ESCOLA SUPERIOR ABERTA DO BRASIL – ESAB
CURSO LATU SENSU EM ENGENHARIA DE SISTEMAS

JALERSON RAPOSO FERREIRA DE LIMA

METODOLOGIAS ÁGEIS

NATAL – RN
2010

JALERSON RAPOSO FERREIRA DE LIMA

METODOLOGIAS ÁGEIS

Monografia apresentada à ESAB – Escola Superior Aberta do Brasil, sob a orientação da profª. Maria Ionara Barbosa de Andrade Gonçalves.

NATAL – RN

2010

JALERSON RAPOSO FERREIRA DE LIMA

METODOLOGIAS ÁGEIS

Aprovada em ___ de _____ de _____

_____

_____

_____

NATAL – RN

2010

A minha querida noiva, Jeovana Araújo, pelo carinho, paciência e amor que ela dedica a mim diariamente.

AGRADECIMENTOS

Agradeço primeiramente a Deus, por ter me concedido tudo o que tenho até hoje.

A minha noiva, Jeovana Araújo, pela paciência e imenso carinho que me ajudaram a concluir esse trabalho.

A minha família por ter me dado tão boas condições de educação e de vida.

E finalmente, mas não menos importante, a equipe de profissionais que compõem a ESAB, pelos conhecimentos transmitidos e a boa vontade em ajudar seus alunos.

“A mente que se abre para uma nova ideia jamais voltará a seu tamanho original.”

(Albert Einstein)

RESUMO

As metodologias de desenvolvimento ágeis são relativamente novas, pois apesar de seus conceitos e ideais terem surgido na década de 1980, apenas em 1990 elas passaram a ser largamente difundidas através da publicação do Manifesto Ágil. Esse manifesto, elaborado por vários especialistas, teve como objetivo mudar os valores nos projetos de software. Essa mudança de valores teve como motivação os altos índices de projetos fracassados, entregues fora do prazo ou com o escopo reduzido. Esses índices foram apresentados pelos relatórios anuais do *Standish Group*, conhecidos como Relatório do Caos. Esse trabalho tem como objetivo apresentar um amplo estudo acerca das metodologias ágeis, de forma a demonstrar suas vantagens, desvantagens, compará-las com as metodologias prescritivas e explicar em quais situações elas podem ser aplicadas.

SUMÁRIO

1. INTRODUÇÃO

Palavras-chave: metodologias ágeis, scrum, xp.

Atualmente, os negócios entre empresas, instituições e países são realizados em escala global e estão sujeitos a rápidas mudanças, sejam elas econômicas, de mercado, de oportunidades, etc. Os softwares são, frequentemente, ferramentas utilizadas nessas negociações, e dessa forma estão sujeitos a mudanças decorrentes do contexto aos quais eles são aplicados (SOMMERVILLE, 2007).

Sendo assim, os sistemas precisam ser adaptativos para que possam atender a essas mudanças de contexto, e nesse sentido, os processos de desenvolvimento de software também precisam permitir essa flexibilidade.

Contudo, grande parte dos projetos de software não é concluída com sucesso. Em 1968, o termo “crise do software” foi criado numa conferência da OTAN sobre Engenharia de Software, e até hoje é usado para fazer referência à grande quantidade de projetos fracassados (TELES, 2005).

Desde 1994, uma empresa americana chamada *Standish Group* publica um relatório chamado Relatório do Caos (do inglês, *Chaos Report*). Trata-se de um amplo trabalho de levantamento envolvendo milhares de projetos de software, que aponta estatísticas de sucesso e fracasso nesses projetos (TELES, 2005).

A Figura 1 resume os resultados apresentados por esses relatórios entre 1994 e 2009.

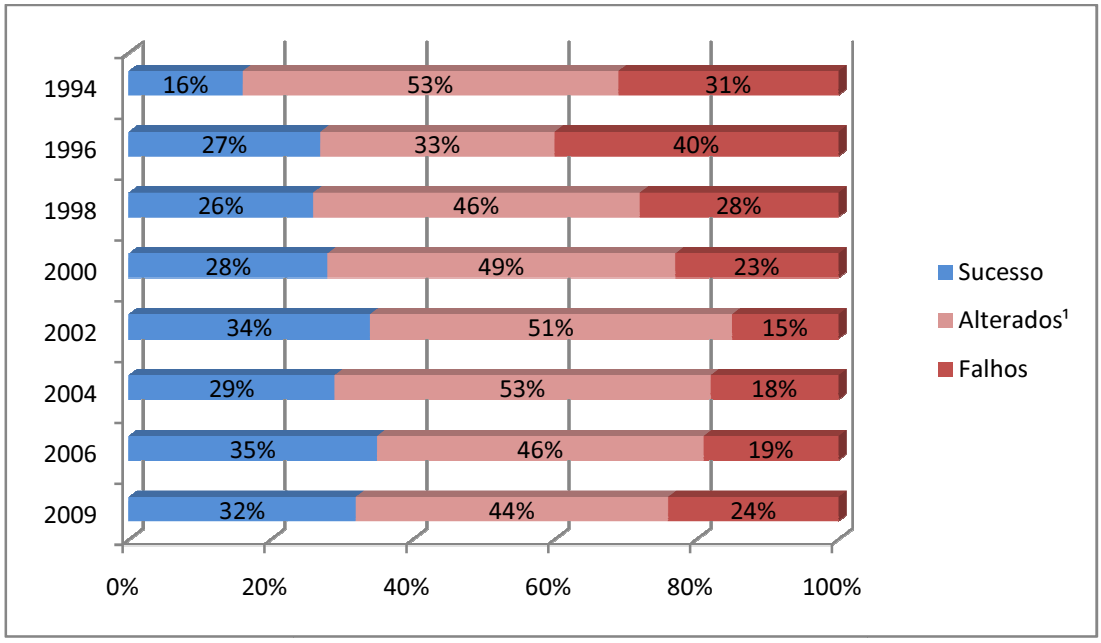

Figura 1 - Taxas de sucesso e de falhas dos projetos de software.
Fonte: Dominguez (2009)

Segundo os relatórios de 1994 a 2009, em média apenas 28% dos projetos de software foram concluídos com sucesso. Os outros 72% foram concluídos com atraso, fora do orçamento, com escopo reduzido ou não foram entregues.

Em 2002, durante uma conferência sobre *Extreme Programming* na Itália, Jim Johnson, o presidente da *Standish Group*, apresentou um estudo sobre o uso das funcionalidades por parte dos usuários dos sistemas (Figura 2).

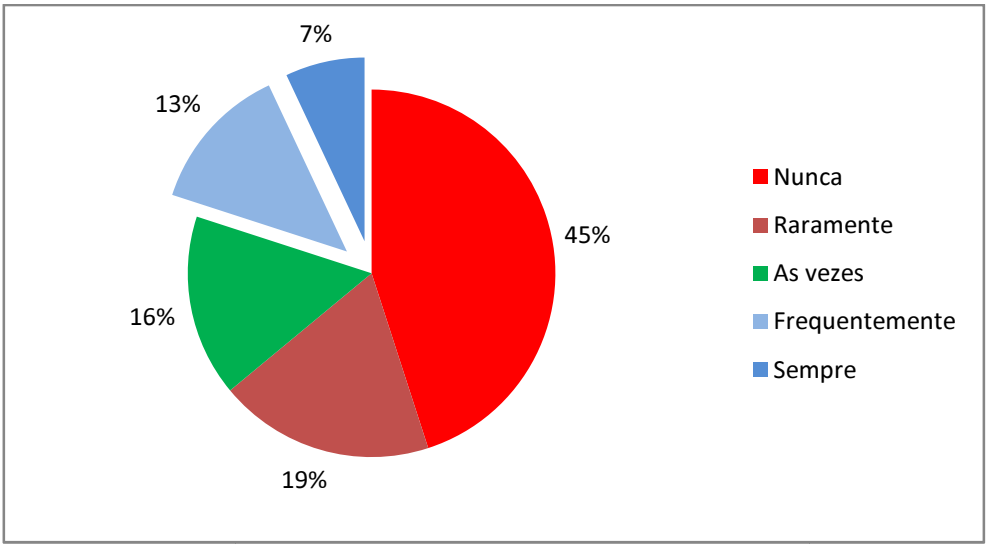

Figura 2 - Frequência de uso das funcionalidades.
Fonte: Teles (2005)

[1] Entregues atrasados, fora do orçamento e/ou com escopo reduzido.

O resultado dessa pesquisa aponta que apenas 20% das funcionalidades solicitadas pelos clientes são de fato utilizadas. Esses estudos demonstram claramente que algo precisa ser mudado nos projetos de software.

Em 2001, Kent Beck e 16 outros profissionais da área de software, conhecidos como Aliança Ágil, criaram o Manifesto para o Desenvolvimento Ágil de Software, conhecido também apenas por Manifesto Ágil.

Segundo Valente (2007), esse manifesto foi elaborado como alternativa às metodologias de desenvolvimento prescritivas, ou seja, metodologias pesadas e orientadas à documentação e planejamento como, por exemplo, o *Rational Unified Process* (RUP, 2001).

Contudo, seria o desenvolvimento ágil a solução definitiva para a produção de softwares de sucesso? Quais as vantagens e desvantagens dessas metodologias? Elas podem ser aplicadas a qualquer projeto?

1.1 MOTIVAÇÃO DO TRABALHO

O desenvolvimento ágil é um assunto relativamente novo. Em 2010 o Manifesto Ágil completa apenas nove anos. Nesse período, muitos profissionais da área de software têm apontado essas metodologias como sendo a solução para o grande número de projetos fracassados. A motivação desse trabalho é responder até que ponto essa afirmação é verdadeira.

