FACULDADE DE TECNOLOGIA DO ESTADO DE SÃO PAULO
CURSO DE TECNOLOGIA EM PROCESSAMENTO DE DADOS

KLEBER HIROKI UTIDA

METODOLOGIAS TRADICIONAIS E METODOLOGIAS ÁGEIS:

ANALISE COMPARATIVA ENTRE RATIONAL UNIFIED PROCESS E EXTREME PROGRAMMING

São Paulo – SP

2012

FACULDADE DE TECNOLOGIA DO ESTADO DE SÃO PAULO
CURSO DE TECNOLOGIA EM PROCESSAMENTO DE DADOS

KLEBER HIROKI UTIDA

METODOLOGIAS TRADICIONAIS E METODOLOGIAS ÁGEIS:

ANALISE COMPARATIVA ENTRE RATIONAL UNIFIED PROCESS E EXTREME PROGRAMMING

Monografia submetida como exigência parcial para a obtenção do Grau de Tecnólogo em Processamento de Dados

Orientadora:
SANDRA HARUMI TANAKA

São Paulo – SP
2012

FOLHA DE APROVAÇÃO

KLEBER HIROKI UTIDA

METODOLOGIAS TRADICIONAIS E METODOLOGIAS ÁGEIS:
ANALISE COMPARATIVA ENTRE RATIONAL UNIFIED PROCESS E EXTREME PROGRAMMING

Monografia submetida como exigência parcial para a obtenção do Grau de Tecnólogo em Processamento de Dados

Banca examinadora

_____

Sandra Harumi Tanaka

_____

Professor 1 (Avaliador)

_____

Professor 2 (Avaliador)

São Paulo, ____ de _____ de 2012

DEDICATÓRIA

Aos meus pais, familiares, amigos e professores por acreditarem no meu crescimento.

AGRADECIMENTOS

Agradeço a todos os meus amigos e professores por compartilharem o conhecimento e a experiência.

E agradeço principalmente à professora, orientadora e mestre, Sandra Harumi Tanaka, pelas boas conversas ao final das aulas e principalmente pela paciência.

“Praticamente todos os países, hoje em dia, dependem de complexos sistemas com base em computadores. Cada vez mais os produtos incorporam, de algum modo, computadores e software de controle. Nesses sistemas, o software representa uma grande e crescente proporção do custo total do sistema. Por isso, produzir software de um modo que apresente uma boa relação custo-benefício é essencial para o funcionamento das economias nacionais e internacionais.”

Ian Sommerville.

RESUMO

As metodologias tradicionais e as metodologias ágeis como principais áreas do desenvolvimento de *software* têm sido, atualmente, utilizadas mais frequentemente devido à problemas como estimativas de custos e prazos errôneos.

As metodologias tradicionais, conhecidas também como pesadas, que tem como característica uma grande quantidade de documentação gerada no processo de desenvolvimento de *software* que, muitas vezes, é apontada como a causa do atraso do projeto. E as metodologias ágeis que focam no código e evitam, se possível, a documentação.

A proposta deste trabalho é realizar um estudo das metodologias de desenvolvimento de *software* utilizando o *Extreme Programming* e o *Rational Unified Process*, principais expoentes das metodologias ágeis e tradicionais respectivamente, e comparar as características que estes possuem em comum.

E explicar as vantagens e desvantagens destas metodologias, levando em consideração fatores externos que podem influenciar a implantação e o desenvolvimento do projeto.

Palavras-chave: Metodologias Tradicionais, Metodologias Ágeis, desenvolvimento de *software*, Rational Unified Process, Extreme Programming.

ABSTRACT

The traditional and agile methodologies as main areas of the software's development has been, currently, used more frequently due the problems like cost's estimates and erroneous deadlines.

The traditional methodologies, known too as heavy, it has like characteristic a big quantity of documentation generated at development process software that, many times, is pointed like the cause of the project delay. The agile methodologies focus on code and avoid, if possible, the documentation.

The proposal this work is realize a study of the software development methodologies using the Extreme Programming and the Rational Unified Process, main exponents of the agile and traditional methodologies respectively and compare the characteristics it these has in common.

And explain the advantages and disadvantages of these methodologies, taking in consideration external factors that can influence the implantation and the project development.

Keywords: Traditional Methodologies, Agile Methodologies, software development, Rational Unified Process, Extreme Programming

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

COCOMO: Constructive Cost Model

MER: Modelo de Entidade e Relacionamento

RAD: Rapid Application Development

RUP: Rational Unified Process

SLOC: Source Line of Code

UML: Unified Modeling Language

USDM: Unified Solution Delivery Methodology

XP: Extreme Programming

1. INTRODUÇÃO

> À medida que a importância do *software* cresceu, a comunidade de *software* tem continuamente tentado desenvolver tecnologias quer tornem mais fácil, mais rápido e menos dispendioso construir e manter programas de computadores de alta qualidade. (PRESSMAN, 2006, p.2)

A Comunidade de desenvolvimento de *Software* sempre enfrentou dificuldades na criação e desenvolvimento de *software*. Na década de 70 ocorreu a crise do *Software*, causando uma mudança no método de criação e desenvolvimento de *software*.

Após a Crise, empresas desenvolvedores de *software* começaram a utilizar métodos para criação de *softwares*, gerando documentação para acompanhar o produto do *Software*, facilitando o entendimento do produto para o cliente e para empresa.

Essa documentação era gerada a partir da analise do projeto seguindo um método de desenvolvimento. Surgiram diferentes métodos, todos esses métodos dividiam o processo em etapas, mantendo o foco na qualidade do produto final

O processo de criação de *software* pode ser bastante mutável, pois o surgimento de novos requisitos por parte do cliente é comum, assim é necessário a constante modificação da documentação e do *Software*.

A modificação constante do *software* e de sua documentação demanda tempo. Para sanar esse problema surgiram os Métodos Ágeis, aqueles com foco no código e otimizados para alterações de requisitos, como a *Extreme Programming* - XP, esses métodos também prezam pela qualidade do *Software*, mas a sua filosofia de desenvolvimento é diferente, dando ênfase principalmente no código, sendo que as alterações necessárias não devem acarretar em tanto tempo gasto.

O surgimento recente de novas Metodologias de Desenvolvimento de *Software* fez com que ocorresse a divisão das Metodologias em dois principais grupos, as Metodologias Tradicionais, baseadas no Projeto, criando documentos para guiar o processo de desenvolvimento e as Metodologias Ágeis, baseadas no código, utilizando menos documentação e adotando processos mais simplificados.

Embora as metodologias ágeis tenham sido apontadas como alternativa às abordagens tradicionais para o desenvolvimento de *software*, as metodologias tradicionais, conhecidas como rigorosas, pesadas ou orientadas a planejamento, são

as mais utilizadas em situações onde os requisitos do sistema são estáveis e os requisitos futuros são previsíveis.

A abordagem ágil tem mostrado bons resultados com seu desenvolvimento baseado em código e sua facilidade em adaptar-se às mudanças de requisitos. Isto tem despertado um grande interesse entre as comunidades de desenvolvimento de *software* e muitos desenvolvedores tem aderido a este novo paradigma, mesmo sem ter conhecimento de suas limitações.

Neste trabalho será analisado as metodologias ágeis e metodologias tradicionais, estudando o *Rational Unified Process* – RUP – para as tradicionais e o *Extreme Programming* – XP – para as ágeis.

1.1. OBJETIVOS E METODOLOGIA DO TRABALHO

O objetivo principal deste trabalho é estudar as metodologias de desenvolvimento, em especial o RUP e o XP, e a partir deste estudo observar qual metodologia é mais viável na aplicação de um projeto considerando a cultura da empresa.

Porém, para entender melhor como cada uma define o processo de desenvolvimento foi necessário estudar cada uma das metodologias separadamente. Para compreender melhor as metodologias estruturadas foi estudado seu principal expoente, o *Rational Unified Process* – RUP – e como referencial para as metodologias ágeis foi adotado a *Extreme Programming* – XP – uma metodologia que nasceu na década de 90 e vem ganhando muitos adeptos.

Para reunir informações sobre cada uma das metodologias foi utilizada como metodologia de pesquisa a pesquisa bibliográfica. Baseada em livros de conceituados autores como Pressman, Sommerville e Kent Beck.

2. DESENVOLVIMENTO DE *SOFTWARE*: VISÃO GERAL

O desenvolvimento de *Software*, na década de 70, ficou conhecido pela Crise do *Software* (PRESSMAN, 2006), pois nesse período a produção de *Software* era feita de forma desorganizada, desestruturada e sem planejamento. O desenvolvimento de *software* era feito sem produção de documentação e a análise do projeto era feito sem a utilização de métodos. Com isso, prazo e custo não correspondiam a real necessidade.

Neste cenário surgiu a necessidade da criação de processos estruturados, planejados e padronizadas para o Desenvolvimento de *Software*, para que as necessidades fossem atendidas e os gastos com informatização de processos de informações se tornassem compensadores.

Com isso, surgiram as Metodologias de Desenvolvimento. Tais metodologias dividem o Desenvolvimento de *Software* em fases pré-definidas.

Mesmo utilizando técnicas avançadas de desenvolvimento e padrões consolidados de criação de *Software* características da Crise do *Software* perduram até hoje, como estimativas de custos e prazos errôneos.

2.1. O PROCESSO DE DESENVOLVIMENTO DE *SOFTWARE* SISTEMATIZADO

Inicialmente, foram utilizados conceitos típicos da engenharia para a padronização do desenvolvimento de *software*, que auxiliaram na sua sistematização, e mais tarde, levaram a criação da engenharia de *software*.

