa de aniversário.

De acordo com Highsmith (2004), um processo de desenvolvimento de sistema de software extremamente definido tem uma probabilidade de sucesso drasticamente reduzida quando utilizado no desenvolvimento de um produto. Para o autor, o *Scrum* utiliza uma implantação de um controle descentralizado que lida de forma eficiente com situações menos previsíveis, o que aparece como uma possível solução para atacar esse problema.

Conforme apresentado por Schwaber (2004), o *Scrum* é um modelo de desenvolvimento ágil de sistema de software que permite manter o foco na entrega de maior

valor de negócio e no menor tempo possível. Com isso, permite-se a rápida e contínua inspeção do sistema em produção, em intervalos de duas a quatro semanas conhecidos como *sprints*. Esse método é dividido sucintamente em três papéis:

a) *Product Owner*: É o representante de todos envolvidos e responsável por listar as prioridades e os requisitos. Juntamente com outros envolvidos, ele é o responsável por revisar e aprovar as entregas ao final de cada *sprint*;
b) *Scrum Master*: É o gerente do projeto, responsável por garantir a aplicação das práticas e valores do *Scrum*, assim como repassar os ensinamentos do método aos outros membros do projeto. As suas principais responsabilidades são remover os obstáculos, conduzir o *daily scrum*, revisar cada *sprint*, intermediar a comunicação entre o time e o *product owner*, etc;
c) *Scrum Team*: Correspondem aos membros encarregados de realizar as atividades do projeto. Suas principais atividades são organizar e gerenciar suas próprias atividades e geralmente são dedicados integralmente ao projeto.

Schwaber (2004) apresenta alguns novos artefatos em relação às metodologias anteriores:

a) *Product Backlog*: Corresponde a lista de requisitos e atividades do projeto que foram priorizados pelo *product owner* e possuem tempo estimado de entrega. Normalmente as estimativas são em dias, sendo que para casos mais prioritários os prazos são mais precisos. Os requisitos podem sofrer mudanças de acordo com a necessidade do projeto e qualquer membro do time pode alterar os itens com a autorização do *product owner*. Podem ser requisitos funcionais ou não-funcionais, sendo que os itens são selecionados para o próximo *sprint* de acordo com a prioridade. Também podem ser atualizados pelo *product owner* a qualquer momento, que detém a responsabilidade desse artefato;
b) *Release Burndown*: É considerado o principal gráfico de controle do *Scrum* e representa o trabalho restante dentro do *sprint* de uma versão do sistema que será entregue. No exemplo mostrado na FIG. 7, o eixo horizontal do gráfico exibe as iterações e o eixo vertical a quantidade de trabalho restante. Segundo Mountain (2011), esse tipo de gráfico funciona bem em muitas situações e com diversas equipes, porém, em projetos com muitas mudanças de requisitos, talvez tal gráfico não seja adequado;

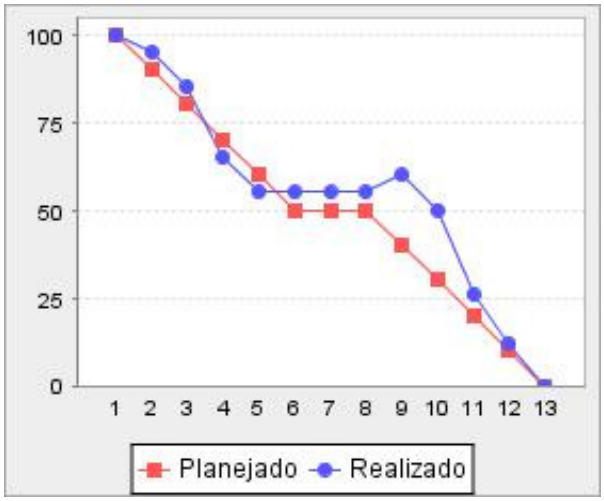

Figura 7 – *Release Burndown*
Fonte: Mountain (2011)

c) *Taskboard*: É um grande painel que possibilita colocar várias informações relevantes para o acompanhamento do *sprint*. Como pode ser visualizado na FIG. 8, o *product backlog* (*Story*), as atividades do *sprint* juntamente com o seu estado (*To Do* (Para fazer), *In Process* (Em Desenvolvimento), *To Verify* (Para verificar), *Done* (Realizado)) ficam sempre visíveis e disponíveis para todos os interessados no desenvolvimento do projeto. Geralmente, o painel é desenhado e colocado em uma parede, sendo que as atividades são descritas em *post-its*.

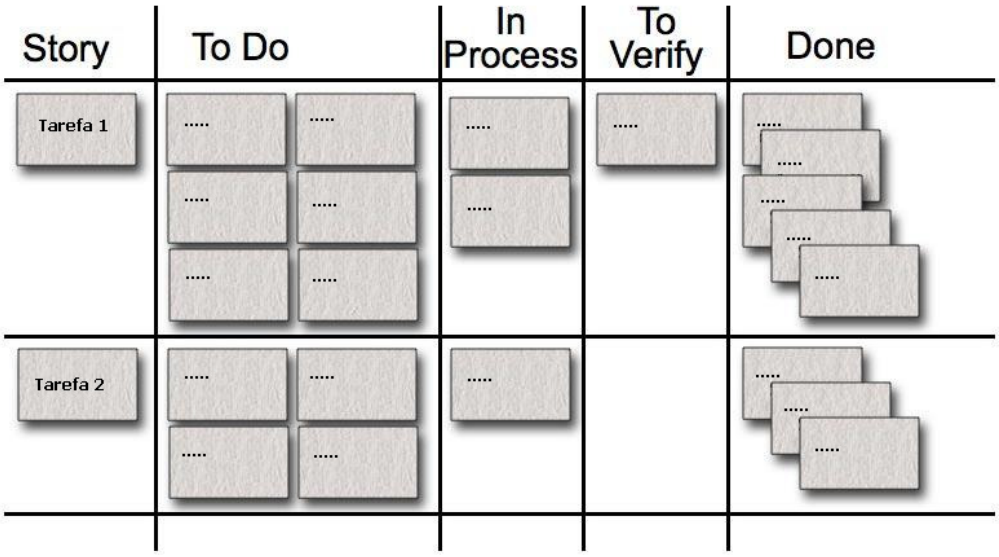

Figura 8 – *Taskboard*
Fonte: Mountain (2011)

Segundo Mountain (2011), no gráfico da FIG. 9 é exibida uma introdução do que é essencial para o desenvolvimento utilizando o método *Scrum*. À esquerda, é representado o *product backlog*, que é priorizado pelo *product owner* e contém todos os requisitos necessários para o desenvolvimento do ciclo atual. As semanas do *sprint* estão representadas no maior círculo.