1.2 OBJETIVOS

1.2.1 Geral

Apresentar e analisar as vantagens e desvantagens proporcionadas pela implantação de metodologias ágeis no desenvolvimento de software.

1.2.2 Específicos

1. Apresentar as metodologias convencionais;
1.1. Compreender seus problemas;
2. Estudar as metodologias ágeis;
2.1. Analisar o Manifesto Ágil;
2.2. Estudar *Extreme Programming* (XP);
2.3. Estudar o *Scrum*;
3. Demonstrar as vantagens e desvantagens dessas metodologias.

1.3 METODOLOGIA

Pesquisa bibliográfica em livros, revistas, artigos, trabalhos acadêmicos tais como monografias, dissertações e teses.

2. CAPÍTULO I – METODOLOGIAS CONVENCIONAIS

Segundo Soares (2010), um processo de software (ou metodologia de desenvolvimento) é um conjunto de atividades que guiam a criação de um produto de software. Essas atividades produzem resultados que são conhecidos por artefatos.

As metodologias convencionais são conhecidas por serem orientadas a planejamento e usarem muita documentação, e por esse motivo são consideradas pesadas. Contudo, muitos projetos de software sofrem diversas mudanças de requisitos durante seu desenvolvimento, e a readaptação do planejamento é necessária (SOARES, 2010).

Este capítulo tem como objetivo apresentar algumas metodologias convencionais utilizadas em projetos de software atualmente.

2.1 MODELO CASCATA

Teles (2005) explica que, desde o surgimento da Engenharia de Software como área de estudo, ela deu origem a vários processos de desenvolvimento. O primeiro e mais famoso deles é o processo linear, também conhecido como linear ou cascata (Figura 3). Esse modelo organiza o projeto em várias etapas que são executadas sequencialmente: requisitos, análise, projeto, implementação, testes e implantação. Ainda hoje esse modelo de desenvolvimento é amplamente utilizado.

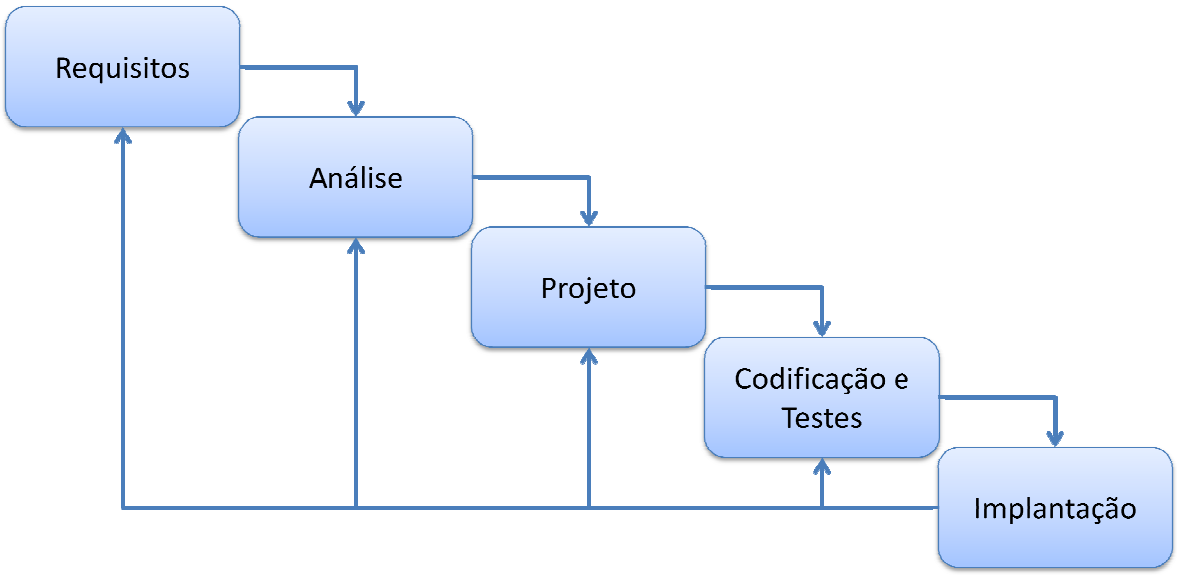

Figura 3 - Sequência de fases do Modelo Cascata.
Fonte: Sommerville (2007)

Teles (2005) ainda acrescenta que o modelo sequencial é uma tentativa de disciplinar o desenvolvimento e melhorar a previsibilidade. Contudo, o único resultado obtido é o formalismo do método, e não a melhoria de previsibilidade. Isso ocorre porque mudanças em projetos de software são frequentes, e o modelo linear não permite alterações do planejamento inicial durante o desenvolvimento do sistema.

Além disso, esse modelo não valoriza o *feedback* durante o desenvolvimento, aumentando as chances de que o projeto produza um sistema que não resolva as necessidades reais do cliente, mesmo que siga o planejamento inicial.

Ao longo do tempo, os engenheiros de software elaboraram alternativas ao modelo cascata. Algumas delas mantiveram características desse modelo, enquanto outras usavam abordagens diferentes.

2.2 PROCESSO UNIFICADO

O Processo Unificado foi criado por Ivar Jacobson, Grady Booch e James Rumbaugh em 1999, na publicação do livro *Unified Process*. Pressman (2006) explica que esse processo reúne as melhores características e práticas dos modelos convencionais. Portanto é o candidato ideal para exemplificar essas metodologias.

A seguir, são apresentadas as principais características do Processo Unificado.

2.2.1 Orientado por Casos de Uso

Bezerra (2002) explica que um caso de uso é o relato de uma funcionalidade do sistema, abstraindo os detalhes da estrutura e o comportamento interno dessa funcionalidade. Um modelo de casos de uso possui um conjunto de casos de uso que representam as funcionalidades do sistema e os agentes externos que interagem com ele.

Dessa forma, um processo orientado a casos de uso é guiado através do diagrama de casos de uso elaborado durante o projeto de software, e seu desenvolvimento será coordenado através das funcionalidades descritas nesse diagrama.

2.2.2 Centrado na Arquitetura

A arquitetura é a integração de todos os modelos que juntos representam o sistema completo. No início do projeto, um esboço da arquitetura é elaborado e à medida que o sistema é desenvolvido a arquitetura é refinada.

2.2.3 Iterativo e Incremental

Um processo iterativo é aquele que é dividido em ciclos de desenvolvimento, no qual cada ciclo é chamado de iteração. Cada iteração resulta em um produto parcial que nesse caso, também pode ser chamado de iteração. Portanto, um processo iterativo e incremental é dividido em várias iterações, e cada iteração resulta em um incremento no sistema que está sendo desenvolvido (JÚNIOR, 2001).

2.2.4 Fases e Atividades

Pressman (2006) explica que o Processo Unificado possui seis fases: concepção, elaboração, construção, transição e produção. Todas as fases, bem como as atividades desempenhadas em cada uma delas, estão ilustradas na Figura 4.

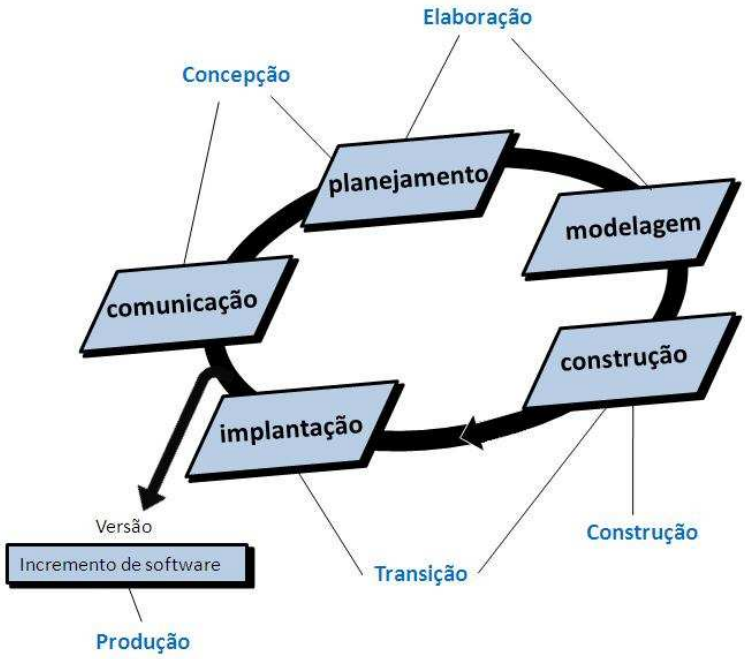

Figura 4 - Fases do Processo Unificado.
Fonte: Pressman (2006).

2.2.4.1 Concepção

Durante essa fase são executadas as atividades de comunicação e planejamento, uma vez que é necessário identificar os requisitos de negócio juntamente com o cliente e usuários finais do sistema.

Esses requisitos são especificados por meio de casos de uso preliminares, os quais descrevem as características e funcionalidades desejadas pelo cliente. Conforme explicado anteriormente, os casos de uso ajudam a identificar o escopo do projeto e fornecem base para o planejamento.

Os artefatos dessa fase são: documento de visão, modelo inicial de casos de uso, glossário inicial do projeto, caso de negócio inicial, avaliação inicial de risco, plano de projeto com fases e iterações, modelo de negócio (opcional) e um ou mais protótipos.