Fritz Bauer define a Engenharia de *Software* como: “a criação e utilização de sólidos princípios da Engenharia a fim de obter *software* de maneira econômica, que seja confiável para trabalhar eficientemente em máquinas reais”. (PRESSMAN, 2006)

A Engenharia de *Software* utiliza o Processo de Desenvolvimento, que consiste na criação de documentos, artefatos e marcos, capazes de representar o contexto do *software*, levando em consideração recursos, ferramentas, prazos, restrições, e outros aspectos que envolvem o desenvolvimento de um produto de *software*, para no final produzir *software* de qualidade. (PRESSMAN, 2006)

Os Métodos de Engenharia de *Software* fornecem técnicas para auxiliar o Processo de Desenvolvimento, abrangendo a análise de requisitos, projetos, construção de programas, testes e manutenção.

O processo de desenvolvimento pode ser subdivido em três fases genéricas que independem do tamanho ou complexidade do projeto.

2.2. FASES DO DESENVOLVIMENTO DE *SOFTWARE*

O desenvolvimento de *software* pode ser subdivido em três fases genéricas que independem do projeto, de sua complexidade, tamanho ou sua área de aplicação. As três fases são definição, desenvolvimento e manutenção.

2.2.1. DEFINIÇÃO

A fase de definição é uma das principais fases do projeto, onde se identifica funcionalidades, restrições, validações, interfaces e, principalmente os requisitos chave do projeto.

Nesta fase existe uma grande interação com o cliente para validar as informações recebidas e coletadas, a fim de que todos os requisitos-chave sejam atendidos na implementação do projeto.

É composta de três sub-tarefas principais, que podem variar de acordo com a metodologia utilizada, a Engenharia de Sistemas que define objetivos do sistema; o Planejamento do Projeto que determina com a máxima precisão os custos, tempo, esforço e recursos necessários para conclusão do projeto; e a Análise de Requisitos que quantifica e qualifica os requisitos específicos para conclusão do projeto de qualidade.

2.2.2. DESENVOLVIMENTO

Define como os dados serão estruturados e como a função deve ser implementada, é quando o projeto que antes estava documentado passa a ser transformado em código.

O Projeto de *Software* e a Geração de código são partes fundamentais dessa fase. O Projeto de *Software*, parte central do desenvolvimento, mostra o que e como será desenvolvido o *Software* e a Geração de Código é a tradução em linguagem de programação o que foi especificado no projeto de *software*.

A fase de testes é usualmente colocada nesta fase como parte das tarefas básicas, ela não pode deixar de existir, pois é nesta fase onde se encontram as não conformidades com o que foi especificado na fase de Definição.

2.2.3. MANUTENÇÃO

A fase de manutenção é a fase final, em que o produto é analisado e modificado. Seu foco principal são as modificações: correções de erros, adaptações necessárias e novas funcionalidades. A fase de manutenção engloba algumas características das fases anteriores, porém seu enfoque é um *software* já existente.

Quatro tipos de modificações podem ocorrer, durante a fase de manutenção. São elas:

- Manutenção Corretiva: corrige defeitos que ocorreram durante a fase de desenvolvimento;
- Manutenção Adaptativa: altera o *software* para adaptá-lo a alterações no ambiente externo, composto de variáveis que não do escopo do sistema, como uma mudança de sistema operacional;
- Manutenção Perfectiva: adiciona novas funcionalidades ao *software* que não estão no projeto original;
- Manutenção Preventiva: prepara o produto para que o impacto de alterações externas não afetem o sistema. Também é conhecida como Reengenharia de *Software*, que é a reconstrução de algo que já foi desenvolvido, para melhorá-lo.

2.3. MODELOS DE PROCESSO DE *SOFTWARE*

Modelos de Processos de *Software*s foram criados para tornar a atividade de desenvolvimento de *software* menos caótica e visam organizar o desenvolvimento utilizando técnicas e métodos (SOMMERVILLE, 2008).

Independentemente do projeto a ser desenvolvido os modelos de processos de *software* seguem um ciclo determinado onde existem quatro fases distintas que caracterizarão o projeto.

A primeira fase é a “Situação Atual” que define o ambiente, a “Definição do Problema” indica o problema específico a ser resolvido, o “Desenvolvimento

Técnico” resolve o problema de maneira que satisfaça as necessidades, e por último a Integração da Solução entrega a solução ao cliente.

Essas quatro fases formam um ciclo, dentro de cada uma dessas fases, existe outro ciclo com um problema em escala menor, e assim sucessivamente. Essa recursividade acaba em níveis racionais, quando uma solução ao problema é definido (PRESSMAN, 2006).

A seguir são descritos os modelos mais utilizados (SOMMERVILLE, 2008), (PRESSMAN, 2006).

2.4. MODELO EM CASCATA

Modelo proposto em 1970, também conhecido como Modelo Sequencial Linear onde as fases são sistematicamente seguidas de maneira linear (PRESSMAN, 2006).

É o modelo mais utilizado no mercado, porém não é considerado o mais eficaz, pois raros projetos seguem fluxo linear, além de mudanças de requisitos que ocorrem no projeto não serem de fácil adaptação, porque alteram toda a documentação desenvolvida (PRESSMAN, 2006).

Este modelo é dividido em etapas pré-definidas:

- Modelagem, análise do sistema onde o *software* será desenvolvido; Análise, os requisitos do *software* são levantados e definidos;
- Projeto, representação dos requisitos, é sub-dividido em quatro atributos: Estrutura de Dados, que é como os dados serão tratados, Arquitetura de *Software*, que define a base estrutural de como o sistema será desenvolvido, Caracterizações das Interfaces, representa o meio entre o usuário e o sistema, bem como entre os módulos do sistema e Detalhes Procedimentais (PRESSMAN, 2006);
- Codificação, traduz os requisitos das etapas anteriores em linguagem de máquina;
- Testes, identificam se as funcionalidades desenvolvidas estão funcionando perfeitamente;
- Manutenção, corrige erros encontrados após a entrega para o cliente, implementando melhorias ao produto – implicando em um novo ciclo de desenvolvimento.

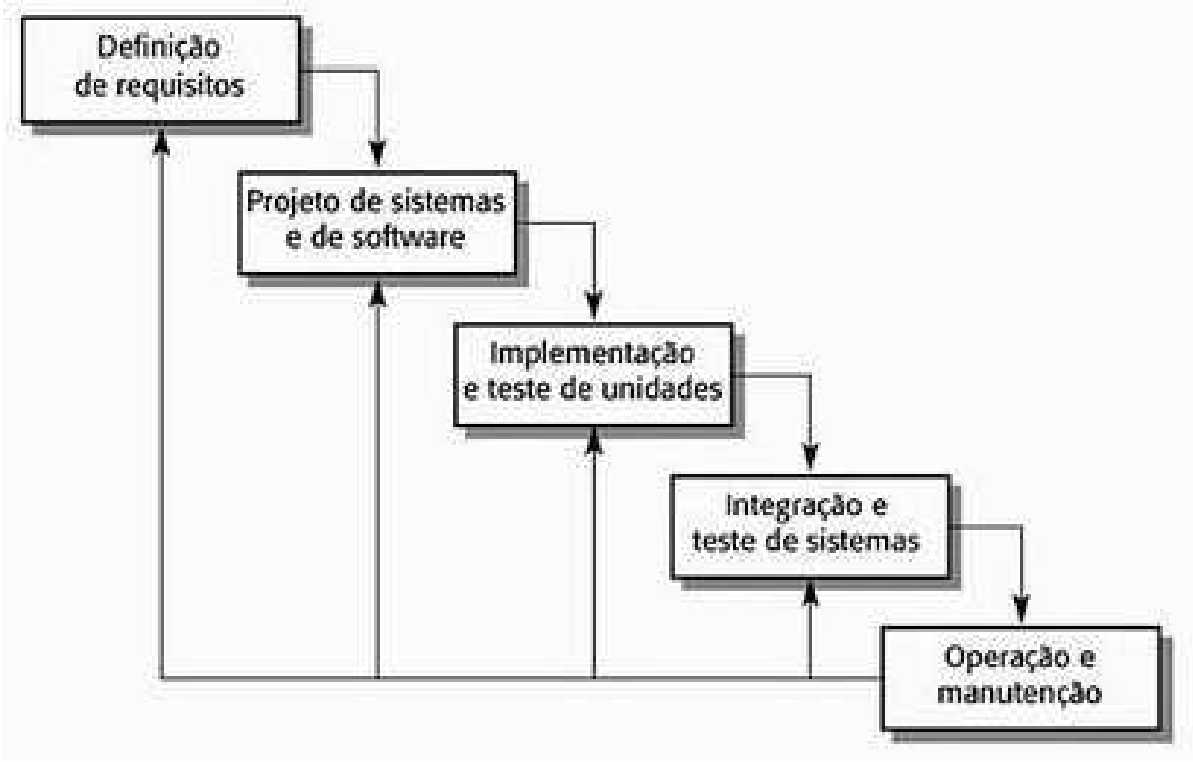

Figura 1: Modelo Cascata
Fonte: Sommerville, 2008

2.5. MODELOS INCREMENTAIS DE PROCESSO

Modelo de processo utilizado quando há necessidade de produzir *software* em incrementos.

2.5.1. MODELO INCREMENTAL

Esse modelo é uma adaptação do “Modelo Sequencial Linear”. Assume que o *software* agregará novas funcionalidades, e a cada nova funcionalidade ou conjunto de novas funcionalidades será um incremento e seguirá as fases do modelo linear (PRESSMAN, 2006).

O primeiro incremento deste modelo é o Núcleo do Produto, conterá as principais funcionalidades do sistema. Os próximos incrementos desenvolvidos agregarão funções ao Núcleo do Produto e aos incrementos anteriores (PRESSMAN, 2006).

O modelo incremental é mais utilizado em projetos longos, onde novas funcionalidades são acrescentadas ao longo do tempo e o prazo de entrega é curto.

Como há uma versão estável do produto ao final de cada incremento, o cliente vê a evolução do produto.