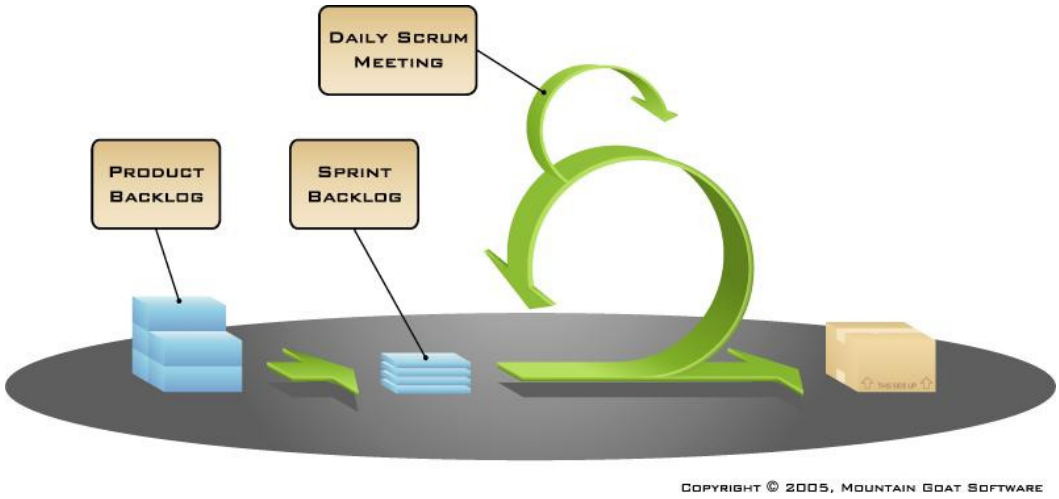

Figura 9 – Visão geral do *Scrum*
Fonte: Mountain (2011)

No início de cada *sprint*, a equipe seleciona uma quantidade de trabalho a partir do *product backlog* e se compromete a concluir o trabalho durante o *sprint*. No final de cada *sprint*, a equipe produz uma versão do sistema de software potencialmente utilizável. Todo dia, os membros da equipe se reúnem para discutir os progressos e eventuais entraves para conclusão do trabalho. Isso é conhecido como *daily scrum* e é exibido no menor círculo da FIG. 9.

2.2. Extreme Programming (XP)

O modelo de desenvolvimento *Extreme Programming* é um processo de desenvolvimento ágil. Segundo Beck (2001), os processos ágeis são aplicados com maior relevância em projetos curtos ou com equipes de trabalho co-localizadas. Eles apresentam uma visão parecida sobre as devidas práticas necessárias no desenvolvimento de um sistema e

a obtenção da qualidade, tais como a preocupação com os requisitos (funcionais e não-funcionais), o desenvolvimento iterativo e o envolvimento dos clientes no projeto.

De acordo com Astels (2002), as práticas do *Extreme Programming* foram criadas para funcionar em conjunto e disponibilizar maior valor do que cada uma poderia fornecer individualmente. Essa metodologia de desenvolvimento de sistemas de software também preocupa-se demasiadamente com relação a alteração constante dos requisitos do sistema. Sobre esse assunto, o processo enfatiza que o desenvolvedor deve procurar permitir que o projeto seja flexível, ao invés de lutar contra as alterações de requisitos durante o desenvolvimento.

Segundo Fowler (2001), esse processo é uma metodologia ágil centralizada em equipes pequenas e médias que irão desenvolver sistemas com requisitos vagos e em constante mudança. Nesse modelo, é adotada a estratégia de acompanhamento constante e realização de vários ajustes durante o desenvolvimento do sistema.

Segundo Beedle (2001), o *Extreme Programming* possui um conjunto de tarefas que devem ser seguidas pelas equipes. Umas das principais diferenças dessa metodologia em relação a outras são: *feedback* constante, incentivo a comunicação entre as pessoas e abordagem incremental.

Conforme relatado por Highsmith (2001), o primeiro projeto que realmente utilizou o *Extreme Programming* foi o C3, da montadora de carros *Chrysler*. Após alguns anos de tentativas utilizando metodologias tradicionais sem sucesso, a montadora optou por trocar o processo de desenvolvimento para *Extreme Programming* e o projeto acabou ficando pronto em pouco mais de um ano.

Alguns autores, como apontado por Cockburn (2001), afirmam que a maioria das principais regras do *Extreme Programming* podem causar polêmica a primeira vista e outras nem fazem sentido se forem aplicadas isoladamente. A sinergia de seu conjunto que sustenta o sucesso dessa metodologia, encabeçando uma verdadeira revolução na engenharia de software.

De acordo com Beck (2001), o *Extreme Programming* foca no desenvolvimento rápido do sistema de software e procura garantir a satisfação do contratante, além de ainda favorecer o cumprimento das estimativas. As práticas, valores e regras desse método proporcionam um ambiente agradável de desenvolvimento de sistemas. Os princípios chaves e as práticas que compõem essa metodologia, não possuem nada de novo, quando analisadas individualmente. Nesse método, essas típicas práticas das metodologias ágeis foram reunidas e alinhadas de tal forma que se criasse uma independência entre elas e, assim, surgiu uma

nova metodologia de desenvolvimento de software. Os quatro princípios chaves dessa metodologia, os papeis e as responsabilidades e as principais práticas estão listadas a seguir.

a) *Comunicação*: Muitos problemas que ocorrem no decorrer do desenvolvimento do sistema de software podem ser relacionados como falhas de comunicação entre a equipe de desenvolvimento e o contratante ou entre a própria equipe de desenvolvimento. Um indivíduo pode deixar de comunicar um fato importante à outra pessoa, um desenvolvedor pode deixar de levantar uma questão importante à equipe ou ao próprio cliente. O *Extreme Programming* mantém o fluxo de comunicação através da utilização de algumas práticas que não podem ser desenvolvidas sem comunicação. Alguns exemplos são: programação em pares (prática descrita a seguir), testes de unidade e estimativa de esforço de cada ação;
b) *Simplicidade*: Sempre deve-se selecionar a alternativa mais simples que tem uma probabilidade de funcionar. O *Extreme Programing* baseia-se no fato de que é menos custoso fazer algo mais simples e alterá-lo conforme as necessidades que surgirem do que tentar prever as possíveis alterações futuras, introduzindo uma complexidade que pode não ser necessária no futuro;
c) *Feedback*: Todos os problemas devem ser evidenciados o quanto antes para que possa ser corrigido o mais cedo possível. Para que possa ser incorporada de forma rápida ao sistema de software que está sendo construído, todas as necessidades são descobertas o quanto antes;
d) *Coragem*: Para apontar um problema no sistema de software é preciso coragem, para simplificar um código que já foi testado e aprovado, para solicitar ajuda quando for necessário, para comunicar ao cliente possíveis alterações nos prazos estimados para entrega de versões e também para realizar alterações no processo de desenvolvimento do sistema.