2.2.4.2 Elaboração

Na fase de elaboração são executadas as atividades de comunicação e modelagem, necessárias para refinamento e expansão dos casos de uso preliminares e do projeto arquitetural para incluir as cinco visões do software – modelo de casos de uso, modelo de análise, modelo de projeto, modelo de implementação e o modelo de implantação.

Nessa fase também pode ser criada uma “referência arquitetural executável”, que é uma primeira versão do sistema em execução, e tem como objetivo demonstrar a viabilidade da arquitetura projetada, contudo, sem fornecer todas as características e funcionalidades desejadas.

Além disso, os planos são cuidadosamente revisados a fim de garantir que o escopo, os riscos e os prazos estejam razoáveis, pois ainda é possível realizar modificações nessa fase do projeto.

Os artefatos dessa fase são: modelo de casos de uso, requisitos suplementares (não funcionais), modelo de análise, descrição da arquitetura do software, protótipo arquitetural executável (opcional), modelo de projeto preliminar, lista de riscos revisada, plano de projeto (incluindo planos de iteração, fluxos de trabalho adaptados, marcos e produtos técnicos de trabalho) e manual preliminar do usuário.

2.2.4.3 Construção

Nessa fase é executada a atividade de construção do sistema, que é a implementação de fato em uma linguagem de programação. Com base no projeto arquitetural, nessa fase são desenvolvidos os vários componentes que irão constituir o sistema completo. Há a possibilidade também de adquirir componentes terceirizados e integrá-los ao software.

Os casos de uso são utilizados pelos desenvolvedores para tornar cada um deles operacional para os usuários finais. Eles também se baseiam nos modelos de análise e projeto que foram elaborados nas fases anteriores.

À medida que os componentes são concluídos, testes unitários são criados e executados para validar seu funcionamento. Além disso, as atividades de integração são conduzidas para unir os componentes, e os casos de uso são utilizados para

criar uma sequência de testes de aceitação, que são executados antes do início da próxima fase.

Nessa fase são utilizados os artefatos: modelo de projeto, componentes de software, incremento integrado de software, plano e procedimentos de teste, casos de teste e documentação de apoio (manuais do usuário, manuais de instalação e descrição do incremento atual).

2.2.4.4 Transição

A fase de transição inclui as atividades de construção e implantação, pois o software é instalado para que os usuários finais possam realizar os testes beta, e quaisquer problemas são reportados através de relatórios de *feedback*, para que a equipe de desenvolvimento proceda com as correções necessárias.

Além disso, nessa fase são escritos os manuais de uso do software, guias de solução de problemas e procedimentos de instalação. Na conclusão dessa fase, o incremento do software torna-se uma versão utilizável.

Os artefatos dessa fase são: incremento de software entregue, relatório de testes beta e realimentação geral do usuário.

2.2.4.5 Produção

Na fase de produção a atividade de implementação conclui o ciclo. Durante essa fase, a utilização do software é monitorada. Dúvidas sobre seu funcionamento podem ser sanadas através de suporte técnico. Relatórios de defeitos e solicitações de mudanças são submetidos e avaliados.

2.3 RATIONAL UNIFIED PROCESS (RUP)

O *Rational Unified Process* (RUP) é um processo de desenvolvimento de software criado e mantido pela *Rational Software*, uma divisão da IBM (VALENTE, 2007). Esse processo tem algumas semelhanças com o Processo Unificado, como por exemplo, ser orientado a casos de uso, centrado na arquitetura e ser iterativo e incremental.

Barros (2007) aponta as principais características do RUP.

- **Gerenciamento de Requisitos:** controla os requisitos do projeto de forma a garantir que eles estejam sendo desenvolvidos corretament e que o escopo não esteja sendo reduzido ou ampliado;
- **Arquitetura Baseada em Componentes:** os componentes de software permitem que o sistema seja desenvolvido mais rapidamente, além de mantê-lo extensível, compreensível e reusável;
- **Modelagem Visual:** o RUP recomenda o uso da *Unified Modeling Language* (UML) para modelar o sistema durante sua análise e projeto;
- **Busca Contínua da Qualidade:** controla os aspectos de qualidade do sistema durante o seu desenvolvimento;
- **Controle de Mudanças:** essa prática gerencia as mudanças que ocorrem durante o desenvolvimento do projeto, principalmente alterações nos requisitos.

2.3.1 Fases

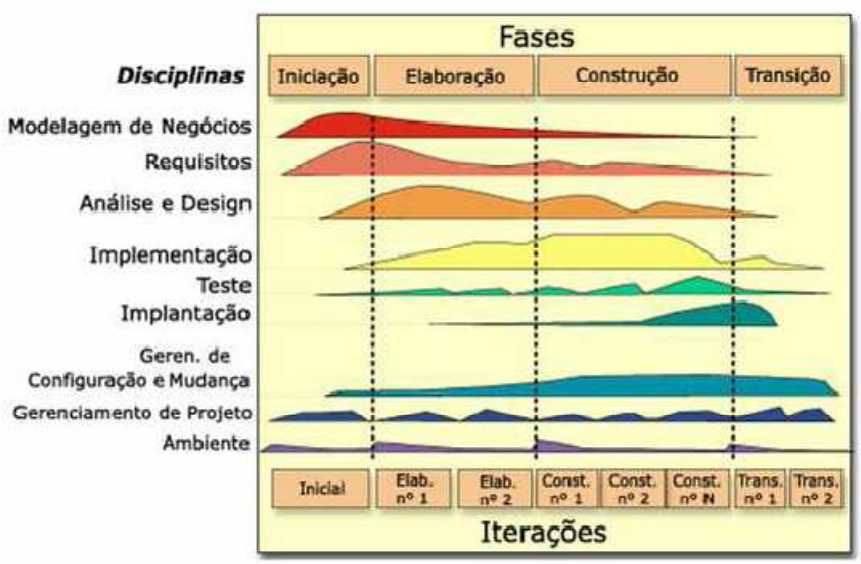

Figura 5 - Fases e disciplinas do RUP.
Fonte: Valente (2007)

Bork (2003) explica que o RUP divide o desenvolvimento em quatro fases consecutivas (Figura 5): iniciação, elaboração, construção e transição. Um marco é definido na conclusão de cada uma das fases.

2.3.1.1 Iniciação

Um dos artefatos mais importantes dessa fase é o documento de visão. Esse documento estabelece o domínio do projeto, além de conter os requisitos iniciais do projeto, características mais importantes, avaliação de riscos, estimativa de recursos e definição de prazos e dos marcos.

2.3.1.2 Elaboração

Na fase de elaboração a equipe deve estudar o domínio do problema e construir uma base arquitetural, que será usada para testar e validar a solução arquitetural proposta. Também deve-se criar o planejamento do projeto e tentar eliminar os maiores riscos.

2.3.1.3 Construção

Na fase de construção os desenvolvedores implementam o projeto em uma linguagem de programação, preferencialmente orientada a objetos.

2.3.1.4 Transição

O objetivo dessa fase é a instalação de uma versão experimental (beta) do software no ambiente dos usuários. Uma vez instalado, o software deve ser testado pelos usuários finais, e esses devem reportar correções e solicitações de mudanças para a equipe de desenvolvimento.

2.3.2 Disciplinas

O RUP possui nove disciplinas que são executadas durante as fases do projeto. Valente (2007) explica cada uma delas.

- **Modelagem de Negócio:** descrição dos problemas que motivaram o cliente a contratar o desenvolvimento de um sistema, e apresentar em que pontos ele poderia melhorar as atividades de negócio;
- **Requisitos:** especificar em detalhes os requisitos funcionais e não-funcionais do futuro sistema;
- **Análise e *Design*:** analisar os requisitos do sistema e elaborar modelos de análise e projeto que definirão como o sistema deve ser desenvolvido para atingir seus objetivos;
- **Implementação:** desenvolvimento das funcionalidades e integração de componentes terceirizados;
- **Testes:** nessa disciplina são realizados os testes de integração e de conformidade com os requisitos especificados. Os problemas identificados nessa fase devem ser resolvidos antes da implantação;
- **Implantação:** Desenvolvimento de versões estáveis do sistema e instalação no ambiente de produção. Nessa fase também são ministrados treinamentos para os usuários do sistema.

2.4 ANÁLISE DAS METODOLOGIAS CONVENCIONAIS

2.4.1 O Modelo Cascata

Pressman (2006) alega que, nas últimas duas décadas, esse modelo tem recebido críticas que colocam em questionamento sua eficácia.

- A grande maioria dos projetos de software não segue o fluxo sequencial que esse modelo sugere. Portanto, as mudanças podem causar confusão à medida que o projeto prossegue;
- É difícil para o cliente prever todos os requisitos no início do projeto. O modelo em cascata exige isso e não dá flexibilidade para acomodar mudanças de requisitos, que são comuns em projetos de software;
- O cliente não tem um *feedback* do software que está sendo produzido, pois uma versão executável do produto só estará disponível no final do projeto. Dessa forma, erros de identificação de requisitos podem causar resultados desastrosos no final do projeto.

Outro aspecto interessante desse modelo é que, devido a sua natureza linear, ele leva a estados de bloqueio da equipe, de forma que alguns membros precisam esperar que outros concluam suas tarefas para que possam dar início ou dar continuidade a seus trabalhos. Além disso, essa espera pode demorar mais do que o próprio trabalho produtivo.

Atualmente, os projetos de software precisam ser executados rapidamente e estão sujeitos a uma série de modificações durante seu desenvolvimento. O modelo em cascata é frequentemente inadequado para essas situações.