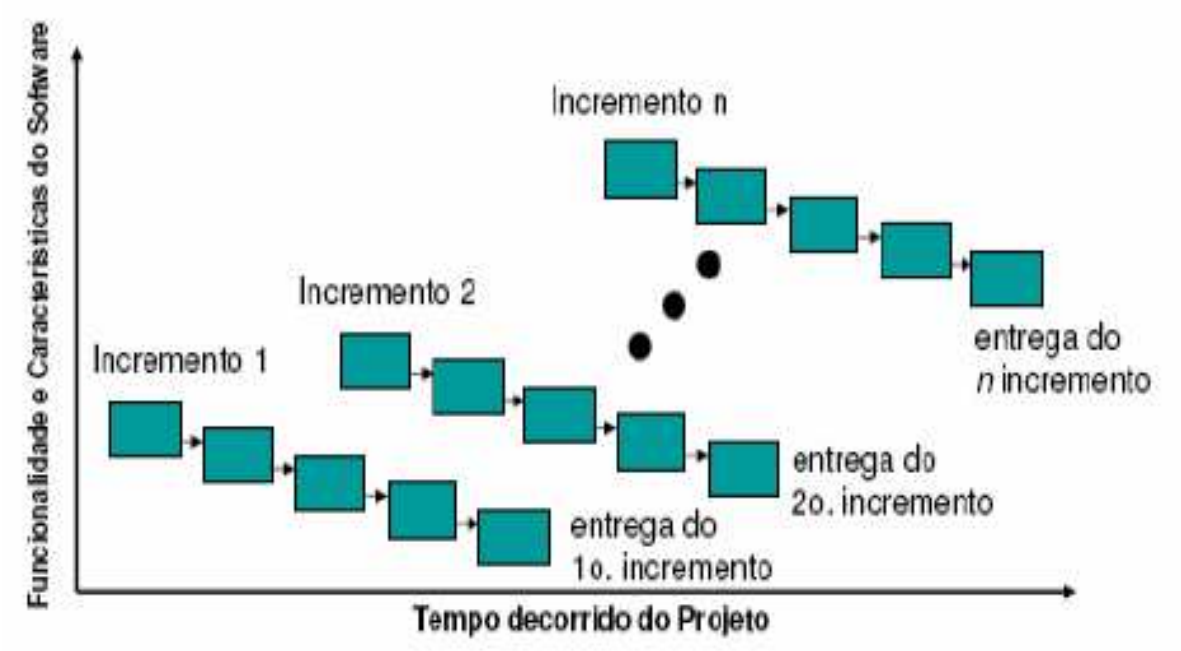

Figura 2: Modelo Incremental
Adaptado de: Pressman, 2006

2.5.2. MODELO *RAPID APPLICATION DEVELOPMENT* – RAD

O RAD é um modelo considerado uma adaptação do “modelo sequencial linear” para projetos de curta duração, usualmente com prazo máximo de 90 dias. Sua principal característica é a o desenvolvimento do produto em componentes, pois pode se reutilizar o código para que a equipe de desenvolvimento possa desenvolver um sistema completamente funcional em curto prazo.

O RAD, como os outros modelos, é subdividido em fases:

- Modelagem de Negócios: define arquiteturas que permitem o sistema utilizar as informações coletadas de forma efetiva;
- Modelagem de Dados: utilizam diagramas como o modelo de entidade e relacionamento (MER) para responder questões relacionadas a objetos de dados;
- Modelo de Processo: descreve os processos básicos para manipular as informações contidas nos objetos de dados gerados na fase anterior;
- Geração da Aplicação: criam ou reusam componentes já desenvolvidos utilizando ferramentas automatizadas;

- Teste e Entrega: testa a integração dos componentes criados em outros projetos, que foram reutilizados, e os que foram desenvolvidos para o atual projeto.

A principal característica do RAD é a modularização, e para isto os requisitos deverão estar definidos e ter independência entre eles.

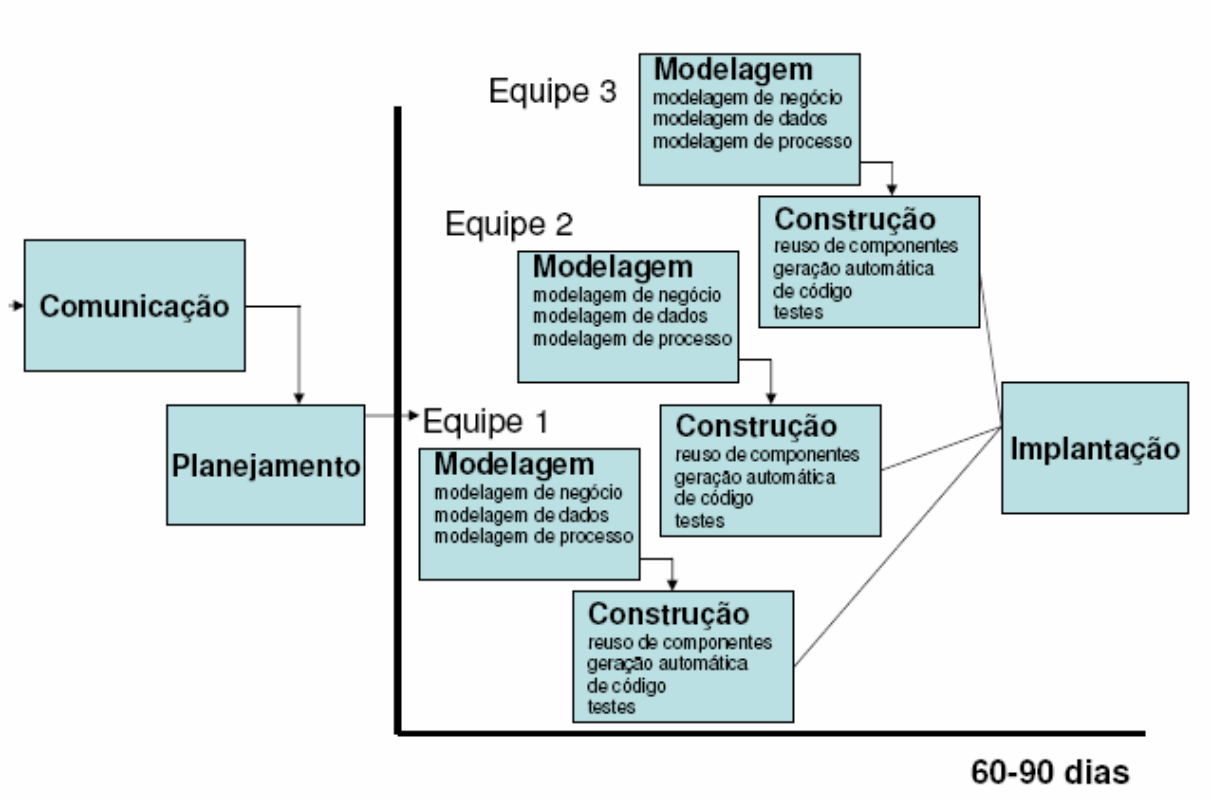

Figura 3: Modelo RAD
Adaptado de: Pressman, 2006

Para que o RAD seja utilizado de eficientemente é necessário que existam uma quantidade de recursos suficientes para montar as equipes RAD de desenvolvimento. Isto é considerado o principal problema nos grandes projetos pois há uma maior dificuldade de gerenciamento.

2.6. MODELOS EVOLUCIONÁRIOS DE PROCESSO DE *SOFTWARE*

Os “modelos evolucionários” têm como principal característica a interatividade, a cada nova versão, release, do sistema se agrega novas funcionalidades.

A seguir são descritos alguns exemplos de “Modelos Evolucionários”.

2.6.1. MODELO DE PROTOTIPAGEM

Esse modelo é utilizado quando alguns requisitos de sistema não são definidos de maneira clara pelo cliente, apenas seus objetivos, e não como os dados serão processados e como a saída será demonstrada.

Este modelo se caracteriza principalmente na criação de protótipos do sistema com as definições dadas pelo cliente. Esse protótipo é então testado pelo cliente para validar suas funcionalidades (PRESSMAN, 2006).

Após o teste do protótipo se faz um levantamento de novas funcionalidades requisitadas pelo cliente que serão desenvolvidas e adicionadas ao protótipo. Após todas as funcionalidades do sistema estiverem desenvolvidas e validadas, o protótipo é descartado e o sistema real é desenvolvido com base no que foi especificado no protótipo, dando ênfase as características não-funcionais: segurança, facilidade de manutenção e qualidade do produto.

O modelo de prototipagem gera resultados visíveis ao cliente de maneira veloz ao final de cada ciclo, demonstrando ao cliente de maneira fácil a evolução do sistema, sendo uma de suas maiores vantagens em comparação aos outros modelos.

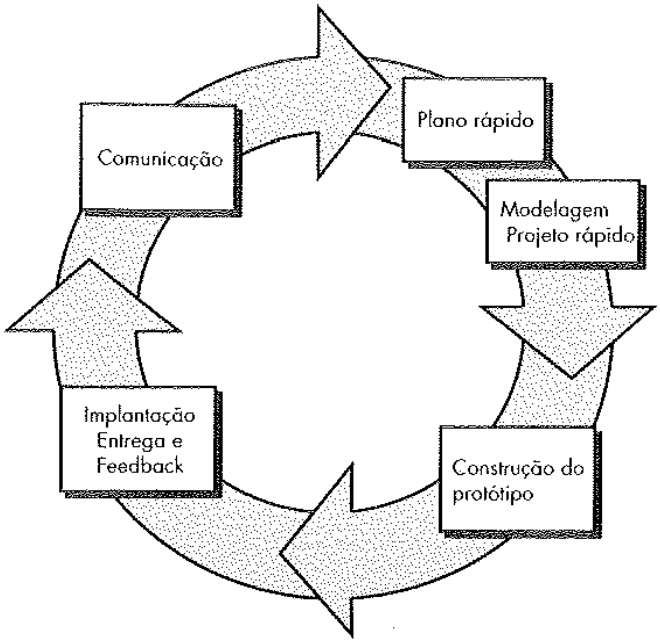

Figura 4: Modelo de Prototipagem
Fonte: Pressman, 2006

O protótipo por não levar em consideração variáveis de ambiente, como o sistema operacional onde o *software* será desenvolvido ou a linguagem de

programação utilizada, pode não utilizar soluções adequadas quando colocadas no projeto, e o *software* sendo baseado no protótipo pode conter erros caso não seja levado em consideração esses fatores ao transformar o protótipo no produto final (PRESSMAN, 2006).