Os papéis e responsabilidades do *Extreme Programming* estão descritos a seguir:

a) *Gestor*: É o elemento responsável pela tomada de decisões. Para desempenhar o seu papel, o gestor comunica-se a equipe do projeto para identificar qualquer dificuldade, deficiência no processo ou algum outro tipo de impedimento, além de determinar também qual a situação atual do projeto;

b) *Consultor*: É um elemento externo à equipe e possui conhecimento técnico específico necessário para o projeto em questão. O consultor também ajuda a equipe a resolver problemas específicos;
c) *Desenvolvedor*: Codifica programas, organiza testes e mantém o código fonte o mais simples e coeso possível. Uma característica essencial que o desenvolvedor deve possuir é a habilidade de comunicação e coordenação com os outros elementos da equipe;
d) *Cliente*: É responsável por escrever as estórias e os testes dos requisitos funcionais, além de selecionar e validar os requisitos do sistema. É ele quem define a prioridade de implementação para cada requisito;
e) *Testador*: É o elemento responsável por auxiliar o cliente a escrever os testes dos requisitos funcionais. Ele também executa os testes dos requisitos periodicamente e comunica o resultado desses testes para o restante da equipe;
f) *Monitor*: Acompanha a conformidade das estimativas realizadas pela equipe de desenvolvimento do sistema, como estimativas de esforço, e informa o quanto precisas são tais estimativas, com o intuito de melhorar as futuras estimativas. O monitor também acompanha o progresso de cada iteração e é responsável por avaliar se o objetivo é viável de acordo com as limitações de recursos e tempo, além de verificar se alguma mudança pode ser necessária no processo;
g) *Treinador*: É o elemento responsável por garantir a integridade do processo como um todo. Geralmente, uma pessoa que detém uma longa experiência em *Extreme Programming* é designada para esse papel, devido ao fato que ele é o guia para todos os elementos do projeto executarem o processo de forma adequada.

As doze práticas do *Extreme Programming* estão listadas a seguir e na FIG. 10 é exibido uma ilustração do agrupamento delas:

a) *Padrão na codificação*: Os desenvolvedores devem escrever todo o código fonte alinhado as regras que enfatizam a comunicação durante a codificação. Esse padrão é definido antes de iniciar o projeto e deve ser seguido por toda a equipe de desenvolvimento;
b) *Semanas de quarenta horas*: O *Extreme Programming* possui uma regra, para cada elemento da equipe, que limita em quarenta horas o tempo total que deve-se trabalhar em uma semana;

c) *Cliente no mesmo local da equipe*: Um usuário que responda pelo cliente deve integrar-se a equipe de desenvolvedores do sistema, disponível preferencialmente em tempo integral para responder as possíveis dúvidas dos elementos do projeto;
d) *Integração contínua*: São geradas versões internas e o sistema é integrado quantas vezes ao dia for preciso, após uma estória ser finalizada;
e) *Propriedade coletiva*: Os desenvolvedores da equipe possuem a permissão de alterar parte do código fonte em qualquer lugar do sistema e a qualquer momento;
f) *Programação em pares*: Dois desenvolvedores codificam o sistema em um mesmo computador. Geralmente, um é responsável por codificar enquanto o outro fica responsável por procurar erros;
g) *Jogo de planejamento*: Os desenvolvedores do sistema e o cliente entram em comum acordo para definirem o esforço necessário para implementar as estórias e a duração das iterações. Sendo que o cliente decide o escopo que será executado na iteração;
h) *Incrementos curtos*: Um incremento funcional e simples qualquer é gerado rapidamente com duração máxima de dois meses. Isso é feito para ter um retorno por parte do cliente em tempo hábil e poder incorporar correções e mudanças do sistema que está sendo desenvolvido;
i) *Metáfora*: Nessa prática é elabora uma descrição que permite todos os envolvidos no projeto explicarem como o sistema funciona. A prática da metáfora é de grande ajuda para os envolvidos compreenderem os elementos básicos do sistema bem como os seus relacionamentos, gerando um vocabulário comum para os envolvidos. Ela também sugere uma estrutura de como o problema e a solução podem ser vislumbrados do sistema que está sendo criado;
j) *Projeto simples*: Essa prática solicita que o sistema deve ser projetado da forma mais simples possível, respeitando as necessidades atuais do projeto. As complexidades irrelevantes devem ser removidas quando descobertas;
k) *Testes*: Os testes funcionais são escritos pelo cliente, enquanto que os testes de unidade são escritos pela equipe do projeto e executados continuamente;

l) *Reestruturação*: Consiste na melhoria contínua do sistema. São realizadas atividades como remoção de códigos fontes duplicados, melhorias na comunicação, simplificações no processo, etc.

De acordo com Beck (2001), as práticas do *Extreme Programming* são agrupadas em quatro grupos de acordo com a sua finalidade (codificação, equipe, processo e produto). Na FIG. 10 são exibidas as práticas agrupadas, começando da elipse central e indo até as extremidades.