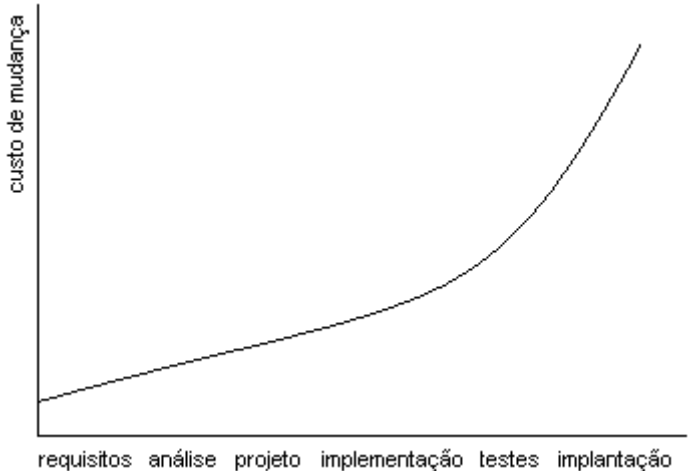

Figura 6 - Custo de mudanças no Modelo Cascata.
Fonte: Soares (2009)

Soares (2009) explica que o gráfico da Figura 6 apresenta o custo da mudança de requisitos em cada uma das fases do Modelo Cascata. O custo cresce de forma exponencial de acordo com o avanço nas fases do projeto, ou seja, estima-se que uma alteração que tenha o custo de “1x” na fase de requisitos, tenha um custo de “60x” a “100x” quando feita na fase de implantação.

2.4.2 Rational Unified Process (RUP)

Teles (2005) explica que o RUP possui um vasto conjunto de artefatos, além de recomendar práticas que devem ser seguidas para se atingir o resultado esperado. Contudo, determinar quais práticas e artefatos serão produzidos pelo projeto é uma tarefa difícil, que exige um profissional com conhecimento e experiência nessa metodologia.

Porém, existem poucas pessoas com esse perfil e a equipe de software acaba, muitas vezes, adotando práticas e produzindo artefatos desnecessários, o que pode levar a burocratização dos projetos.

Obviamente não é objetivo do RUP deixar os projetos “pesados”, contudo, processos que possuem um arcabouço extenso de artefatos a serem produzidos, frequentemente levam projetos a se burocratizarem demais devido a inexperiência dos membros da equipe.

Teles (2005) ainda alerta que, apesar do RUP estabelecer o desenvolvimento iterativo e incremental como forma de prover *feedback* e aprendizado contínuo,

muitas equipes adotam iterações muito longas, ou até mesmo executam o projeto em uma única iteração. Nesses casos o processo está seguindo o modelo cascata, e não o iterativo e incremental.

3. CAPÍTULO II – METODOLOGIAS ÁGEIS

As metodologias ágeis estão cada vez mais populares e as grandes empresas estão adotando essas práticas: Google, Yahoo, Symantec, Microsoft, e a lista continua aumentando (SHORE et. al., 2008).

Esse capítulo é destinado à apresentação do Manifesto Ágil, das duas mais famosas metodologias ágeis: *Extreme Programming* (XP) e *Scrum*, além de esboçar outros três métodos menos conhecidos: *Feature Driven Development* (FDD), *Dynamic Systems Development Method* (DSDM) e *Crystal*.

3.1 O MANIFESTO ÁGIL

Um manifesto é a expressão de uma nova ideia e normalmente está associado a um movimento político emergente, ou seja, um movimento que se contraponha às ideias tradicionais e sugira mudanças revolucionárias (PRESSMAN, 2006).

Pressman (2006) ainda acrescenta que, embora os princípios que sustentam o desenvolvimento ágil tenham sido elaborados há muitos anos, apenas na década de 1990 essas idéias foram condensadas em um movimento. Em essência, os métodos ágeis foram propostos para tentar vencer as deficiências da engenharia de software. Nesse manifesto era declarado (BECK et. al., 2009):

> Estamos descobrindo melhores modos de desenvolvimento de softwares, fazendo-o e ajudando outros a fazê-lo. Por meio desse trabalho passamos a valorizar:
>
> Indivíduos e iterações em vez de processos e ferramentas.
>
> Softwares funcionando em vez de documentação abrangente.
>
> Colaboração do cliente em vez de negociação de contratos.
>
> Resposta a modificações em vez de seguir um plano.
>
> Isto é, ainda que haja valor nos itens à direita, valorizamos mais os itens à esquerda.

Highsmith et. al. (2001) argumenta que softwares funcionando mostram aos desenvolvedores e aos clientes o que eles realmente possuem funcionando no momento, ao contrário de planos com promessas dizendo o que eles terão no futuro.

O software funcionando pode sofrer alterações, mas mesmo com modificações, ele ainda sim será um produto real.

As pessoas podem transferir idéias mais facilmente conversando pessoalmente do que lendo e escrevendo documentos. *Designers* sentados juntos produzem interfaces melhores do que se trabalhassem separados. Quando desenvolvedores conversam com os clientes e usuários, as dificuldades podem ser resolvidas em conjunto, ajustando prioridades e avaliando caminhos alternativos, de forma que isso não seria possível se eles trabalhassem separados (HIGHSMITH et. al., 2001).

Processos, ferramentas, documentação, contratos e planos tem seus valores e são úteis, contudo quando os prazos são curtos, e eles frequentemente são, é preciso tomar uma decisão sobre o que será entregue ao cliente. Por isso o manifesto recomenda que se entregue, por exemplo, software funcionando ao invés de documentação.

A interação entre indivíduos facilita a troca de informações e mudanças no processo quando elas precisam ser feitas. Tendo o software funcionando como meta, é possível medir a velocidade com que a equipe produz resultados reais e, dessa forma, dar um *feedback* mais rápido aos clientes. A frequente interação entre indivíduos compensa a redução da documentação.

Com a colaboração do cliente, todos se tornam membros da equipe de desenvolvimento – o cliente, os futuros usuários e os desenvolvedores. Unindo seus conhecimentos de negócio e técnico, eles são capazes de mudanças mais rápidas, proporcionando melhores resultados sem gastar tempo com muitos planejamentos.

Essa mudança de valores permite aos projetos melhor flexibilidade diante das mudanças. Soares (2009) explica que o gráfico da Figura 7 apresenta o custo de mudanças nas metodologias ágeis, que não cresce muito no decorrer do projeto.

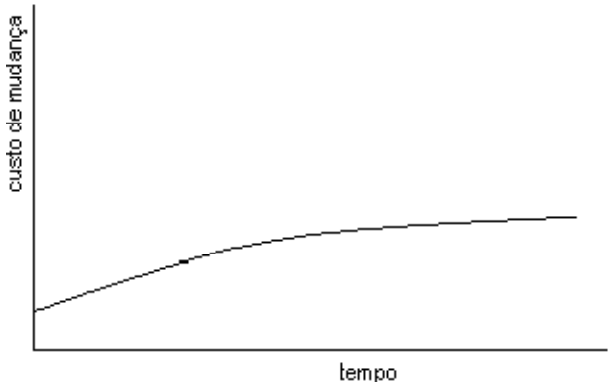

Figura 7 - Custo de mudanças em projetos ágeis.
Fonte: Soares (2009)

3.2 EXTREME PROGRAMMING (XP)

As ideias e métodos que estão associados a esse processo foram elaborados durante o final da década de 1980, contudo apenas em 1999, Kent Beck publicou um trabalho pioneiro sobre o assunto (PRESSMAN, 2006).

O XP recomenda o uso do paradigma orientado a objetos para o desenvolvimento de projetos de software. Ele também inclui valores e práticas que são seguidas em quatro atividades distintas (Figura 9): planejamento, projeto, codificação e teste (PRESSMAN, 2006).

3.2.1 Ciclo de Vida

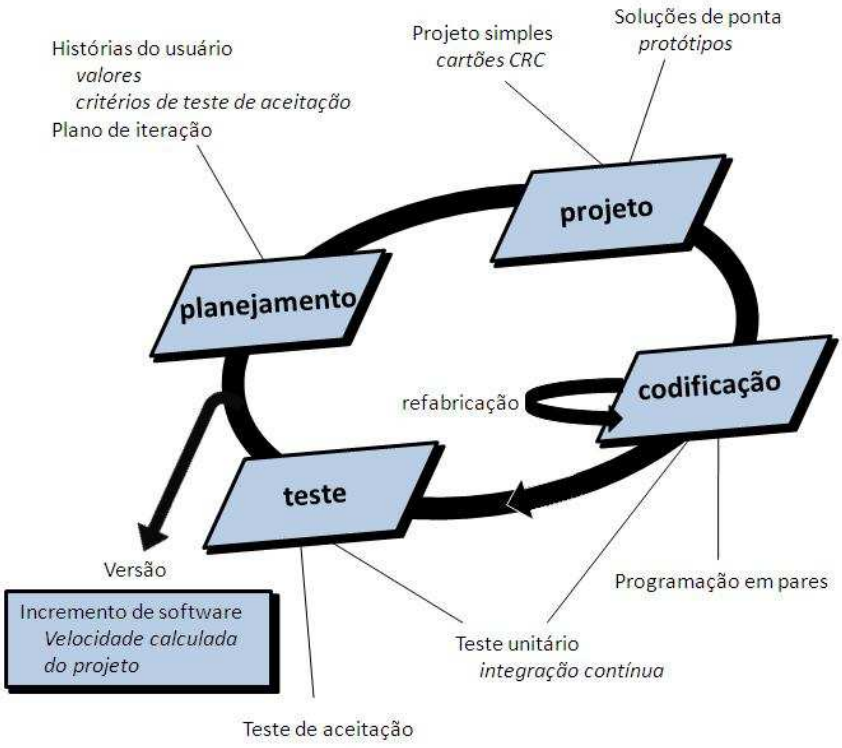

Figura 8 - Atividades e práticas do XP.
Fonte: Pressman (2006)

3.2.1.1 Planejamento

Essa atividade começa com a elaboração de um conjunto de histórias de usuário. Cada história descreve características e funcionalidades que o sistema deve ter. Cada história é escrita pelo cliente e lhe é atribuído um valor, que representa a prioridade (importância) daquela história para o cliente. Feito isto, a história é inserida em um cartão de indexação (PRESSMAN, 2006).