2.6.2. MODELO ESPIRAL

O Modelo Espiral é um modelo evolucionário de processo de *software* que combina a natureza iterativa da prototipagem com os aspectos controlados e sistemáticos do modelo cascata (PRESSMAN, 2006).

Ao invés de representar o processo de *software* como uma sequência de atividades, o processo é representado como uma espiral (SOMMERVILLE, 2008).

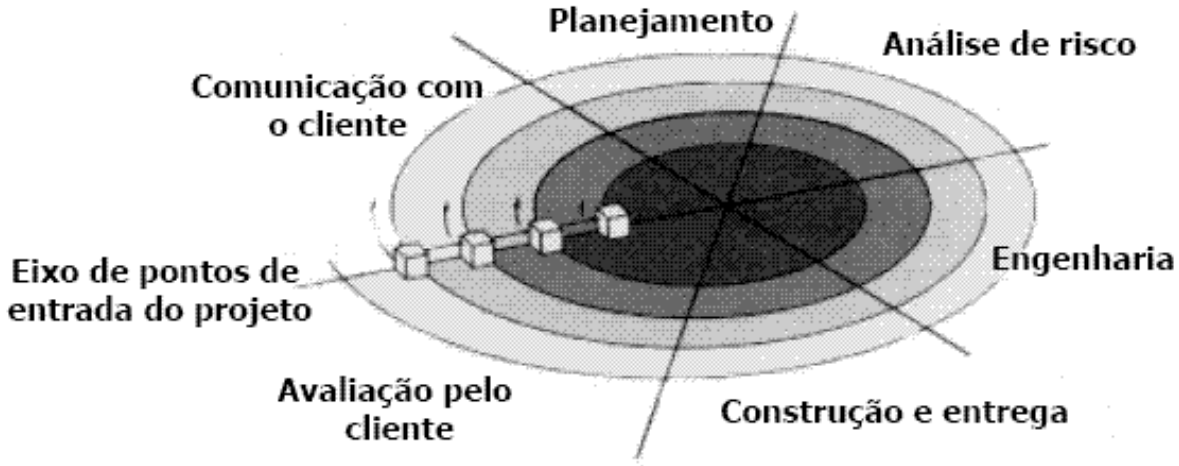

Figura 5: Modelo Espiral
Adaptado de: Pressman, 2006

Existem seis regiões, sendo que todas podem ser utilizadas:

- Comunicação com o Cliente: coleta de informações úteis ao sistema;
- Planejamento: definição custos e o tempo de desenvolvimento;
- Análise de Risco: gerenciamento e recuperação de riscos;
- Engenharia: construção da representação da aplicação;
- Construção e Liberação: implementação, teste, instalação e suporte da aplicação;
- Avaliação pelo Cliente: analise do *feedback* do cliente do incremento desenvolvido (PRESSMAN, 2006).

Nesse modelo há o acompanhamento da aplicação, após a entrega ao cliente o que o torna bastante utilizado pelas empresas, pois muitas empresas de desenvolvimento de *software* têm seu principal foco de atuação na manutenção.

Este modelo assume que usuários, analistas de desenvolvedores adquirem conhecimento do projeto com o decorrer do tempo, por isso é considerado um dos modelos mais realísticos (SOMMERVILLE, 2008).

3. METODOLOGIAS DE DESENVOLVIMENTO TRADICIONAIS OU “PESADAS”

Metodologia de Desenvolvimento é o conjunto de práticas recomendadas para o Desenvolvimento de *Software*s. Essas práticas podem ser subdividas em fases, para ordenar e gerenciar o processo (SOMMERVILLE, 2008).

As metodologias tradicionais, também conhecidas como “pesadas”, têm como característica marcante sua divisão em etapas ou fases. Essas fases são definidas e englobam atividades como Análise, Modelagem, Desenvolvimento e Testes.

Na conclusão de cada fase gerasse um marco, que pode ser um documento, como Diagramas de UML, um protótipo ou versão do *software*.

Metodologias pesadas geralmente são desenvolvidas no “modelo em cascata”, e a cada alteração do projeto, será necessário à volta ao inicio do projeto para alteração da documentação ou de outro marco (PRESSMAN, 2006).

3.1. RUP – RATIONAL UNIFIED PROCESS

Para O RUP será utilizado para analisar as metodologias pesadas, pois é o *framework* mais utilizado comercialmente.

O RUP foi desenvolvido pela *Rational® Software*, adquirido pela IBM que agora o mantêm. Esta metodologia foi criada para aumentar a produtividade da equipe desenvolvedora de *software*, mantendo a qualidade do *software* produzido e é derivado do Processo unificado, UP.

Este processo é um framework adaptável, pode ser utilizado tanto por pequenas equipes de desenvolvimento e por equipes de desenvolvimento de grande porte, pois é amplamente configurável.

A linguagem de modelagem UML (*Unified Modeling Language*) é utilizada para padronizar a documentação dos projetos desenvolvidos utilizando o RUP, UML foi originalmente criado pela *Rational® Software* e agora é mantido pela *Object Management Group* (OMG).

O RUP é uma metodologia iterativa, trabalha em ciclos e cada ciclo trabalha com uma versão nova do produto, cada ciclo é divido em quatro fases consecutivas: Concepção, elaboração, construção, transição.

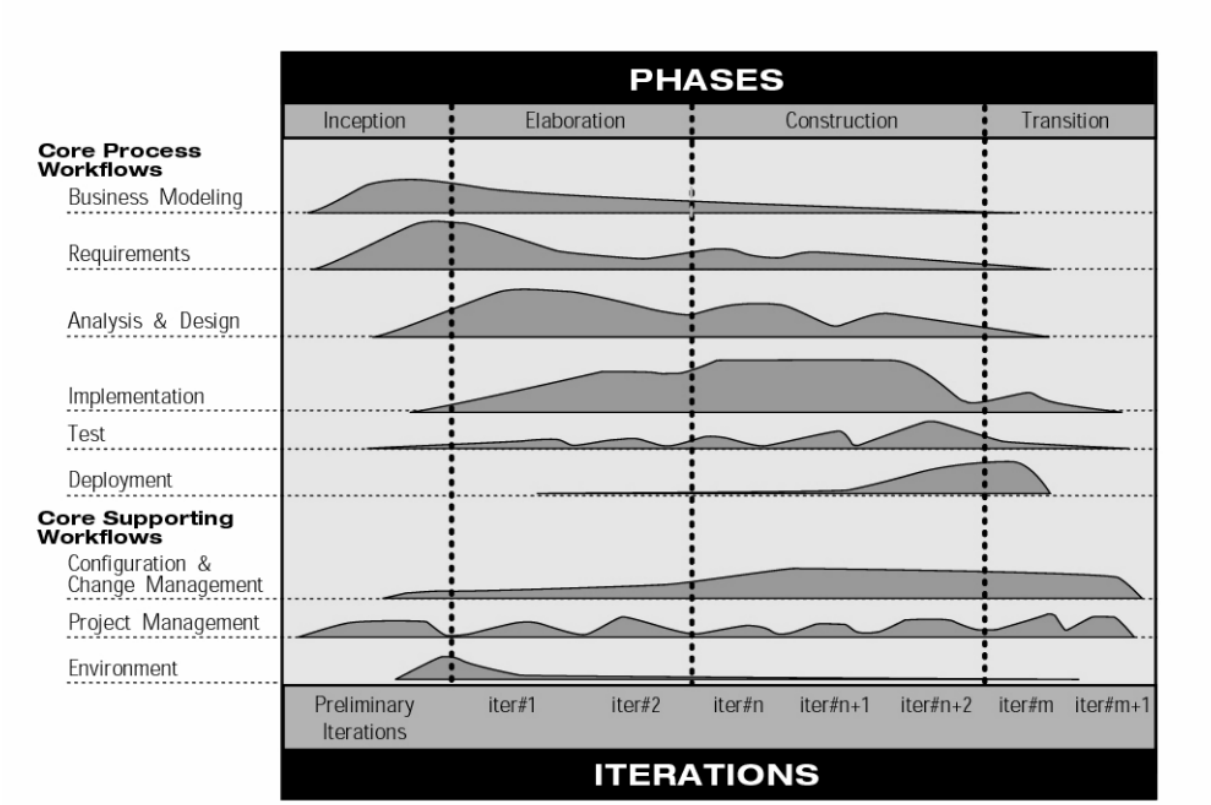

Figura 6: Relação fases x iterações do RUP
Fonte: Rational, 2012

3.1.1. FASES DO RUP

Concepção. Na fase de concepção delimita-se o escopo total do projeto, para isto se identifica todos os fatores envolvidos no projeto e suas interações. Nesta fase também se verifica a viabilidade econômica do projeto e a estimativa dos recursos que serão utilizados.

Elaboração. O propósito da fase de elaboração é analisar o problema principal, eliminar os principais riscos e definir a arquitetura do projeto.

Construção. A ênfase desta fase é o desenvolvimento dos componentes do projeto e outras funcionalidades do sistema. Nesta fase é onde ocorre a maior parte da codificação do projeto.