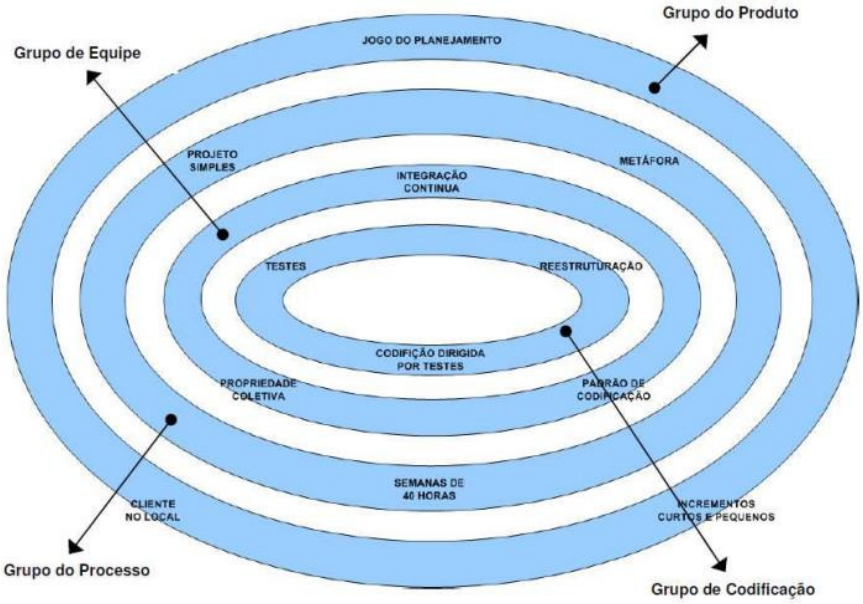

Figura 10 – Agrupamento das práticas do *Extreme Programming*
Fonte: Beck (2001)

2.3. Considerações finais

Segundo Boehm (2006), pode-se afirmar que a qualidade dos profissionais envolvidos no desenvolvimento do sistema atinge diretamente a qualidade do produto final e o desempenho do time do projeto. Apesar disso, ter excelentes profissionais na equipe, não é garantia de sucesso, pois essas pessoas dependem diretamente do processo de desenvolvimento. Um processo ineficaz pode fazer com que o talento dos melhores profissionais não seja aproveitado apropriadamente.

Ainda de acordo com o mesmo autor, a mescla da maneira correta de um processo adequado para o desenvolvimento de um sistema e com profissionais de talento, pode não ser suficiente. Também é necessário que a equipe possa interagir integralmente entre si. A

comunicação nesse caso é mais importante do que o talento técnico dos profissionais. Um profissional mediano com bom relacionamento é melhor e mais produtivo com o trabalho em equipe, do que um excelente profissional trabalhando sozinho.

De acordo com Beck (2001), é necessário levar em consideração que as ferramentas utilizadas no processo de desenvolvimento do sistema são importantes para o sucesso do projeto, entretanto elas não podem ganhar mais importância dos que as pessoas que utilizam. Ainda mais importante que o meio onde se pretende trabalhar, é o grau de interação e a qualidade da equipe.

Conforme relatado por Schwaber (2004), a documentação do projeto apesar de ser menos importante no conceito das metodologias ágeis, ainda não pode ser descartada, já que o código não é a melhor via de comunicação entre o racional e a estrutura do sistema. A documentação do sistema ainda se faz necessária para auxiliar nas tomadas de decisões do projeto. Ainda assim é necessário tomar cuidado com a documentação do projeto, porque ela em excesso é pior do que não ter nenhum tipo de documentação. De nada adianta uma série de documentos que demoram muito tempo para serem gerados, se eles não estão em sincronia com o que foi especificado e desenvolvido.

Segundo Beck (2001), documentos sem sincronia desvirtuam da realidade do sistema, atrapalhando na tomada de decisões. No manifesto ágil é sugerido que seja criada somente a documentação necessária e ainda que esteja sempre sincronizada com o projeto. A existência da documentação auxilia na integração da equipe, devido ao fato de que a passagem de conhecimento sobre o sistema é realizada no trabalho lado a lado, com a leitura do código fonte.

Ainda segundo o mesmo autor, o *Extreme Programming* define que para um sistema obter sucesso e aceitação por parte do contratante gerando um sistema de qualidade é necessário que exista um *feedback* do usuário periodicamente para ter a garantia de que o sistema está sendo desenvolvido de acordo com seus requisitos.

De acordo com Neto (2004), ao assumir que as mudanças de requisitos sempre irão acontecer em todos os projetos, podemos afirmar que o melhor projeto será o que se adaptar melhor a tais mudanças. A flexibilidade do projeto é um fator fundamental para se ter sucesso na entrega do produto final e ainda determina o quanto adaptável o sistema é.

Cockburn (2001) caracteriza as metodologias ágeis como uma abordagem de desenvolvimento que absorve os problemas de mudanças rápidas. Essas metodologias adaptam-se a novas mudanças decorrentes do desenvolvimento, em vez de analisar primeiro tudo o que pode ocorrer no andamento do projeto. Pode ser considerado como uma mescla

entre a ausência de processo e o processo exagerado. É dada preferência para uma documentação apropriada, que evite redundância e excessos, com o intuito de auxiliar de maneira efetiva o desenvolvimento do sistema.

As duas metodologias ágeis analisadas são bem parecidas em diversos aspectos, ambas apóiam as mudanças de requisitos durante o desenvolvimento do sistema sem perder a qualidade final do produto, além de enfatizar a importância da comunicação entre os elementos da equipe, bem como, com o cliente e o usuário final.

Pode-se dizer que as duas metodologias se completam. O *Extreme Programming* é uma metodologia centralizada em equipes pequenas e médias, que trabalha geralmente em ciclos com duração de uma semana e são mais indicados para projetos de curta duração, enquanto o *Scrum* trabalha com ciclos variando de duas a quatro semanas, e é adequado para projetos de grande porte e que demandam maior tempo para serem construídos.

3. ANÁLISE COMPARATIVA DA FASE DE IMPLEMENTAÇÃO DO SCRUM E XP

Segundo Highsmith (2004), as metodologias ágeis têm ganhado uma relevante aceitação por parte da indústria de software e pelos pesquisadores de Engenharia de Software. Mesmo ainda havendo falta de uma grande base de comparações históricas, os primeiros resultados têm sido substancialmente positivos. O *Extreme Programming* apesar de ser apontada como uma metodologia revolucionária, não apresenta muitas melhorias em relação as outras metodologias ágeis.

O *Extreme Programming* segundo Beck (2001), agrupa uma gama de práticas usadas desde o início da computação eletrônica. Como exemplo, pode ser citado a propriedade coletiva de código fonte e a programação em pares. Entretanto, as práticas dessa metodologia podem ser implantadas uma a uma para que sejam evitados desentendimentos, confusões e pressões desnecessárias, porque tem-se a tendência de voltar a utilizar as práticas anteriores, quando os desenvolvedores são colocados sobre pressão.

Segundo Schwaber (2004), o *Scrum* deve ser utilizado por equipes de até dez pessoas. Em projetos que precisam de equipes maiores, deve-se dividir os profissionais em times de até dez pessoas. Para melhorar a comunicação da equipe. O ciclo de desenvolvimento de cada iteração do *Scrum* não deve ultrapassar o período de duas a quatro semanas.