A equipe XP avalia as histórias e atribui um custo a cada uma delas. Esse custo é medido em semanas de desenvolvimento. Caso o custo seja superior a três semanas, é solicitado ao cliente que divida a história em histórias menores. Dessa forma, a atribuição do valor e do custo ocorre novamente. Vale salientar que novas histórias podem ser escritas a qualquer momento (PRESSMAN, 2006).

No início de cada iteração, os clientes e a equipe XP trabalham juntos para decidir quais histórias serão desenvolvidas. Feito isso, a equipe XP pode decidir por implementar todas as histórias imediatamente, priorizar histórias com valor mais alto ou priorizar histórias com risco maior (PRESSMAN, 2006).

Nessa atividade a equipe XP também calcula a velocidade de projeto da iteração anterior. A velocidade de projeto é a quantidade de histórias desenvolvidas durante a iteração. A velocidade de projeto é útil para:

- Ajudar a estimar as datas de entrega e o cronograma para as iterações posteriores;
- Descobrir se houve esforço excessivo no desenvolvimento das histórias. Se houver, a equipe XP precisará tomar mais cuidado ao definir, juntamente com o cliente, o conjunto de histórias de cada iteração.

Obviamente, se a equipe estiver na primeira iteração, o cálculo da velocidade de projeto não será realizado (PRESSMAN, 2006).

Vale salientar que à medida que o desenvolvimento prossegue, o cliente pode adicionar, alterar, dividir e eliminar histórias. Caso isto ocorra, a equipe XP precisa analisar as alterações e modificar os planos adequadamente (PRESSMAN, 2006).

3.2.1.2 Projeto

Um dos princípios mais importantes dessa atividade é o KIS (*keep it simple*), ou seja, mantenha a simplicidade. O XP encoraja a produção de softwares simples, ao invés de representações complexas. As histórias dos usuários devem ser implementadas exatamente como estão escritas, ou seja, nada mais e nada menos. Produção de funcionalidades ou melhorias extra são desencorajadas, mesmo que se tenha certeza que o cliente irá solicitar no futuro (PRESSMAN, 2006).

O XP também recomenda o uso de cartões CRC (*Class-Responsability-Colaborator*). Esses cartões identificam e organizam as classes orientadas a objetos que são importantes para o incremento da iteração (PRESSMAN, 2006).

Quando um problema de projeto de difícil solução é encontrado, o XP recomenda a criação de um protótipo operacional (denominado solução de ponta) como proposta de solução para o problema. O objetivo desse protótipo é avaliar se o problema é de fato resolvido antes que se inicie a implementação verdadeira. Ele também ajuda a validar as estimativas iniciais (PRESSMAN, 2006).

Outra prática recomendada pelo XP é a refabricação, que é um processo de modificar trechos de código do sistema sem que o comportamento externo desse

código seja alterado. A intenção é que essas modificações melhorem a estrutura interna do sistema, de forma a minimizar a ocorrência de defeitos.

3.2.1.3 Codificação

O XP encoraja que, antes que a equipe comece a implementar as histórias, ela desenvolva uma série de testes unitários, que serão utilizados após a implementação das histórias para avaliar o trabalho realizado. Após a elaboração dos testes, o desenvolvedor estará melhor preparado para codificar as histórias, pois ele saberá o que é necessário para que o código passe nos testes (PRESSMAN, 2006).

Durante a codificação, o desenvolvedor deve estar ciente de implementar apenas o necessário (KIS), e uma vez completado o código, pode executar os testes unitários (PRESSMAN, 2006).

Uma das práticas mais importantes do XP é a programação em pares. Nessa atividade recomenda-se fortemente que dois desenvolvedores trabalhem no mesmo computador. Dessa forma, enquanto um programador codifica, o outro acompanha seu trabalho para detectar erros ou melhorias que podem ser aplicadas (PRESSMAN, 2006).

À medida que as duplas concluem seu trabalho, os códigos produzidos são integrados ao trabalho de outras duplas. Essa tarefa é realizada diariamente por uma equipe de integração, que em alguns casos, pode ser a própria dupla que desenvolveu o código (PRESSMAN, 2006).

Essa estratégia de integração contínua é importante pois ajuda a evitar problemas de compatibilidade.

3.2.1.4 Teste

Nessa etapa são executados todos os testes unitários criados no início da codificação. Vale salientar que esses testes devem ser automatizados, de forma que eles possam ser executados fácil e repetidamente.

Os testes unitários, integração e validação do sistema são práticas que fornecem uma indicação contínua do progresso do projeto. Segundo Wells (apud PRESSMAN,

2006, p. 66) “resolver pequenos problemas a cada intervalo de umas poucas horas leva menos tempo do que resolver grandes problemas perto da data de entrega”.

Os testes de aceitação são especificados pelo cliente e avaliam as características e funcionalidades do sistema, sendo eles criados com base nas histórias do usuário.

3.2.2 Valores

Os valores são elementos, recursos, práticas e conceitos que o XP considera importante. São eles: *feedback*, comunicação, simplicidade, coragem e respeito. Esses valores devem ser seguidos pelos envolvidos no desenvolvimento durante todo o projeto.

3.2.2.1 Feedback

De todas as atividades desempenhadas durante o desenvolvimento de um sistema, uma das mais difíceis e importantes é compreender as necessidades do cliente, tendo em vista que essa compreensão direcionará todos os demais esforços.

Contudo, compreender requisitos costuma ser difícil, bem como também para os usuários transmiti-los. Segundo Brooks (1987), os clientes não têm como prover as funcionalidades que necessitarão, e Teles (2005) completa essa afirmação dizendo que “a compreensão das necessidades dos usuários é um processo de aprendizado contínuo no qual os desenvolvedores aprendem sobre os problemas de negócio e os usuários tomam conhecimento das dificuldades e limitações técnicas”.

“Um princípio psicológico bem conhecido indica que para maximizar a taxa de aprendizado, a pessoa precisa receber *feedback* sobre quão bem ou mal ela está indo” (WEINBERG apud TELES, 2005, p. 102).

Por esse motivo o XP é dividido em ciclos curtos de *feedback* que permitem aos usuários aprender sobre suas próprias necessidades, e dessa maneira, fazer com que os esforços sejam concentrados em tarefas realmente relevantes.

3.2.2.2 Comunicação

Num processo de desenvolvimento de software a comunicação é fundamental, e a forma como ela é realizada também é importante. Teles (2005) explica que quando uma pessoa está na presença de outra, essa comunicação é auxiliada por vários

elementos: expressões faciais, gestos, postura, tom de voz, etc. Contudo, a mesma conversa por telefone seria desprovida de todos os elementos visuais.

O XP recomenda que os usuários do sistema se tornem parte integrante da equipe e participem pessoalmente do processo de desenvolvimento. Isso permite que a equipe XP tenha acesso rápido e fácil a um especialista de negócio (TELES, 2005).

3.2.2.3 Simplicidade

O conceito de simplicidade é derivado do Sistema de Produção da Toyota, também conhecido como *Just In Time*. A Toyota acredita que o maior desperdício que existe é produzir algo que ninguém irá utilizar, e estudos apontam que cerca de 64% das funcionalidades produzidas não serão utilizadas (IMPROVEIT, 2010).

Por esse motivo, a equipe XP é encorajada a desenvolver apenas o necessário, e mesmo funcionalidades que com certeza serão solicitadas futuramente pelo cliente devem ser desconsideradas (IMPROVEIT, 2010).

3.2.2.4 Coragem

Problemas comuns a qualquer projeto de desenvolvimento, tais como prazos curtos, mudanças de requisitos, componentes terceirizados que não funcionam como deveriam, etc., não deixarão de existir em projetos XP (IMPROVEIT, 2010).

Contudo, é necessário que a equipe XP confie nas práticas e valores dessa metodologia para sanar esses problemas, ou seja, é preciso ter coragem para implantar as práticas que o XP recomenda, pois elas serão necessárias, uma vez que XP parte do princípio que esses problemas certamente ocorrerão (IMPROVEIT, 2010).

3.2.2.5 Respeito

O respeito é o valor mais importante de todos, pois sem ele nenhum outro valor poderá ser aplicado como deveria. Saber ouvir e compreender a opinião de outras pessoas é importante em projetos de software.