Transição. Esta fase é onde ocorre a “transição” do *software* de seu desenvolvimento para o usuário, tornando-o disponível e compreensível para o usuário final. Testa-se o sistema para validá-lo contra as expectativas dos usuários finais. Nesta fase ocorre também, o treinamento e capacitação dos usuários.

No final de cada fase existe um marco, *milestone*, bem definido onde decisões críticas devem ser feitas. Nesses *milestones* a fase anterior e seus resultados são analisados e verificam-se critérios específicos em cada *milestone*, caso não estejam em conformidade com os critérios estabelecidos deve se determinar se o projeto será abortado, re-planejado ou continuará sendo desenvolvido antes de passar para próxima fase.

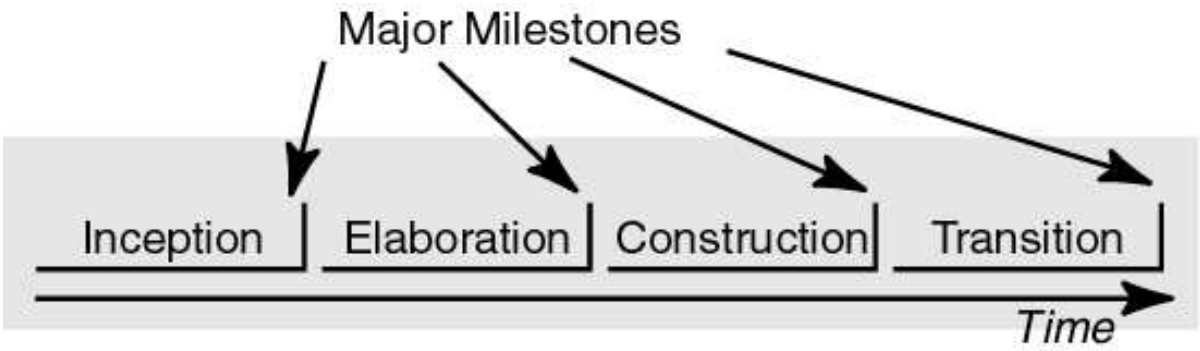

Figura 7: As fases e os marcos importantes no processo
Fonte: Rational, 2012

Cada fase do RUP também pode ser subdividida em iterações, neste caso uma iteração seria a repetição desta fase.

Além das fases outro conceito importante do RUP são os quatro elementos: funções, atividades, artefato, disciplinas (*workflows*).

Funções: define as responsabilidades de um individuo ou de um grupo de indivíduos, no RUP funções são como os indivíduos ou grupo de indivíduos devem exercer seu papel no projeto. Um membro da equipe do projeto geralmente desempenha muitas funções distintas.

Atividades: a atividade é uma unidade de trabalho que uma função pode ser solicitada a executar. A atividade tem um objetivo claro. Por exemplo, executar um teste de performance.

Artefato: artefato é uma informação produzida, utilizada por um processo de desenvolvimento de software, pode ser um documento, um modelo ou um software

Disciplinas: as disciplinas também conhecidas como workflows, são descrições em sequência das atividades, artefatos e papéis envolvidos para produzir um resultado observável. Existem nove disciplinas básicas no RUP, sendo seis de engenharia de software e três de suporte:

- **Modelagem de negócio (Business Modeling):** visa estabelecer uma comunicação entre a engenharia de software e a engenharia de negócios, e também deve garantir entendimento comum da organização entre os usuários e desenvolvedores. Cria uma visão da organização que pode ser entendida por todos os envolvidos no projeto;
- **Requisitos (Requirements):** define os escopos do sistema para isto deve se documentar e organizar as funcionalidades requisitadas no sistema;
- **Análise e Design (Analysis & Design):** o objetivo desta disciplina é demonstrar como o sistema será traduzido na fase de construção;
- **Implementação (Implementation):** o objetivo dessa disciplina é definir a organização do código, implementar objetos e componentes no software, testar os componentes desenvolvidos e integrar os resultados em um sistema;
- **Testes (Tests):** o objetivo dessa disciplina é testar o sistema verificando todas as interações e integração entre os objetos, identificando defeitos antes da implementação do software e verificar que todos os requisitos foram implementados corretamente;
- **Implantação (Deployment):** tem como objetivo o treinamento, instalação e suporte ao usuário final;
- **Gerência de Projeto (Project Management):** tem como objetivo o gerenciamento de riscos, planejamento e acompanhamento do projeto;
- **Gerenciamento de Configuração e Mudança (Configuration & Change Management):** define como controlar os diversos artefatos produzidos pelos diversos indivíduos para que não ocorra conflito;
- **Ambiente (Environment):** define o ambiente em que o software será desenvolvido para a equipe de desenvolvimento de software.

4. METODOLOGIAS DE DESENVOLVIMENTO ÁGEIS OU LEVES

O termo “Metodologias Ágeis” tornou-se popular em 2001 depois que um grupo de especialistas, em desenvolvimento de *software*, reuniram-se para compartilhar experiências e discutir medidas que aumentassem as chances de sucesso de um projeto.

Após algum tempo de pesquisa, esses especialistas publicaram um manifesto, que ficou conhecido como *Manifesto for Agile Software Development* [FOWLER 01]. Esse manifesto destacava quatro valores principais:

- Indivíduos e iterações mais que processos e ferramentas;
- *Software* funcional mais que documentação detalhada;
- Colaboração do Cliente mais que negociação de contratos;
- Responder às mudanças mais que seguir um plano.

4.1. MANIFESTO ÁGIL

É importante entender que o manifesto ágil não nos diz para esquecer os processos e ferramentas, a documentação, a negociação ou o planejamento, mas devemos tratá-los de maneira diferente, priorizando o foco em outros conceitos.

4.1.1. INDIVÍDUOS E ITERAÇÕES MAIS QUE PROCESSOS E FERRAMENTAS

De fato a capacitação e qualidade dos profissionais envolvidos no desenvolvimento do projeto implica diretamente nos resultados do produto tanto na qualidade como no desempenho da equipe no decorrer do projeto. Mas, onde ter os melhores profissionais não é certeza de sucesso, estes dependem do processo e em um processo mal definido, mesmo os melhores desenvolvedores podem não ser capazes de ter sucesso.

Além disso, é importante salientar que, mesmo com a combinação certa do processo adequado e bons profissionais, é necessário o entrosamento da equipe e a boa comunicação. Trabalhando em equipe, é mais produtivo um desenvolvedor mediano que saiba se comunicar com o restante do time do que um talento na programação que não consiga isto e trabalhe sozinho.

Vale lembrar que as ferramentas utilizadas são importantes e influenciam para o sucesso final do projeto, mas ainda não devem ser mais importantes que a qualidade e o nível de interação da equipe.

4.1.2. *SOFTWARE* FUNCIONAL MAIS QUE DOCUMENTAÇÃO DETALHADA

Em um projeto, a documentação é de suma importância para o seu sucesso, principalmente porque uma boa descrição do sistema ainda é necessária para auxiliar nas tomadas de decisão no decorrer de seu desenvolvimento e, além do mais, é mais simples para entender suas funcionalidades do que ler diretamente o código. Mas é preciso tomar muito cuidado com seu excesso, pois pode ser pior que sua falta, uma vez que se a documentação não estiver em sincronia com o andamento do projeto, elas distorcem sua realidade, fazendo com que decisões erradas possam ser tomadas.

O manifesto sugere que somente a documentação necessária e significativa seja gerada, além desta estar sempre em sincronia com o sistema.

Com o mínimo de documentação possível é necessário uma boa integração da equipe, pois o conhecimento sobre o projeto é transmitido durante o seu desenvolvimento, trabalhando em equipe.

4.1.3. COLABORAÇÃO DO CLIENTE MAIS QUE NEGOCIAÇÃO DE CONTRATOS

Para gerar um produto de boa qualidade, sucesso e aceitação do cliente, as metodologias ágeis afirmam que é necessário um *feedback* continuo dele, para garantir que o *software* esteja sendo desenvolvido de maneira que atenda suas necessidades atuais.

Os contratos devem determinar basicamente a forma como ocorrerá a comunicação e o relacionamento do cliente com a equipe de desenvolvimento. E não especificando custos, prazos e requisitos, pois no decorrer do projeto alguns requisitos podem se tornar dispensáveis como também pode surgir a necessidade de adicionar outros não previstos no contrato.

4.1.4. RESPONDER ÀS MUDANÇAS MAIS QUE SEGUIR UM PLANO

Sabendo que requisitos são mutáveis e mudanças nas especificações vão ocorrer, o melhor para o projeto é estar pronto para se adaptar às mudanças que irão aparecer.

O planejamento deve ocorrer, mas para períodos menores. Pois, como as alterações no projeto são inevitáveis planos muitos longos serão difíceis de ser concretizados.

4.2. XP – *EXTREME PROGRAMMING*

A *Extreme Programming* (XP) é uma metodologia ágil para equipes pequenas e médias que desenvolvem *software* baseado em requisitos vagos e que se modificam rapidamente (BECK, 2004). Dentre as principais diferenças da XP em relação às outras metodologias estão:

- *Feedback* constante;
- Abordagem incremental;
- Encorajamento de comunicação face a face.

O XP é uma abordagem deliberada e disciplinada para o desenvolvimento de *software*, que vem sendo bastante utilizada e vem tomando espaço que antes pertencia a metodologias tradicionais, como RUP – *Rational Unified Process* (BECK, 1999).

4.2.1. VALORES DA XP

A XP, como todas as metodologias, visa garantir a satisfação do cliente, enfatizando o desenvolvimento ágil do projeto para o cumprimento das estimativas. Seus adeptos devem seguir basicamente quatro valores (BECK, 2004):

- Comunicação.
- Simplicidade.
- *Feedback.*
- Coragem.

O princípio da comunicação visa manter o melhor relacionamento possível entre o cliente e desenvolvedores. Este deve ser feito de maneira pessoal evitando ao máximo os contatos por telefone, e-mails, etc. Aplica-se tanto para o relacionamento cliente/desenvolvedor como equipe/gerente.