Este capítulo está dividido da seguinte forma. Na seção 3.1, são apresentadas as atividades comuns no processo de desenvolvimento de sistemas de software, como análise, desenho, implementação e teste, além de algumas ferramentas, técnicas e métodos utilizados nesse processo. Sendo que na seção 3.2, é abordado o processo adotado na fase de implementação das metodologias ágeis *Scrum* e *Extreme Programming*. Na seção 3.3, é apresentada uma análise comparativa das metodologias ágeis abordadas no capítulo anterior, em que é realizado um contraste dos principais pontos de vista, apontando algumas diferenças e semelhanças. Por fim, na seção 3.4, são apresentadas as considerações finais.

3.1. Atividades comuns no processo de desenvolvimento de software

Neto (2004) classifica modelo de processo como algo de natureza teórica, por exemplo, uma série de ações a serem tomadas no momento adequado, e processo, como um procedimento que determina as ações práticas a serem realizadas pela equipe como medidas de avaliação, prazos pré definidos, etc.

Conforme Sommerville (2007), um artefato é produzido dentro de um método e o resultado pode ser na forma de modelo, documento, código fonte, incremento, ou até um *build* ou *release*. Um *build* pode ser considerado um artefato parcial de um software e um *release*, um *build* que poderá ser entregue ao contratante para validação. O incremento é uma espécie de componente aproveitado na construção de um *build*.

Segundo Soares (2004), a adoção de um processo de desenvolvimento de software tem sido apontado como um fator primordial para o sucesso de empresas que desenvolvem sistemas de software. Sommerville (2007) define que um processo de desenvolvimento de software pode ser compreendido como um conjunto de atividades estruturadas e ligadas por padrões de relacionamento que são exigidas para desenvolver um produto final de qualidade. Logo um processo objetiva um software de alta qualidade com baixo custo. Um processo que não aumenta a produção ou não produz sistemas de software de boa qualidade, não podem ser considerados adequados.

De acordo com Pressman (2006), as atividades aplicadas em um processo de desenvolvimento de software se encontram agrupadas em fases, cada qual com suas responsabilidades, com estimativas e definições do que fazer para atingir um objetivo. Para definir um processo devem ser consideradas as atividades que serão realizadas, recursos que serão necessários, ferramentas, procedimentos e métodos adotados, artefatos e o modelo de ciclo de vida escolhido.

As principais razões para a definição de um processo padrão segundo Pressman (2006) são: Economia de esforço e tempo para definir os novos processos adequados a cada projeto, experiência adquirida nos projetos anteriores são incorporadas ao processo padrão, contribuindo assim para melhorias em todos os processos definidos, além da redução de problemas relacionados a revisões, treinamento e suporte as diversas ferramentas utilizadas pela equipe do projeto.

Segundo Sommerville (2007), apesar de existirem vários processos de desenvolvimento de software, as seguintes atividades são comuns à maioria:

a) *Especificação de software*: Define o que o software deve fazer, os requisitos e as restrições no desenvolvimento e na operação do sistema. Sendo que os principais processos de engenharia de requisitos são: Estudo de viabilidade, descoberta, análise, especificação e validação de requisitos;
b) *Projeto e implementação de software*: Corresponde a fase em que o software é desenvolvido de acordo com a sua especificação. No processo de projeto de software, é projetada uma estrutura que realize a especificação, já na implementação, é traduzida essa estrutura em um programa executável. As atividades de projeto e implementação estão relacionadas e podem ser interfoliadas;
c) *Validação do software*: É realizada a validação do software como forma de garantia de que tudo que estava especificado foi implementado. Os testes envolvem executar o software com relatórios de teste que são elaborados a partir da especificação de dados reais que serão processados pelo software. Alguns estágios de teste também são comuns como: Testes de unidade, de módulo, de subsistemas, software e de aceitação;
d) *Evolução do software*: Como o software é flexível, isto é, pode mudar na medida em que os requisitos sofrem alterações (seja por necessidade do cliente ou por outras circunstâncias), o desenvolvimento do software deve acompanhar esse processo. Embora exista uma separação entre desenvolvimento e manutenção, isso se torna cada vez mais irrelevante, pois um número cada vez menor de sistemas de software são completamente novos.

A análise comparativa deste estudo foca exclusivamente na fase de implementação, que é normalmente a fase que demanda mais tempo no processo e que é considerada de suma importância por autores como Pressman (2006) e Sommerville (2007), pois é nela que o software é construído de acordo com o que foi especificado e projetado. O software deve ser implementado da forma mais correta possível para não gastar demasiado tempo nas próximas fases do processo de desenvolvimento de software.

De acordo com o QAI (2011), 36% dos erros encontrados nos sistemas de software são provenientes da etapa de implementação e os outros 64% correspondem aos erros identificados nas etapas de análise e projeto. Quanto antes os erros forem descobertos, menos tempo gasta-se na correção.

3.2. Fase de implementação

Nessa seção, são apresentadas explicações em relação aos processos adotados pelas metodologias ágeis *Scrum* e *Extreme Programming* na etapa de implementação do código fonte, que resultará em um software executável pronto para ser submetido aos testes necessários. A seção está dividida em duas subseções, a 3.2.1 que detalha a metodologia *Scrum* e 3.2.2 que detalha o *Extreme Programming*.

3.2.1. Scrum

De acordo com Schwaber (2004), a equipe de desenvolvimento do *Scrum* trabalha de forma unida e com o foco em entregar um software de alta qualidade. Os desenvolvedores ficam comprometidos com um objetivo comum e possuem autonomia para definir a melhor estratégia para atingí-lo.

Diferentemente dos modelos tradicionais de desenvolvimento de software, em que o processo é extremamente definido e o processo é repetido várias vezes sem grandes diferenças. Ken Schwaber, que é um dos autores do *Scrum*, percebeu que o processo definido tornou-se inadequado devido à grande complexidade na produção dos sistemas, além de que poderia obter melhores resultados desenvolvendo o software como um processo empírico.

Segundo Beedle (2001), a equipe de desenvolvimento do *Scrum* atua de forma auto-organizada, sendo que a metodologia ainda permite a integração com outras metodologias ágeis, devido ao fato de o *Scrum* focar principalmente na gerência do projeto, sem ficar determinando como os desenvolvedores irão dividir as tarefas de programação de código fonte.