3.2.3 Práticas

As práticas são tarefas que devem ser executadas durante o processo de desenvolvimento. São elas (BONA, 2002):

- **Jogo de planejamento:** Determina com rapidez o escopo das próximas versões levando em consideração as prioridades de negócio;
- **Pequenas iterações:** O XP recomenda que as iterações sejam curtas para que a equipe possa ter *feedback* o quanto antes;
- **Metáforas:** São úteis para contribuir na comunicação entre os desenvolvedores e clientes;
- **Simplicidade:** É recomendado implementar os requisitos do cliente da forma mais simples possível, pois é preferível ter rápido *feedback* a um projeto muito elaborado e fora do prazo;
- **Testes:** Os testes precisam ser escritos antes da implementação da tarefa que será testada;
- **Refatoração:** Os desenvolvedores devem sempre aperfeiçoar o código-fonte, sem alterar o seu comportamento externo;
- **Programação em pares:** Toda implementação deve ser realizada por dois programadores na mesma estação de trabalho;
- **Propriedade coletiva:** Qualquer membro pode alterar qualquer parte do código do sistema;
- **Integração contínua:** Quando uma tarefa é concluída, o código produzido deve ser integrado ao sistema. Essa integração acontece várias vezes por dia.
- **Semana de 40 horas:** O XP recomenda que não se deve trabalhar além de 40 horas por semana, pois programadores cansados cometem mais erros;
- **Cliente dedicado:** As equipes XP precisam de um representante do cliente disponível durante todo tempo, responsável pelos requisitos, prioridades e por responder às dúvidas da equipe;
- **Padrões de código:** Todos os programadores devem seguir o mesmo padrão de codificação, a fim de assegurar a clareza e a comunicação através do código.

3.3 SCRUM

O *Scrum* é uma metodologia de desenvolvimento ágil que foi criado por Jeff Sutherland e por sua equipe no início da década de 1990. Mais tarde, Scwaber e Beedle fizeram contribuições com base em suas experiências. Assim como o XP e outras metodologias, o *Scrum* também segue o modelo iterativo e incremental (Pressman, 2006).

3.3.1 Papéis

Araujo et. al. (2006) demonstra que o *Scrum* define alguns papéis importantes para os membros da equipe. São eles:

- ***Product Owner*:** É um especialista de negócio que representa o cliente. Ele é responsável por representar os interesses do cliente durante o desenvolvimento do projeto;
- ***Scrum Team*:** É a equipe que, de uma forma ou de outra, irá produzir o software para o cliente;
- ***Scrum Master*:** É um especialista nessa metodologia e responsável por guiar o desenvolvimento seguindo as práticas e valores recomendadas pelo *Scrum*. Deve intermediar as negociações entre o *Product Owner* e a equipe, e também resolver impedimentos durante o desenvolvimento do projeto.

3.3.2 Conceitos e Práticas

Essa metodologia também introduz alguns conceitos que são importantes para a sua compreensão. São eles (IMPROVEIT, 2010):

- ***Sprint*:** É uma iteração do projeto. É recomendado que esse período não seja superior a 30 dias;
- ***Product Backlog*:** É uma lista que contém todas as funcionalidades desejadas para o sistema. Ela é definida pelo *Product Owner* e não precisa estar completa no início do projeto, pois durante o projeto, o *Product Backlog* crescerá e mudará na medida em que se compreendem as necessidades do cliente;
- ***Sprint Backlog*:** É um conjunto de funcionalidades que deverão ser implementadas durante um *Sprint* específico. As funcionalidades contidas no

Sprint Backlog são negociadas entre o *Scrum Team*, o *Product Owner* e o *Scrum Master*;

- ***Sprint Planning Meeting*:** É uma reunião realizada no início de cada *Sprint*, e nela estão presentes o *Scrum Master*, o *Product Owner* e o *Scrum Team*. Durante essa reunião, o *Product Owner* especifica as funcionalidades de maior prioridade, e juntamente com o *Scrum Team*, negociam quais delas deverão entrar no *Sprint Backlog*;
- ***Sprint Review Meeting*:** Enquanto o *Sprint Planning Meeting* é realizado no início de cada *Sprint*, o *Sprint Review Meeting* é feito no final. Nessa reunião, o *Scrum Team* e *Scrum Master* demonstram para o *Product Owner* o que foi alcançado durante o *Sprint*. Nessa reunião também é avaliado se os objetivos definidos na primeira reunião foram alcançados;
- ***Daily Scrum Meeting*:** No início de cada dia, o *Scrum Team* e o *Scrum Master* realizam uma reunião para avaliar o que foi feito no dia anterior, identificar a reportar impedimentos para o *Scrum Master*, e priorizar o trabalho no dia que se inicia. Essas reuniões devem ser realizadas na mesma hora e lugar, não devem demorar mais que trinta minutos e recomenda-se que elas sejam feitas com todos em pé;
- ***Sprint Retrospective*:** São reuniões também realizadas no fim de cada *Sprint*, e tem como objetivo identificar o que funcionou bem e o que precisa ser melhorado.

3.3.3 Ciclo de Vida

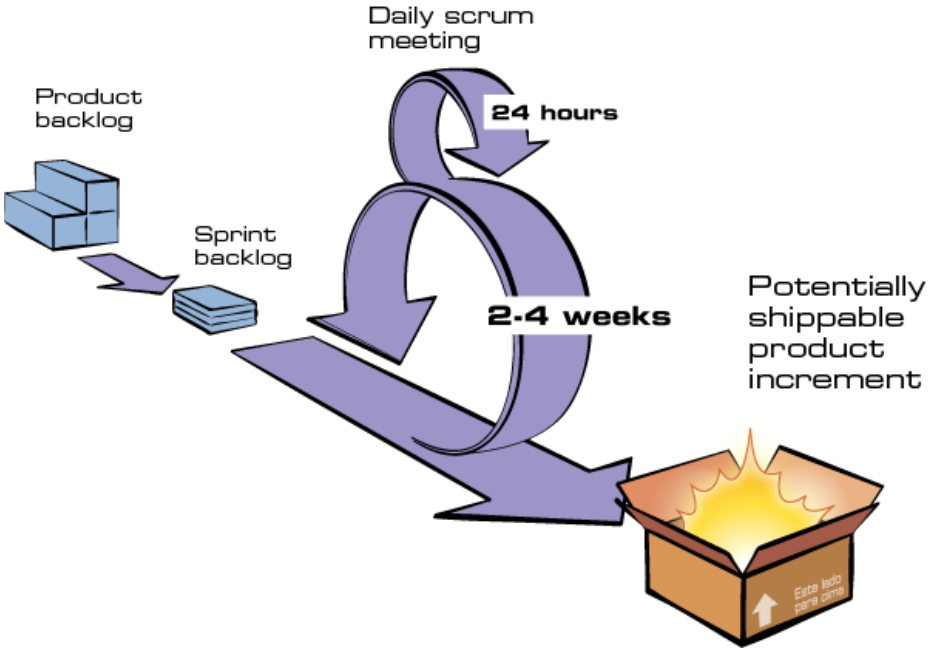

Figura 9 - Funcionamento do Scrum.
Fonte: Improveit (2010)

A Figura 9 demonstra visualmente como funciona o *Scrum*. O *Product Backlog* é o artefato do qual se originam todos os esforços do projeto, e dessa forma, é dele o ponto de partida do *Scrum*.

No início de cada *Sprint*, o *Scrum Planning Meeting* é realizado, e nessa reunião são definidas quais tarefas farão parte do *Sprint Backlog*. Cada *Sprint* deve durar entre duas a quatro semanas, e no início de cada dia de trabalho dentro de um *Sprint* é realizado o *Daily Scrum Meeting*.

Ao final de um *Sprint*, são realizadas a *Sprint Review Meeting* e a *Sprint Retrospective*. Um incremento do produto desenvolvido nessa iteração também é entregue para o cliente.

Depois disso um novo *Sprint* é iniciado, com um novo *Scrum Planning Meeting*, mais tarefas são retiradas do *Product Backlog* e colocadas no *Sprint Backlog*, dando início a uma nova iteração.

3.4 OUTRAS METODOLOGIAS ÁGEIS

3.4.1 Feature Driven Development (FDD)

Pressman (2006) explica que o Desenvolvimento Guiado por Características (do inglês, *Feature Driven Development*), foi criado por Peter Coad e seus colegas como um modelo ágil orientado a objetos. Stephen Palmer e John Felsing melhoraram essa metodologia, adaptando-a para ser aplicada em projetos de software médios e grandes.

Nesse modelo, uma característica "é uma função valorizada pelo cliente que pode ser implementada em duas semanas ou menos" (COAD apud PRESSMAN, 2006, p. 71). Essa definição traz os seguintes benefícios:

- Como as características são pequenas funcionalidades, os usuários podem descrevê-las facilmente;
- As características podem ser organizadas numa hierarquia;
- A cada duas semanas, são entregues características operacionais para o cliente;
- Como as características são pequenas, suas modelagens e códigos são mais fáceis de entender e inspecionar;
- Todo o planejamento do projeto é guiado pela hierarquia de características.

É recomendado que a definição de uma característica obedeça a seguinte sintaxe:

<ação> o **<resultado> <por | para | de | a>** um **<objeto>**

O <objeto> é uma pessoa, papel, lugar, momentos no tempo etc. Alguns exemplos de características são:

Adiciona *o* ***produto*** *a um* ***carrinho de compras****.*

Exibe *as* ***especificações técnicas*** *de um* ***projeto****.*

Armazena *as* ***informações de remessa*** *para um* ***cliente****.*

As características podem ser agrupadas em categorias, para dessa forma permitir sua organização numa hierarquia. Uma categoria de características pode ser definida com a seguinte sintaxe:

<ação (verbo no gerúndio)> um **<objeto>**

Por exemplo: “***Fazendo venda** de **produto***” é uma categoria que poderia incluir as características mencionadas anteriormente.

3.4.2 Dynamic Systems Development Method (DSDM)

Siqueira (2010) explica que o DSDM é um método ágil que se baseia nos métodos RAD (*Rapid Application Development*). O DSDM é organizado por um consórcio de empresas, que fornecem licenciamento e treinamento para aqueles que querem se especializar nessa metodologia.