Simplicidade é criar códigos simples que não devem ter funções desnecessárias. Entende-se por código simples um *software* com o menor número possível de classes e métodos, contemplando somente requisitos atuais evitando funcionalidades futuras. Para a XP vale mais a pena perder um pouco de tempo

para adicionar novas modificações no *software* do que implantar um código mais complicado que talvez possa nem ser utilizado, ainda mais levando em consideração que requisitos são mutáveis.

Constantes *feedbacks* auxiliam os programadores e terem informações atuais tanto do código como do cliente. Testes contínuos podem apontar erros tanto do módulo a ser implantado como deste integrado ao *software*. Para o cliente, o constante *feedback* serve para ele ter sempre uma versão do *software* para avaliar e ver sua evolução, apontando novas melhorias a serem feitas. Dessa forma, eventuais erros são identificados rapidamente e então corrigidos para as próximas versões do *software*. Tendo a constante avaliação do cliente o produto final será conforme as expectativas dele.

Ter coragem para adotar os três valores. Nem todos têm facilidade em se comunicar. Tentar simplificar o código sempre que tiver a possibilidade para tal. Estar pronto para cobrar e receber os constantes *feedbacks* do cliente.

4.2.2. AS ATIVIDADES BÁSICAS

> “Você codifica porque se você não codificar você não terá nada. Você testa porque se você não testar você não saberá quando você terminou de codificar. Você ouve porque se você não ouvir você não saberá o que codificar ou o que testar. E você projeta para que você possa codificar, testar e ouvir indefinitivamente” (BECK, 2004).

De acordo com Kent Beck, a XP possuí quatro atividades básicas:

- Codificar;
- Testar;
- Ouvir;
- Projetar.

E estas são trabalhadas em doze práticas:

Jogo do planejamento: defina o escopo da próxima versão e acerte as prioridades de negócio e estimativas técnicas. Caso ocorra algum problema, atualize o planejamento.

Entregas frequentes: idealizando o prazo de entrega de aproximadamente um a dois meses para cada versão do *software* e trabalhando juntamente com o constante *feedback* do cliente para evitar surpresas na entrega do *software* final.

Metáfora: crie uma historia simples sobre como o sistema funciona com a finalidade de passar para o projeto o que realmente o cliente espera.

Projeto simples: código mais simples possível contemplando somente os requisitos atuais sem se preocupar com os requisitos futuros. Estes somente serão adicionados ao sistema quando realmente for necessário.

Testes: a XP focaliza a validação do projeto durante todo o processo de desenvolvimento. Os programadores desenvolvem o *software* criando primeiramente os testes.

Refatoração: sempre que houver a oportunidade de simplificar o código, remover duplicidade e acrescentar flexibilidade ao sistema os programadores devem reestruturá-lo, mas mantendo sempre seu comportamento inicial.

Programação em pares: a codificação em feita por duplas. Desta forma o código é sempre revisto por dois desenvolvedores diminuindo erros sintáticos e focando na melhoria do código. Estes dois devem sempre alternar suas funções.

Propriedade coletiva: qualquer membro da equipe pode modificar qualquer parte do código, desde que este passe pelos testes necessários.

Integração contínua: sempre que uma tarefa for finalizada, integre e atualiza a versão do sistema.

Semana de 40 horas: trabalhe 40 horas por semana, não faça horas extras por duas semanas seguidas, pois isto indica falha no projeto, que deve ser re-analisado.

Cliente presente: considere o cliente um membro da equipe disponível a qualquer hora, para que este tire duvidas em relação aos requisitos a serem implementados.

Padrões de codificação: os códigos deverão respeitar padrões definidos pela equipe.

4.2.3. CINCO REGRAS

O aspecto mais surpreendente da *Extreme Programming* são suas regras simples, que a deixa muito parecido a um quebra-cabeça. Há muitos pequenos pedaços. Individualmente as peças não fazem sentido, mas quando combinados formam uma imagem completa. As regras podem parecer estranhas e talvez até

ingênuas no início, mas são baseadas em valores e princípios sólidos (WELLS, 2012).

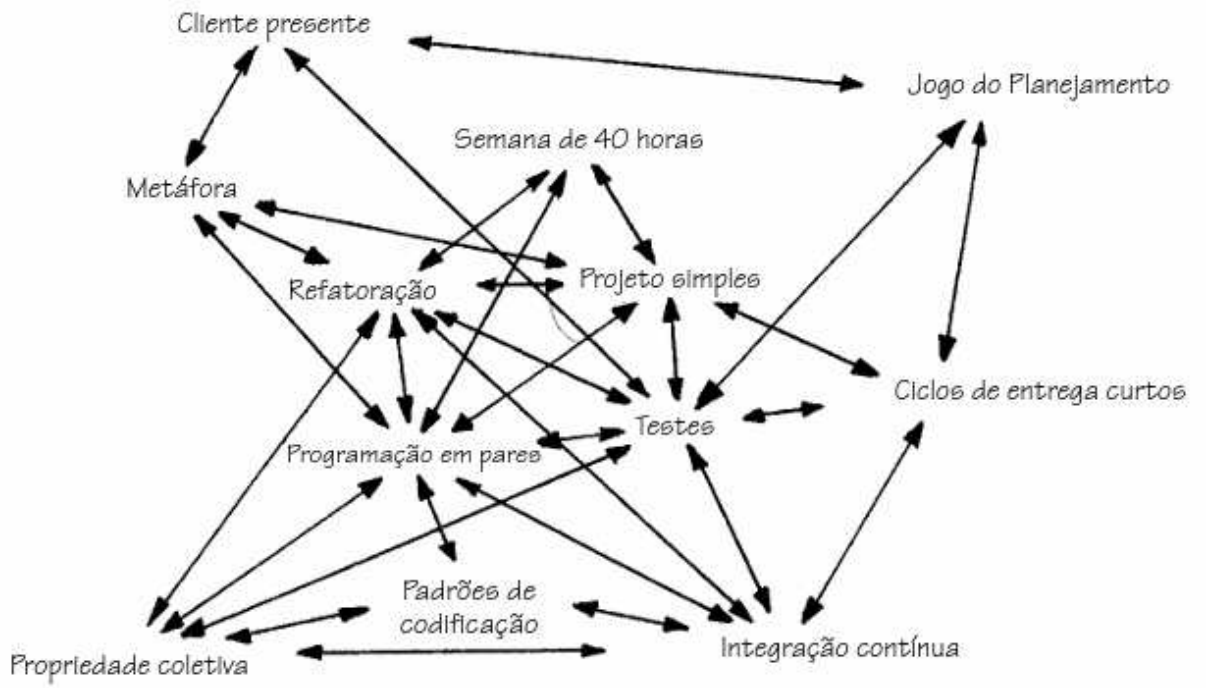

Figura 8: As práticas reforçam uma as outras
Fonte: Beck, 2004

Don Wells utiliza as doze praticas em cinco regras:

- Planejando
- Gerenciando
- Projetando
- Codificando
- Testando

Planejando

O cliente escreve cartões de histórias (*user stories*) que geralmente seguem o padrão de três sentenças, e são utilizadas suas terminologias evitando termos técnicos.

Os cartões de historias são utilizados na definição de prazos das reuniões para planejamento de liberação (*Release Planning Meeting)*. Cada história terá seu tempo de desenvolvimento ideal estimado em uma, duas ou no máximo três semanas, caso essa estimativa seja menor que uma semana ela deverá combinar outras histórias e se for maior que três semanas deverá ser quebrada para se

encaixar nos prazos. O tempo de desenvolvimento ideal é a estimativa do tempo necessário para transformar a história em código, considerando que não haverá distrações, nenhuma outra atribuição e o desenvolvedor sabendo exatamente o que deverá ser feito, mas incluindo ainda os testes.

Essas reuniões têm por objetivo estimar cada história em semanas de programação ideal, definir suas prioridades (junto ao cliente), quantas histórias podem ser aplicadas antes de uma determinada data ou por quanto tempo um conjunto de histórias vai demorar para terminar e quantas podem ser desenvolvidas por iteração. A equipe de negócios deve definir o escopo, prioridade, composição das versões e datas de entrega, mas para eles tomarem essas decisões precisam das informações da área técnica como suas estimativas, consequências, processo e cronograma detalhado.

O projeto pode ser quantificado por quatro variáveis:

- Escopo: o quanto deve ser feito.
- Recursos: quantas pessoas estão disponíveis.
- Tempo: quando o *software* ou uma versão dele será entregue.
- Qualidade: quão bom será o *software* e o quanto dele foi testado.

Divida o cronograma de desenvolvimento do projeto em mais ou menos seis iterações de uma a três semanas. Para cada iteração deve se concentrar o esforço nas funcionalidades mais importantes.

No inicio de cada iteração é feito uma reunião onde o cliente escolhe quais as histórias inclusas para esta iteração e suas prioridades. As histórias escolhidas e os testes fracassados são divididos em tarefas para os programadores.

Frequentemente, libere novas versões do sistema para obter o *feedback* do cliente. Quanto mais tempo levar para introduzir uma funcionalidade importante menos tempo terá para corrigi-lo.

Gerenciando

A programação em pares e a propriedade coletiva devem ser estimuladas começando pelo ambiente de trabalho e disposição de mesas e computadores.

Defina um ritmo padrão para acompanhar o progresso do projeto. Um *software* incompleto ou com *bugs* é impossível de ser medido, pois envolve uma quantidade desconhecida de esforço futuro. Caso perceba que não será concluída as tarefas da iteração refaça o escopo da iteração. Vale mais concentrar o foco da

equipe em uma única tarefa concluída com sucesso do que varias tarefas incompletas.