Conforme apontado por Schwaber (2004), os principais artefatos que a equipe de desenvolvimento manipula para cumprir as práticas do *Scrum* são os cartões que listam os requisitos e quando esses estão agrupados formam o *Product Backlog*. Os gráficos de acompanhamento como o *taskboard*, são largamente utilizados e devem ser constantemente atualizados pela equipe de desenvolvedores. Os gráficos são artefatos importantes para toda a equipe terem conhecimento sobre o andamento do projeto. Seguem algumas definições segundo o autor:

a) *Product Backlog*: Lista com todos os cartões contendo as funcionalidades que o software deve prover e estão faltando ser desenvolvidas. Esse artefato é priorizado pelo *product owner* e cada funcionalidade possui tempo estimado para entrega. O *product backlog* não está necessariamente completo desde o

início do projeto, ele pode crescer com o tempo e ainda sofrer alterações nas funcionalidades existentes;

b) *Selected Backlog*: Corresponde a um conjunto menor de funcionalidades escolhida pelo cliente no *product backlog* que será desenvolvida no próximo *sprint*, com a restrição de não poder ser alterado durante esse *sprint*;

c) *Sprint Backlog*: Lista refinada pela equipe de desenvolvimento a partir do *selected backlog*. As tarefas geralmente são quebradas em outras de menor tamanho, sendo que a equipe se compromete a desenvolver as funcionalidades listadas no *sprint* atual. No decorrer do *sprint*, o *scrum master* atualiza o *sprint backlog* para acompanhar as tarefas concluídas e o recálculo das estimativas do que está pendente;

d) *Impediment Backlog*: Controlada pelo *scrum master*, essa é a lista que contém os itens que estão impedindo o andamento do projeto. Podem estar associadas a riscos e não possuem grau de priorização.

Antes de se iniciar o *sprint* no *Scrum*, é realizada uma reunião mais conhecida como *sprint planning meeting* que tem como principal objetivo priorizar os itens que serão desenvolvidos pela equipe durante o próximo *sprint*. De acordo com Cockburn (2001), essa reunião é muito crítica para o sucesso do projeto, pois, se não for devidamente planejada, pode ocasionar grandes problemas como atrasos, estimativas subestimadas, etc.

De acordo com Schwaber (2004), a equipe determina como as tarefas devem ser executadas durante o *sprint*, sendo que eles não podem sofrer interferências externas e são blindados pelo *scrum master* de qualquer desvio que saia do objetivo previamente definido. Como a equipe de desenvolvimento é auto-organizada, eles devem informar as horas gastas em cada tarefa para calcular o tempo restante corretamente e o restante da equipe visualizar o progresso das atividades.

Segundo Highsmith (2004), uma das principais ferramentas da equipe para acompanhar o projeto no *Scrum* é o *release burndown chart* que mostra o andamento diário da equipe por horas de acordo com a soma das tarefas listadas no *sprint backlog*. A equipe também possui uma *taskboard* onde as funcionalidades são colocadas separadas por estados: *To Do*, *In Process*, *To Verify*, *Done*.

A equipe de desenvolvimento irá codificar o software, sendo que podem existir mais de uma equipe trabalhando em conjunto dependendo da complexidade do sistema que está sendo desenvolvido. Segundo Schwaber (2004), as equipes geralmente são formadas por

até oito membros e possuem autonomia para trabalharem de forma auto-organizada sem ultrapassar as diretrizes definidas para o projeto. Podem ser formados por programadores, designers, arquitetos, etc., de acordo com a sua necessidade.

3.2.2. Extreme Programming (XP)

De acordo com Beck (2001), as práticas mais executadas no *Extreme Programming* são revisão de código fonte, testes, participação ativa do cliente, integração ágil, simplicidade, além de outras que aumentam a qualidade do software. Com a proposta de intensificar essas práticas ao máximo, é sugerido que seja realizada a revisão constante do código fonte utilizando a prática de programação em pares, antecipando os testes e utilizando testes automatizados para detectar mais erros e o mais cedo possível, além de permitir a participação e a colaboração integral do cliente durante a fase de desenvolvimento do software.

Alguns dos artefatos utilizados na fase de desenvolvimento do *Extreme Programming* são os cartões de história e os radiadores de informação, que mesmo assim não são sempre utilizados. Seguem as definições segundo Beck (2001):

a) Cartões de história: São simples cartões de papel e são utilizados pelos clientes e pelos desenvolvedores para descreverem as funcionalidades requeridas para o software, e também como guia de trabalho para os desenvolvedores. Esses cartões não possuem toda a definição da funcionalidade especificada, a equipe no Extreme Programming obtém os maiores detalhes através da reunião presencial com o cliente, onde é aprendido o máximo possível, o cartão de história serve sucintamente como um lembrete do que deve ser implementado. No decorrer da iteração, os cartões de história são colocados na parede em forma de mural, para que o andamento da iteração possa ser acompanhado pelo restante da equipe.
b) Radiadores de informação: Cartazes que mostram dados e gráficos que destacam focos importantes do projeto são espalhados pelo ambiente de trabalho, tornando mais informativo para a equipe e permitindo também que análises rápidas sejam retiradas a qualquer hora. Os gráficos podem exibir informações como o número de histórias concluídas na iteração, evolução dos testes, opiniões da equipe com relação a qualidade de códigos fontes a partir dos testes realizados, etc. Esse artefato melhora significativamente a

> comunicação da equipe, pois várias informações ficam disponíveis aos membros da equipe. Com o aumento da equipe torna-se mais difícil acompanhar o que o outro está fazendo, é necessário então ficar mais atento a probabilidade de alguma pessoa sobrepor o trabalho de outra.

Segundo Cockburn (2001), no início de cada ciclo semanal, é realizada uma reunião para planejar o que será desenvolvido e estimar valores de custo e estimativas. De posse das histórias escritas pelo cliente, a equipe de desenvolvimento calcula as estimativas para as histórias e colocam nos respectivos cartões, sendo que o tempo gasto no desenvolvimento da última iteração é levado em consideração para informar o orçamento ao cliente, com o número de histórias que poderão ser desenvolvidas na próxima iteração.

De acordo com Highsmith (2004), a equipe que utiliza *Extreme Programming* trabalha com ciclos curtos, procurando assegurar que não seja feito muito trabalho e também concluído por partes. Os desenvolvedores apenas continuam adiante se o produto parcial estiver totalmente correto, os erros encontrados são corrigidos o mais rápido possível, antes de iniciar a próxima iteração. Essa forma de trabalho garante mais segurança que os eventuais erros sejam corrigidos com maior agilidade devido ao escopo de funcionalidades ser reduzido.