A ênfase dessa metodologia se baseia nos protótipos que são elaborados durante o desenvolvimento, com a constante colaboração dos clientes. Assim como outras metodologias, o DSDM também possui alguns princípios e valores. São eles:

- Envolvimento contínuo dos usuários do sistema;
- O time deve ter o poder de tomar decisões;
- Foco na entrega frequente de versões;
- Desenvolvimento iterativo e incremental;
- Todas as mudanças durante o desenvolvimento são reversíveis;
- Requisitos são alinhas em um alto nível;
- Os testes são realizados durante todo o ciclo de vida;
- Uma abordagem colaborativa e cooperativa entre as partes envolvidas é essencial.

Algumas técnicas que devem ser adotadas durante todo o desenvolvimento também são importantes. São elas:

- **Time-boxes:** período fixo com datas de entrega para a execução do projeto. As funcionalidades que não forem implementadas nesse período devem ser desenvolvidas após esse período, mas antes da fase de pós-projeto;
- **MoSCoW:** é uma regra para priorização de requisitos. A idéia é dar prioridade aos requisitos principais, deixando os menos importantes para depois;
- **Modelagem:** não deve ser uma atividade burocrática, sendo usada apenas para melhor entendimento do problema e da solução;
- **Prototipação:** é uma forma eficiente de obter *feedback* do cliente. O protótipo deve evoluir juntamente com o projeto;
- **Teste:** devem ser executados durante todo o desenvolvimento do projeto.

3.4.3 Crystal

Crystal é o nome dado a um conjunto de quatro métodos ágeis: *clear* (limpo), *yellow* (amarelo), *orange* (laranja) e *red* (vermelho). Quanto mais escura a cor do método, mais burocrático ele é. Dessa forma, o *Crystal* permite que a equipe de desenvolvimento escolha o método mais apropriado para o projeto. A Figura 10 apresenta cada um dos métodos (SIQUEIRA, 2010).

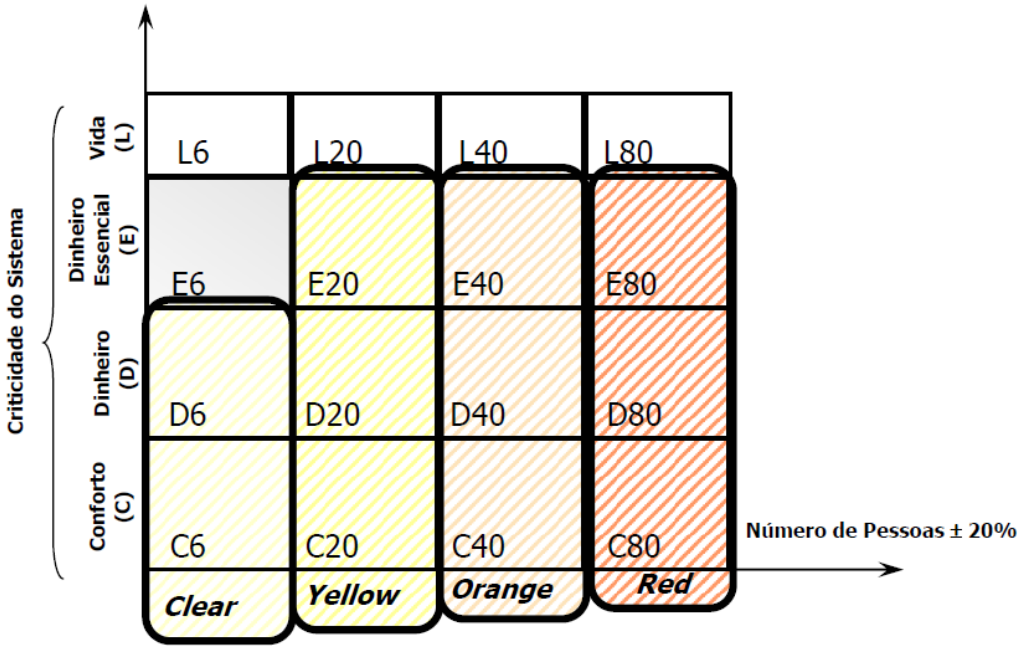

Figura 10 - Métodos da família Crystal.
Fonte: Siqueira (2010)

O peso do método é representado pela quantidade de artefatos produzidos e pela rigidez da gerência. Por exemplo, no *Crystal Clear* existem apenas seis papéis, enquanto no *Orange* existem mais de quatorze.

Apesar das diferenças de burocracia, todos os métodos sugerem valores e princípios comuns. Os valores recomendam que os métodos sejam voltados para as pessoas e para a comunicação, permitindo também que sejam utilizadas práticas e valores de outras metodologias ágeis.

O *Crystal* encoraja que o desenvolvimento seja incremental, com a duração de até quatro meses, e que sejam realizados *workshops* após a conclusão de cada incremento.

O *Crystal* também sugere alguns princípios. São eles:

- Comunicação face-a-face é a forma mais barata e fácil de transmitir informações;
- O excesso de peso do método é custoso;
- Times maiores precisam de métodos mais pesados;
- *Feedbacks* mais rápidos reduzem a necessidade de itens intermediários entregues;
- Disciplina, habilidade e entretenimento contra processo, formalidade e documentação.

3.5 ANÁLISE DAS METODOLOGIAS ÁGEIS

Soares (2010) argumenta que as metodologias ágeis ainda são novidade, contudo já apresentam bons resultados. Um estudo (CHARETTE, 2001) envolvendo 200 empresas comparou métodos tradicionais com metodologias ágeis. Ele aponta que projetos ágeis tiveram melhores resultados no cumprimento de prazos, custos e qualidade.

O mesmo estudo mostrou que, mesmo sob as recomendações de equipes pequenas, o número de membros nos projetos têm aumentado. Aproximadamente 15% dos projetos tinham de 21 a 50 pessoas, e 10% dos projetos tinham mais de 50 membros.

Porém, as metodologias ágeis também têm esbarrado em alguns obstáculos, principalmente em relação a alguns conceitos e práticas que podem parecer desinteressantes para os clientes, como por exemplo a programação em pares. Pode ser difícil para um cliente compreender que dois desenvolvedores trabalhando na mesma estação é melhor (a longo prazo) do que cada um em seu computador.

Outra dificuldade que projetos de software enfrentam é fazer com que o cliente assine um contrato de escopo variável ao invés de um contrato com escopo fixo. Muitos clientes preferem ter a segurança que um contrato de escopo fixo proporciona à flexibilidade de um com escopo variável.

Outra prática difícil de ser implantada é a presença constante de um especialista de negócio no ambiente de produção, pois manter um funcionário dedicado exclusivamente para o desenvolvimento é mais um custo envolvido no processo.

A especificação de requisitos nas metodologias ágeis normalmente é informal, como por exemplo, as histórias de usuários propostas pelo *Scrum* e XP. Essa informalidade pode não ser bem vista pelos clientes. Outra possibilidade é a sensação de amadorismo e incompetência que os clientes podem ter em virtude da refatoração de código (SOARES, 2009).

As próprias metodologias ágeis sugerem práticas que podem causar problemas, como por exemplo, a propriedade coletiva, que apesar de ser útil na correção de falhas ou melhorias, um trecho de código aparentemente incorreto pode ter um bom motivo para ter sido implementado daquela forma (VASCO ET. AL., 2006).

A pouca quantidade de artefatos nas metodologias ágeis pode ser preocupante, pois deixa o conhecimento vinculado aos desenvolvedores (CARLOS ET. AL., 2006).

Outros desafios para os métodos ágeis são resolver a carência de análises de risco sem tornar as metodologias mais pesadas, e adaptá-las para serem usadas em grandes projetos e com membros distribuídos geograficamente, tendo em vista que elas se baseiam em equipes pequenas e em comunicação pessoal.

4. CAPÍTULO III – COMPARAÇÃO ENTRE METODOLOGIAS ÁGEIS E CONVENCIONAIS

Este capítulo apresenta uma comparação entre as metodologias ágeis e as metodologias tradicionais sob vários aspectos: ciclo de vida, iterações, valores, comunicação, atividades, papéis, ferramentas e responsabilidade de código.

4.1 CICLO DE VIDA

Em geral, as metodologias tradicionais como RUP e UP tem o mesmo ciclo de vida que as metodologias ágeis: o modelo iterativo e incremental. Esse modelo é flexível em relação a mudanças de requisitos e funcionalidades devido ao *feedback* proporcionado pelas iterações.

Porém, é importante ressaltar que, apesar de usarem o modelo iterativo e incremental, as metodologias ágeis costumam ter períodos de iterações menores, medidos em semanas, enquanto os métodos tradicionais geralmente medem as iterações em meses. Isso permite que projetos ágeis tenham um *feedback* mais rápido de seus clientes.

4.2 VALORES

As metodologias ágeis valorizam aspectos mais humanos no processo de desenvolvimento, promovem a interação da equipe e a cooperação com o cliente através da comunicação face-a-face. O XP recomenda que o esforço da equipe de desenvolvimento não ultrapasse 40 horas semanais.

Essas práticas ajudam a resolver problemas mais rapidamente e a coibir o trabalho de horas extra, que podem sobrecarregar a equipe e deixá-la mais propensa a cometer erros e menos produtiva.

As metodologias pesadas valorizam mais aspectos relacionados ao processo, como os artefatos produzidos durante o desenvolvimento. Essa prática ajuda no controle do projeto, permitindo gerenciá-lo e planejá-lo de forma mais organizada.