Faça reuniões informais diariamente para a equipe compartilhar idéias e soluções para o projeto. Nessas reuniões, os desenvolvedores relatam pelo menos três coisas: o que foi concluído no dia anterior; o que será tratado no dia e os problemas que estão lhe causando atrasos. Cerca de quinze minutos por dia é o suficiente.

A velocidade do projeto (*Project Velocity*) é a quantidade de historias que foram concluídas durante uma iteração. Ela pode ser mensurada por escopo, que vai basear quanto tempo durará a próxima iteração, de acordo com a quantidade de histórias implementadas na iteração anterior. Como também pode ser mensurada, por tempo, que determina a quantidade de histórias que serão desenvolvidas na próxima iteração. Esse modelo por tempo facilita na estimativa da próxima iteração considerando que estas terão o mesmo tempo de desenvolvimento.

Um remanejamento constante da equipe tem por objetivo distribuir o mesmo conhecimento do projeto para toda a equipe, evitando que este se acumule em poucas pessoas que podem ocasionar transtornos futuros. Isso porque se estas pessoas estiverem muito atarefadas ou se por algum imprevisto alguém sair o projeto como consequecia haverá um atraso, inevitavelmente.

Siga as regras da XP, pois todos os desenvolvedores devem saber o que esperar dos outros, e um conjunto de regras é o que define estas expectativas. Mas não hesite em alterá-las, caso perceba que não está funcionando. Faça reuniões para discutir o que está ou não funcionando na XP, visando maneiras de melhorar seus processos e adequá-los ao time.

Projetando

Focando na simplicidade do código para otimizar o tempo de desenvolvimento e para uma possível alteração futura. Sempre que for encontrado um código complicado substituí-lo por um mais simples.

Escolha uma metáfora do sistema com o objetivo de explicar o projeto do sistema sem precisar recorrer a documentos enormes. Assim, novas pessoas poderão contribuir com o projeto mais rapidamente.

Nunca implante uma funcionalidade precocemente, pois estes podem nunca ser utilizados. E concentre-se apenas nas atividades do dia.

Refatorar em todo o ciclo de vida do projeto economiza tempo e aumenta qualidade. Refatorar impiedosamente para manter o projeto simples e para evitar confusão desnecessária e complexidade. Mantenha seu código limpo e conciso, de modo que é mais fácil de compreender, modificar e estender.

Codificando

O cliente deve estar presente durante todos os processos, é muito importante a comunicação dele com a equipe de desenvolvimento, sempre que necessário. O cliente, auxiliado pelo time de desenvolvimento, define as histórias e seus prazos. Ajuda a garantir que a funcionalidades necessárias do sistema estejam cobertas nas histórias. Negocia quais histórias serão incluídas na liberação de uma nova versão, como também quando será feita essa liberação. Pequenas versões incrementadas são feitas para que o cliente possa ir testando as funcionalidades e dar um *feedback* mais rápido para os desenvolvedores, porque estes precisam obter mais detalhes das histórias, que não estão escritas nos cartões, para completar a codificação. Participa também da fase de testes, estipulando um resultado a ser atingido.

Criar uma unidade de teste antes de começar a codificar facilita a desenvolver e considerar o que realmente precisa ser feito, mantendo o código simples.

Todo código deve ser gerado por duas pessoas trabalhando em um único computador, isto gera mais qualidade ao *software.*

Integrar frequentemente evita divergência no código como a fragmentação do desenvolvimento, onde os desenvolveres não estão se comunicando sobre o que poderia ser reutilizado. Todo mundo precisa trabalhar com a versão mais recente para evitar alterações em códigos obsoletos que causam problemas na integração.

Utilizar um único computador para integração auxilia, no controle de liberação de novas funcionalidades. Quando este computador está sendo utilizado nenhuma outra liberação pode ocorrer, garantindo estabilidade.

A propriedade coletiva encoraja todos a contribuir com idéias novas para todos os segmentos do projeto. Qualquer desenvolvedor pode alterar qualquer linha de código para adicionar funcionalidades, corrigir *bugs*, melhorar códigos ou refatorar.

Testando

Todo código deve ter sua unidade de teste. Sempre que adicionada novas funcionalidades ao repositório, inicia uma nova bateria de testes onde os códigos devem passar por todos os testes de unidade antes de ser lançado assegura que todas as funcionalidades sempre funcionaram.

Quando um *bug* é encontrado é criado um teste para garantir que este não ocorra novamente.

Durante as reuniões, de planejamento da próxima iteração, são criados os testes de aceitação que tomam como base os cartões de histórias. Uma história não é considerada completa enquanto não passar por seus “testes de aceitação”.

Os clientes são responsáveis por verificarem a exatidão dos “testes de aceitação” e rever os resultados para decidir quais falharam nos testes.

5. ANÁLISE COMPARATIVA ENTRE METODOLOGIAS TRADICIONAIS E ÁGEIS

A maioria dos processos possui elementos em comum que tornam possível uma comparação sistemática (SMITH, 2012). Estes elementos em comum são:

- Alocação de tempo e esforço;
- Artefatos;
- Atividades;
- Funções;
- Disciplinas.

5.1. ALOCAÇÃO DE TEMPO E ESFORÇO

A vida de um projeto RUP é divido em quatro fases, Iniciação, Elaboração, Construção e Transição. Cada fase ainda é dividida em iterações e essas iterações podem ter vários ciclos.

Para a maioria dos projetos RUP, a duração de suas iterações leva de duas semanas até seis meses, e o número de iteração durante toda sua vida é de três a nove iterações. Portanto, um projeto RUP abrange uma faixa de tempo entre seis semanas até cinquenta e quatro meses (SMITH, 2012).

No XP, as novas versões do sistema, liberadas para o cliente com o tempo médio de dois em dois meses, equivalem às iterações do RUP, isto para projetos apropriados ao XP, ou seja, para aqueles com cerca de dez pessoas na equipe, pois, mais que isso, seria difícil aplicar a suas práticas.

Utilizando o COCOMOII (BOEHM, 2012), para estimar custo, esforço de pessoal e tempo, um time de dez programadores (idealizado como numero máximo para um projeto XP) é o ideal para um projeto de doze meses de duração com mais ou menos dez mil linhas de código fonte (SLOC).

Projetos que durem mais tempo que o especificado pelo modelo certamente devem ter uma equipe de desenvolvimento maior, pois, pode provocar alguns problemas, tendo em vista que a equipe desenvolverá de maneira serial, enquanto se fosse maior desenvolveria paralelamente.

Então, para sistemas de menores dimensões, que estão dentro dos limites padrões da XP, as iterações do RUP são mais ou menos equivalente aos lançamentos do XP.

Isto não ocorre em projetos grandes, pois estes estão fora dos limites definidos pela XP. Para que ocorra o desenvolvimento em paralelo as pessoas são divididas em equipes e estas ficarão responsáveis, cada uma, por um subsistema do projeto. Normalmente, todos os subsistemas se encaixam nos padrões de desenvolvimento da XP, mas como o projeto precisa ser analisado como um todo, seu escopo ultrapassa as limitações da XP.

5.2. ARTEFATOS

Artefato são produtos criados e utilizados durante os processos. Estes servem para capturar e transmitir as informações durante todo o projeto.

No RUP a comunicação se baseia nos artefatos. Já na XP, a comunicação é oral e direta, limitando o desenvolvimento do projeto por equipes separadas geograficamente.

Uma das principais vantagens apontadas pela metodologia ágil, durante todo o projeto, é ser menos burocrática e gerar menos documentos que as metodologias tradicionais. Sendo que hoje o RUP é capaz de gerar mais de cem documentos diferentes. Mas, muitos desses documentos não precisam ser gerados, sendo apenas escritos quando existe a necessidade dos mesmos, normalmente de acordo com o tamanho do projeto.

A XP segue as práticas do manifesto ágil, portanto, somente serão gerados os documentos que forem considerados realmente necessários. Entretanto, no decorrer de um projeto, sempre serão exigidos alguns documentos, seja na forma de cartões de histórias feitas pelo cliente ou como planos para liberação de uma versão, ambos necessários para o andamento do projeto.

5.2.1. ARTEFATOS PARA PEQUENOS PROJETOS

A comparação será feita em relação aos pequenos projetos, pois, como visto anteriormente, em uma escala maior não é possível aplicar certas práticas utilizadas pela XP.

Adaptando o RUP para pequenos projetos, o número de documentos pode ser reduzido a uma quantidade menor, comparada com o que deve ser documentado nos projetos de grande porte (SMITH, 2012). Esse número pode se comparar à quantidade de documentos que podem ser gerados em projeto,

utilizando a XP. De acordo com suas práticas, sendo esclarecido em alguns documentos aquilo que a XP não trata ou pelo menos não abertamente ou documentalmente. A Tabela 1 mostrada abaixo ilustra um exemplo de uma possível adaptação do RUP para projetos menores.

Artefatos de XP	Artefatos de RUP
Histórias Documentação adicional a partir de conversações	Visão Glossário Modelo de Casos de Uso
Restrições	Especificações Suplementares
Testes de aceitação e testes unitários Dados do teste e resultados do teste	Modelo de Teste
Software (código)	Modelo de Implementação
Releases	Produto (Unidade de Implantação) Notas de Release
Metáfora	Documento de Arquitetura de *Software*
Design (CRC, esboço de UML) Tarefas técnicas e outras tarefas Documentos de design produzidos no final Documentação de suporte	Modelo de Design
Padrões de codificação	Guia de Programação
Espaço de Trabalho Ferramentas e framework de teste	Ferramentas
Plano de release Plano de iteração Estimativas de tarefas e de histórias	Plano de Desenvolvimento de *Software* Plano de Iteração
Orçamento e plano geral	Caso de Negócio Lista de Riscos
Relatórios em andamento Registros de tempo para tarefas Dados de métricas (incluindo recursos, escopo, qualidade, Rastreamento de resultados Relatórios e notas sobre reuniões	Avaliação de Status
Defeitos (e dados associados)	Solicitações de Mudança
Ferramentas para gerenciamento de código	Repositório do Projeto Espaço de Trabalho
Pico (solução)	Protótipos Protótipo de Interface do Usuário Prova de Conceito Arquitetural
[Não incluído no XP]	Modelo de Dados Material de Suporte para o Usuário.