3.3. Análise comparativa

De acordo com Beck (2001), o manifesto ágil ressalta que os contratos, processos, planejamento e documentação possuem valor para o desenvolvimento de software, porém, são menos relevantes do que entender como lidar com as pessoas, ter o cliente ao lado da equipe colaborando para encontrar as melhores decisões no desenvolvimento do projeto, se adaptar facilmente a mudanças e entregar um software de qualidade.

Segundo Schwaber (2004), o *Scrum* propõe-se a obter resultados de forma prática em um período mais reduzido do que as empresas desenvolvedoras de software estavam habituadas, retirando o foco dos processos e preocupando-se primeiro com o produto. Dessa forma, o *Scrum* modificou as atividades executadas nos antigos processos e o modo como a equipe de desenvolvimento e até o cliente realizavam suas atividades. Isso se aplica também ao *Extreme Programming*.

Conforme apresentado por Cockburn (2001), existem algumas características que fazem parte de muitas metodologias ágeis, como o *Scrum* ou o *Extreme Programming*, e

geralmente são usadas durante a fase de implementação do software independente da metodologia ágil escolhida.

a) *Colaboração*: Em ambas as metodologias, o cliente está mais próximo da equipe de desenvolvimento do software e acompanha regularmente a evolução do projeto. Com esse contato constante, ganha-se agilidade na comunicação e na aceitação do cliente em relação ao produto que está desenvolvido. Devido a essa melhora na comunicação, torna-se mais apurada a visão dos envolvidos no acompanhamento do projeto, evitando assim, principalmente problemas durante a entrega do software. As prioridades, a definição do escopo e outras características de implementação são negociadas com maior clareza e praticidade;

b) *Estimativas*: As duas metodologias ágeis analisadas calculam as estimativas com transparência e comunicação, admitindo que existe uma incerteza no valor estimado para cada porção do software e deixa isso evidenciado para o contratante e a equipe terem conhecimento da dificuldade em implementar cada tarefa. Quanto maior for o prazo estimado para entrega, maior será o valor de incerteza associado, mas com o decorrer do tempo e o aumento do conhecimento sobre o incremento, podem ser refeitas as estimativas e o valor de incerteza diminuído;

c) *Desenvolvimento Iterativo*: Essas metodologias desenvolvem o software em ciclos, ou iterações, com o objetivo de produzir e também de integrar cada parte do software. Enquanto que no *Scrum* o ciclo pode durar de duas a quatro semanas, no *Extreme Programming* os ciclos geralmente são semanais, dependendo do projeto que está sendo desenvolvido. Assim, pode-se dizer que o software torna-se flexível em aceitar mudanças de requisitos ou de prioridades na fase de desenvolvimento. Ao fim de cada iteração, um software encontra-se disponível para testes e validação do cliente, redirecionando o que falta ser implementado;

d) *Desenvolvimento Incremental*: No decorrer das iterações nas duas metodologias ágeis analisadas, o software é preparado para receber incrementos de negócio, sejam novas funcionalidades ou alterações das que já estão sendo desenvolvidas. As funcionalidades podem ser desenvolvidas inteiramente e entregues no fim da interação, ou também podem ser criadas

algumas partes do requisito e entregues separadamente no fim das próximas iterações;

e) *Testes*: Diferentemente das metodologias tradicionais que reconhecem as etapas de implementação e testes como etapas totalmente distintas, no *Scrum* e no *Extreme Programming* essas etapas são executadas muitas vezes de forma paralela, com o desenvolvedor criando o código fonte e também testando, além de planejar os relatórios de testes. Segundo Boehm (2006), a execução dos testes desde o início da implementação, facilita na identificação dos erros e reduz o gasto com o desenvolvimento do software como um todo, quanto mais tarde for encontrado o erro, menor é o custo da manutenção, sendo que o valor pode ser cem vezes maior do que se tivesse sido corrigido durante as fases iniciais, devido a esse fato, tanto no *Scrum* quanto no *Extreme Programming*, é extremamente recomendável que os testes sejam executados o mais cedo possível.

3.4. Considerações finais

De acordo com estudos relatados por Highsmith (2004), existem estudos que relatam a combinação do *Scrum* com o *Extreme Programming* trabalhando em conjunto e com sucesso. Ainda de acordo com o autor, como o *Scrum* atua principalmente em gerência de projeto, isso favorece que o *Scrum* possa integrar alguns métodos ou práticas do *Extreme Programming* no seu processo de trabalho.

O desenvolvimento de um software não é uma tarefa trivial e é de suma importância escolher a metodologia mais adequada ao projeto que irá ser desenvolvido. Segundo dados do CHAOS (2009), apenas 34% dos sistemas de software são entregues com sucesso, sendo que 15% são até cancelados. Outro ponto importante a ser ressaltado é a detecção de erros, 56% deles poderiam ser identificados na fase de análise de requisitos, mas são encontrados apenas depois do software ser entregue, logo, a correção acaba gerando um custo maior para a empresa desenvolvedora.

Uma das principais características do *Scrum* e do *Extreme Programming* é que devido ao processo empírico adotado, ambas são adaptativas ao invés de serem preditivas. Logo, elas absorvem facilmente as mudanças originadas no desenvolvimento do projeto, ao invés de tentar analisar primeiro as consequências. Esse tipo de análise prévia utilizada em outras metodologias é mais complicado e acarreta alto custo.

O *Extreme Programming* é mais adequado para ser usado em sistemas no qual o cliente ainda não se decidiu totalmente sobre o que precisa, além de mudar de opinião com elevada frequência no decorrer da implementação do software. Com a prática de *feedback* aplicada nessa metodologia ágil por exemplo, fica mais fácil alterar com rapidez o que for necessário para contemplar as necessidades do cliente. Outra característica positiva são as entregas frequentes do software em partes utilizáveis, assim o cliente tem a possibilidade de visualizar o sistema funcionando mais cedo.

Apesar das qualidades citadas em ambas as metodologias ágeis estudadas, elas tendem a eliminar antigas práticas provenientes das metodologias tradicionais, como a análise da alteração por meio de diagramas. Além do processo de análise de requisitos ser bastante informal, principalmente no caso do *Extreme Programming*, que mesmo com a colaboração assídua do cliente pode não funcionar como o esperado e gerar alguma desconfiança por parte dos clientes, deixando-os inseguros com a falta de documentação.