4.3 COMUNICAÇÃO

A comunicação em processos pesados como o RUP é baseada em artefatos, enquanto nos projetos ágeis ela é feita pessoalmente. A comunicação pessoal é mais barata e rápida, contudo, deixar o conhecimento vinculado aos desenvolvedores pode ser um risco caso um membro da equipe, que normalmente já é pequena, deixe o projeto.

A comunicação baseada em artefatos é mais demorada e cara, porém o conhecimento fica contido em documentos acessíveis a qualquer membro da equipe e, dessa forma, se torna patrimônio da empresa.

4.4 ATIVIDADES E PAPÉIS

As metodologias tradicionais costumam ter mais atividades e papéis do que os métodos ágeis. Por exemplo, o RUP define um total de trinta papéis e nove atividades, enquanto o XP possui apenas quatro atividades e sete papéis.

Outra diferença importante é que o RUP define papéis específicos para determinadas atividades, enquanto o XP propõe uma divisão de papéis de “uso-geral”, que podem atuar em quaisquer atividades do projeto.

A grande variedade de papéis e atividades do RUP se mostra mais adequada para projetos grandes e com equipes numerosas, enquanto os poucos papéis e atividades definidas pelas metodologias ágeis estão voltadas para projetos menores e com poucos membros.

4.5 FERRAMENTAS DE GERENCIAMENTO

As metodologias ágeis não especificam o uso de softwares específicos para o gerenciamento dos projetos, mas vale ressaltar que, apesar de não serem obrigatórias, existem ferramentas pagas e gratuitas que auxiliam o gerenciamento de projetos ágeis, tais como: scrum’d (SCRUMD, 2010), XPlanner (XPLANNER, 2010), retrospectiva (RETROSPECTIVA, 2010), FireScrum (FIRESCRUM, 2010) e

diversos *plugins* (REDMINE, 2010) que adaptam o Redmine para ser usado com metodologias ágeis.

O RUP utiliza softwares como o *Rational Rose* para gerenciamento do projeto. Esse software é adquirido na IBM juntamente com a documentação relativa ao processo.

4.6 RESPONSABILIDADE DE CÓDIGO

Nas metodologias ágeis, normalmente o código-fonte é de responsabilidade coletiva, ou seja, qualquer um pode alterá-lo para correção de erros ou aperfeiçoamento. Nas metodologias pesadas como o RUP, o sistema é dividido em módulos, e para cada um deles existe um responsável.

A propriedade coletiva de código ajuda a evitar “ilhas de conhecimento”, ou seja, situações as quais apenas um desenvolvedor conhece como funciona determinada parte do código. Porém, um programador desavisado pode tentar aperfeiçoar um código aparentemente incorreto, sem saber que esse código tinha um bom motivo para ter sido escrito daquela forma.

5. CONSIDERAÇÕES FINAIS

5.1 CONCLUSÕES

As metodologias ágeis não trazem grandes novidades ao desenvolvimento de sistemas. Na verdade o Manifesto Ágil foi elaborado para mudar os valores nos projetos de software. Essa mudança de valores faz com que os métodos ágeis sejam mais adaptativos ao invés de serem preditivos.

É importante ressaltar que os métodos ágeis não vieram para substituir as metodologias pesadas, mas para ocupar um espaço no mercado de desenvolvimento de software. Mercado esse composto por projetos sujeitos a mudanças frequentes de requisitos, as quais serão melhor geridas pelas metodologias ágeis, que prevêem esses tipos de alterações.

As metodologias ágeis devem ser usadas em projetos que:

- Os clientes não sabem exatamente o que precisam;
- Mudanças de requisitos podem acontecer com frequência;
- O cliente e a equipe de desenvolvimento estão geograficamente na mesma região, permitindo a comunicação pessoal e contínua entre as partes;
- A equipe de software possui poucos membros.

As metodologias pesadas devem ser usadas em projetos que:

- Os requisitos de software são estáveis e previsíveis;
- Não haverão mudanças de requisitos com frequência;
- A exatidão na especificação de requisitos é crítica, como por exemplo em sistemas que lidam com vidas humanas;
- O cliente e/ou a equipe de software não precisam estar necessariamente na mesma região;
- A equipe de software possui muitos membros.

5.2 TRABALHOS FUTUROS

Como sugestão de trabalhos futuros, podem ser feitos estudos mais aprofundados sobre as taxas de sucesso que os projetos ágeis estão tendo atualmente, a fim de tentar avaliar se eles estão atingindo suas metas ou se eles ainda precisam de ajustes.

Outro trabalho interessante acerca das metodologias ágeis é propor uma forma para avaliar a produtividade dos desenvolvedores, e dessa forma, tentar premiar aqueles que atingirem determinadas metas de produção.

Outra sugestão para trabalho é tentar resolver um ou mais problemas das metodologias ágeis apontadas na seção “Análise das Metodologias Ágeis”, como por exemplo, elaborar uma análise de risco sem deixar o método mais pesado.

REFERÊNCIAS

ARAUJO, Alan R. S.; SILVA, Juliana M.; MITTELBACH, Artur F. **Scrum: Novas Regras do Jogo**. V Simpósio Brasileiro de Jogos de Computador e Entretenimento Digital, 2006.

BARROS, Raphael Lima Belém de**. Análise de Metodologias de Desenvolvimento de Software Aplicadas ao Desenvolvimento de Jogos Eletrônicos**. Universidade Federal de Pernambuco (UFPE), 2007.

BECK, Kent. **Agile Manifesto**. Disponível em: <http://www.agilemanifesto.org>. Acesso em: 23 jan. 2010.

BEZERRA, Eduardo. **Princípios de Análise e Projeto de Sistemas com UML**. 2ª edição. Rio de Janeiro: Elsevier, 2007.

BONA, Cristina. **Avaliação de Processos de Software: Um Estudo de Caso em XP e ICONIX**. Universidade Federal de Santa Catarina (UFSC), 2002.

BORK, Claudio Juliano. **Customização e Implantação de um Processo de Desenvolvimento de Software Baseado no Rational Unified Process (RUP)**. Universidade Regional de Blumenau, 2006.

BROOKS, Frederick P. **No silver bullet: essence and accidents of software engineering**. IEEE Computer, v. 20, n. 4, 1987. P. 10-19.

CHARETTE, R. Fair Fight? **Agile Versus Heavy Methodologies**. Cutter Consortium E-Project Management Advisory Service, 2001.

DOMINGUEZ, Jorge. The **Curious Case of the Chaos Report 2009**. Disponível em: <http://www.projectsmart.co.uk/the-curious-case-of-the-chaos-report-2009.html>. Acesso em: 01 fev. 2010.

FIRESCRUM. **FireScrum**. Disponível em: <http://www.firescrum.com>. Acesso em: 03 fev. 2010.

HEPTAGON. **FDD Estrutura**. Disponível em: <http://www.heptagon.com.br/fdd-estrutura>. Acesso em: 02 fev. 2010.

HIGHSMITH, Jim; COCKBURN, Alistair. **Agile Software Development: The Business of Innovation**. IEEE Computer Society, 2001.

IMPROVEIT. **Extreme Programming, XP: metodologia de desenvolvimento ágil**. Disponível em: <http://www.improveit.com.br/xp>. Acesso em: 31 jan. 2010.

JÚNIOR, Rodolfo Moacir Seabra. **Análise e Projeto Orientado a Objetos Usando UML e o Processo Unificado**. Universidade Federal do Pará (UFPA), 2001.

PRESSMAN, Roger S. **Engenharia de Software**. 6ª edição. McGraw Hill. 2006.

REDMINE. **Redmine Plugin List**. Disponível em: <http://www.redmine.org/wiki/redmine/Plugin_List>. Acesso em: 03 fev. 2010.

RETROSPECTIVA. **Retrospectiva**. Disponível em: <http://retrospectiva.org>. Acesso em: 03 fev. 2010.

RUP. **Rational Unified Process: Best Practices for Software Development Teams**. 2001. Disponível em: <http://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf>. Acesso em: 23 jan. 2010.

SCRUMD. **Scrum'd**. Disponível e: <http://scrumd.com>. Acesso em: 03 fev. 2010.

SHORE, J.; WARDEN, S. **A Arte do Desenvolvimento Ágil**. 1ª edição, Alta Books. 2008.

SIQUEIRA, Fábio Levy. **Métodos Ágeis**. Disponível em: <http://www.levysiqueira.com.br/artigos/metodos_ageis.pdf>. Acesso em: 03 fev. 2010.

SOARES, Michel dos Santos. **Comparação entre Metodologias Ágeis e Tradicionais para o Desenvolvimento de Software**. Disponível em: <http://wiki.dcc.ufba.br/pub/Aside/ProjetoBitecIsaac/Met._Ageis.pdf> Acesso em: 28 jan. 2010.

SOARES, Michel dos Santos. **Metodologias Ágeis Extreme Programming e Scrum para o Desenvolvimento de Software**. Universidade Presidente Antônio Carlos, 2009.

SOMMERVILE, Ian. **Engenharia de Software**. 8ª edição. São Paulo: Pearson Addison-Wesley, 2007.

TELES, Vinícius Manhães. **Um Estudo de Caso da Adoção das Práticas e Valores do Extreme Programming**. IM-NCE/UFRJ, 2005.

VALENTE, Carlos. **Engenharia de Software**. ESAB – Escola Superior Aberta do Brasil, 2007.

VASCO, Carlos G.; VITHOFT, Marcelo Henrique; ESTANTE, Paulo Roberto C. **Comparação entre Metodologias RUP e XP**. PUCPR, 2006.

XPLANNER. **XPlanner**. Disponível em: <http://www.xplanner.org>. Acesso em: 03 fev. 2010.