Tabela 1: mapeamento de artefatos XP para um projeto pequeno em RUP

Adaptado de: Smith, 2012

Embora varie a quantidade de artefatos em ambos os lados em geral os artefatos do RUP para projetos pequenos são mapeados de um projeto XP sem nenhum problema.

5.3. ATIVIDADES

Para o RUP, atividade é uma unidade de trabalho de uma função e possui um objetivo claro. Seu objetivo é trabalhar os artefatos de modo que estes se tornem menos abstratos e mais próximos das metas definidas para o projeto.

A XP já trabalha de forma mais simplificada e não muito especifica. Possuem quatro atividades básicas que são: codificar, testar, ouvir e projetar.

5.4. FUNÇÕES

A maior diferença, em relação às funções, é a quantidade apresentada em cada metodologia. No RUP, temos listado trinta funções especificadas dentro de suas nove disciplinas, enquanto que na XP temos sete funções sem atribuições especificas dentro de suas atividades.

As funções, em ambas as metodologias, podem ser desempenhadas por uma equipe ou mesmo por um único indivíduo e estes pode desempenhar mais de uma função, principalmente no caso da XP que geralmente trabalha com equipes pequenas.

5.5. DISCIPLINAS

Em relação às disciplinas, o RUP possui nove ao todo e as divide em seis de engenharia de *software* e três de suporte. O equivalente na XP seria as doze práticas definidas por BECK (2012), como o Jogo do Planejamento na XP com a disciplina Gerência de Projeto no RUP

Mas nem todas as atividades e práticas da XP possuem uma disciplina equivalente no RUP, como também nem todas as disciplinas do RUP podem ser mapeadas nas práticas da XP. Por exemplo, dependendo da escala do projeto, a Propriedade Coletiva da XP não seria uma prática aplicável no RUP.

5.6. CONCLUSÃO DO COMPARATIVO

Observando as comparações conclui-se que ambas as abordagens possuem, originalmente, focos diferentes. Enquanto o *Rational Unified Process* foi feito para trabalhar com projetos grandes, a *Extreme Programming* lida com projetos pequenos.

5.7. CASOS DE SUCESSO

Os tópicos abaixo apresentam dois casos de sucesso. O primeiro deles da implantação do XP na LocaWeb. O segundo da padronização da metodologia de desenvolvimento de sistemas da Ford Moto Credit Company, utilizando RUP.

5.7.1. LOCAWEB

A Locaweb é uma das maiores empresas de hospedagem de sites da América Latina. No final de 2006, Daniel Cukier começou a usar XP na equipe de desenvolvimento de serviços de voz (LocaWeb Telecom). Usando a metodologia, a equipe conseguiu entregar funcionalidades de forma rápida e com baixo índice de *bugs* (AGILCOOP, 2012).

A diretoria da empresa percebeu as vantagens dos métodos ágeis e decidiu realizar um treinamento envolvendo toda área de tecnologia. Em agosto de 2007, Daniel e Maurício Dediana realizaram um curso para 85 pessoas das áreas de desenvolvimento e produtos da empresa.

A partir daí, as equipes começaram a adotar *Scrum* e algumas práticas de XP. Os diretores ficaram muito satisfeitos com o resultado. Hoje a empresa é muito mais eficiente em entregar *software* de qualidade para seus clientes.

Vale ressaltar que foram utilizados padrões para introduzir novas idéias para disseminar a inovação dentro da empresa. Sem esses padrões, talvez não tivesse ido tão longe (AGILCOOP, 2012).

5.7.2. FORD MOTOR CREDIT COMPANY

A companhia de crédito da *Ford Motor* se destaca como uma das maiores companhias financeiras automotivas do mundo com operação em 36 países desde 1959. A companhia administra cerca de US$150 bilhões e financia para concessionárias e clientes de 8 empresas, entre elas a própria *Ford Motor, Jaguar e Volvo*.

Em todo o mundo a companhia de crédito da *Ford Motor* emprega aproximadamente 14000 funcionários incluindo 700 profissionais da área de TI que projetam, constroem, testam e mantêm o sistema central da empresa.

Recentemente seu *software* de desenvolvimento foi padronizado com a metodologia IBM *Rational Unified Process* (RUP), porém adaptado conforme suas

necessidades. Após iniciar com a metodologia J3EE, a implantação do sistema foi realizada com sucesso e personalizaram sua versão do RUP com a chamada *Unified Solution Delivery Methodology* (USDM).

Para chegar a esse sistema, foram propostos opções para o desenvolvimento de um sistema que suprisse as necessidades da empresa. As diferentes equipes tinham seus projetos peculiares, dificultando o compartilhamento das informações. Outra dificuldade foi a percepção tardia dos riscos que poderiam ter.

Logo uma nova abordagem era necessária e a equipe avaliou possíveis soluções. Eles procuraram por uma metodologia que fosse fácil de personalizar e de integrar aos processos já existentes. A escolha do RUP aconteceu devido a maioria dos desenvolvedores já conhecer e da gama de recursos disponíveis.

Aprovado o sistema a companhia instituiu um programa de treinamento, com esse treinamento as equipes receberam orientação e suporte prático de técnicos do RUP. Assim, novos projetos começaram.

Com o USDM a equipe precisaria manter o padrão e garantir que os processos fossem aplicados da mesma maneira entre todas as equipes, então criaram um modelo de treinamento em que os treinadores atuariam proativamente com suas equipes, estando não apenas no inicio do processo ou quando houvesse dificuldades, mas em todo o processo para ter um *feedback* instantâneo.

No passado a companhia utilizava uma metodologia que eram descritos e passados para as equipes. Essa metodologia foi substituída por um processo detalhado em uma Web site que é atualizado periodicamente. O que reduz custos e garante que as equipes obtenham as informações rapidamente.

Para garantir que as equipes tomem medidas para evitar riscos, a companhia criou um mapeamento de processo para priorizar riscos associados a cada caso de uso do sistema.

E mesmo com diferentes equipes espalhadas pelo mundo ter a metodologia padrão, mostra um benefício significativo uma vez que todos usam a mesma terminologia, e os mesmos artefatos, o que simplifica a comunicação e evita maiores confusões.

A companhia de crédito *Ford Motor* com a utilização do RUP tem sido capaz de atender as necessidades dos usuários e de seus negócios assim como as novas demandas que possam surgir. Eles lidam com riscos antes a fim de evitar surpresas

no final, assim, a equipe de TI tem conseguido satisfazer os usuários de negócios e atender aos altos padrões de qualidade da empresa.

6. CONCLUSÃO

De modo geral, a XP é utilizada em projetos que possuem requisitos mutáveis ou onde estes mesmos não são claros para a equipe e para o cliente, porém ela possui limitações que dificultam sua prática:

A XP não está pronta para definir as funções, de cada membro da equipe, em um grande projeto, muito menos gerenciar a quantidade necessária de pessoas para a conclusão deste tipo de projeto, pois falharia devido à ênfase na comunicação oral e falta de documentação.

Outro ponto importante a ser considerado é a cultura do cliente e também da equipe. Há clientes que precisam da documentação para manter sua sensação de controle sobre o projeto, como também podem não estar sempre disponíveis para dialogar. Isto causa atrasos significativos em um projeto XP. Para os programadores, mesmo os mais capacitados, podem sentir dificuldades em trabalhar nas definições da XP, caso não estejam acostumados à constante comunicação no decorrer do projeto.

O RUP possui uma estrutura que possibilita ser trabalhado em grandes projetos, devido à sua divisão bem definida de atividades, funções e tarefas junto a uma coleção de artefatos utilizados como produtos de entrada e saída de processos, mas todo esse esforço (custo) pode não ser justificável com uma equipe pequena.

Com suas devidas modificações, o RUP consegue contemplar os processos da XP tornando-os equivalentes em relação ao tempo e esforço, mas lembrando que a essência da XP esta nos quatro valores do manifesto ágil focando na organização, nas pessoas e na cultura. E isso, certamente, é um ponto importante em qualquer projeto, independente de qual metodologia será utilizada.

REFERÊNCIAS BIBLIOGRÁFICAS

AGILCOOP. **Casos de sucesso**. Disponível em: <http://ccsl.ime.usp.br/agilcoop/casos_de_sucesso>. Acessado em: 12 mai. 2012.

BECK, K. **Programação Extrema (XP) Explicada: Acolha as Mudanças**, Bookman, 2004.

BOEHM, B. **COCOMOII**. Disponível em: <http://sunset.usc.edu/csse/research/COCOMOII/cocomo_main.html> Visitado em: 06 mai. 2012.

RATIONAL Unified Process: Best Practices for Software Development Teams. Disponível em: <http://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf>. Acesso em: 05 mai. 2012

FORD Motor Credit Company standardizes development methodology on IBM Rational Unified Process. Disponível em: <ftp://ftp.software.ibm.com/software//rational/web/success/fmc_v2.pdf>. Acessado em: 12 mai. 2012.

PRESSMAN, R.S. **Engenharia de Software**. 6ª Ed, McGraw-Hill, 2006.

SMITH, J. A **Comparison of the IBM Rational Unified Process and eXtreme Programming**. Disponível em: <ftp://ftp.software.ibm.com/software/rational/web/whitepapers/2003/TP167.pdf> Acessado em: 06 mai. 2012

SOMMERVILLE, I. **Engenharia de Software**. 8ª Ed, Pearson Prentice-Hall, 2008.

WELLS, D. Disponível em: <http://www.extremeprogramming.org/> Acessado em: 06 mai. 2012