CONSIDERAÇÕES FINAIS

Pode-se dizer que o simples ato de executar um software é uma tarefa precisa, porém a sua construção não é. Enquanto a ação de executar é feita normalmente por um sistema operacional, a tarefa de construir é realizada por indivíduos. Essa diferença não é levada em conta pelos processos tradicionais de desenvolvimento de software que focam seu trabalho nos processos como se esse fosse o responsável direto pela construção de um sistema, quando, na verdade, eles são produzidos pelas pessoas. Assim, um processo mais adequado deve ser baseado nas características humanas, focando nas pessoas que irão trabalhar no projeto.

As metodologias ágeis são mais focadas nas pessoas do que em documentações ou no processo em si. Ninguém consegue manter um ritmo uniforme de produção e nem executar tarefas na mesma precisão de um computador. No entanto, em compensação, conseguem visualizar mais informações do que se pode cadastrar em um banco de dados e adquirem habilidades que computadores ainda não são capazes.

Normalmente, o foco de um projeto é sempre na qualidade final do software que será entregue e apesar de existirem metodologias que podem ser mais adequadas a cada negócio, nenhuma delas traz a satisfação completa do contratante. O comparativo das metodologias apresentado no trabalho auxilia na escolha de qual a mais adequada a determinado negócio, com o objetivo de reduzir a quantidade de erros e aumentar a satisfação dos usuários. Além de apresentar alguns dados de mercado relacionados a desenvolvimento de projetos.

Acredita-se que as metodologias ágeis são mais adequadas para os sistemas de software que funcionam sobre a plataforma web, devido à característica de possuir um ambiente dinâmico e com frequentes mudanças. Para uma empresa aplicar o *Scrum* ou o *Extreme Programming* é necessário que ela saiba previamente que deverá ter confiança na competência e na motivação das pessoas que farão parte do projeto, pois eles vão estar diretamente envolvidos nas decisões.

Conforme o comparativo apresentado nesse trabalho, apesar de existir uma série de semelhanças entre os processos adotados nas metodologias *Scrum* e *Extreme Programming*, algumas características são comuns apenas ao *Scrum*, como as reuniões de planejamento e diárias, e as revisões do software. Também existem algumas práticas comuns

somente ao *Extreme Programming* como a programação em pares, estórias escritas pelos clientes e a elaboração dos testes antes do desenvolvimento da funcionalidade.

Em grande parte das organizações, é normal não utilizar nenhum processo para desenvolvimento de sistemas de software, devido à vários fatores como a falta de recursos financeiros ou até comodismo, que acaba gerando atrasos e quantidade excessiva de erros. Por outro lado, a empresa ao adotar o *Scrum* ou o *Extreme Programming* remete à equipe de desenvolvimento a sensação de que a empresa tem confiança nas suas habilidades e o sentimento de controle sobre o projeto, sentindo-se valorizados e motivados para crescerem profissionalmente.

REFERÊNCIAS BIBLIOGRÁFICAS

ASTELS D., MILLER G., NOVAK M., eXtreme Programming: Guia prático, Campus, Rio de Janeiro, 2002.

BECK, K. **Agile Manifesto**. 2001. Disponível em: <http://www.agilemanifesto.org>. Acesso em: 10 mar. 2011.

BECKER, B. E.; HUSELID, M. A; ULRICH, D. **Gestão estratégica de pessoas com "scorecard"**: Interligando pessoas, estratégia e performance. Rio de Janeiro: Campus, 2001.

BEEDLE, M. **SCRUM:** An extension pattern language for hyperprodutive software development. Disponível em: <http://www.controlchaos.com>. Acesso em: 10 mar. 2011.

BOEHM, B. **A View of 20th and 21st Century Software Engineering**. In: Proceedings of the 28th International Conference on Software Engineering (ICSE'06), Shanghai, China, 2006.

CAMPOS, A; FONSECA, I. **Por que Scrum?** Engenharia de Software Magazine. 4.ed. Rio de Janeiro 2008. Disponível em: <http://www.devmedia.com.br/articles/viewcomp.asp?comp=9868>. Acesso em: 10 mar. 2011.

CHAOS. **The Chaos Report**. Disponível em: <http://www.standishgroup.com>. Acesso em: 30 mar. 2011.

COCKBURN, A e HIGHSMITH, J. **Agile Software Development**: The Business of Innovation. IEEE Computer, 2001.

FOWLER, M. **The New Methodology**. 2001. Disponível em: <http://www.martinfowler.com/articles/newMethodology.html>. Acesso em: 10 mar. 2011.

HIGHSMITH, J. **Agile Project Management**: Creating Innovative Products. Addison Wesley, 2004.

MOUNTAIN. **Mountain Goat Software**. 2011. Disponível em: <http://www.mountaingoatsoftware.com/scrum>. Acesso em: 28 mar. 2011.

NETO, O. **Análise Comparativa das Metodologias de Desenvolvimento de Softwares Tradicionais e Ágeis**. 2004. Disponível em: <http://www.cci.unama.br/margalho/portaltcc/tcc2004/oscar.pdf>. Acesso em 10 mar. 2011

PRESSMAN, R. **Engenharia de software**. 6.ed. São Paulo: McGraw-Hill, 2006.

QAI. **Quality Assurance Institute**. Disponível em: <http://www.qaiglobalinstitute.com>. Acesso em: 30 mar. 2011.

SCHWABER, K. **Agile Project Managment With Scrum**. 1.ed. Microsoft Press, 2004.

SOARES, M. **Comparação entre Metodologias Ágeis e Tradicionais para o Desenvolvimento de Software.** 2004. Disponível em: <http://www.dcc.ufla.br/infocomp/artigos/v3.2/art02.pdf>. Acesso em: 10 mar. 2011.

SOMMERVILLE, I. **Engenharia de software**. 8.ed. Addison-Wesley, 2007.

SPÍNOLA, R.O. **Introdução à gestão do conhecimento**. Engenharia de Software Magazine. 6.ed. Rio de Janeiro, 2008.

TAKEUCHI, H.; NONAKA, I. **The New New Product Development Game**. Harvard Business Review, 1986.

VERSIONONE. **State of Agile Development**. Disponível em: <http://www.versionone.com/pdf/StateofAgileDevelopmentSurvey.pdf>. Acesso em: 10 mar. 2011.