UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ORLANDO PAMPLONA EING

ADAPTAÇÕES NA METODOLOGIA ÁGIL DE DESENVOLVIMENTO DE SOFTWARE XP (PROGRAMAÇÃO EXTREMA)

DISSERTAÇÃO SUBMETIDA À UNIVERSIDADE FEDERAL DE SANTA CATARINA COMO PARTE DOS REQUISITOS PARA OBTENÇÃO DO GRAU DE MESTRE A CIÊNCIA DA COMPUTAÇÃO.

Orientação: Dr. Rogério Cid Bastos

Florianópolis
Julho, 2003

ADAPTAÇÕES NA METODOLOGIA ÁGIL DE DESENVOLVIMENTO

DE SOFTWARE XP (PROGRAMAÇÃO EXTREMA)

ORLANDO PAMPLONA EING

Esta dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação, área de Concentração Sistemas de Computação e aprovada em sua forma final pelo programa de Pós-Graduação em Ciência da Computação.

Prof. Fernando Álvaro Ostuni Gauthier, Dr.
Coordenador do Programa

BANCA EXAMINADORA

Prof. Fernando Álvaro Ostuni Gauthier Dr.
Universidade Federal de Santa Catarina

Prof. Rogério Cid Bastos, Dr. Eng.
Universidade Federal de Santa Catarina
Orientador

Prof. , Jovelino Falqueto Dr.
Universidade Federal de Santa Catarina

DEDICATÓRIA

À minha esposa e

meus pais que souberam entender a

minha ausência durante o período

de estudos.

AGRADECIMENTOS

Agradecimentos

À Faculdade de Ciências Sociais

Aplicadas de Cascavel - UNIVEL;

À Muni, Mimo, Montie e Mitiu (in memorian),

fiéis camaradas nos momentos mais difíceis;

Ao orientador Prof. Rogério Cid Bastos, Dr. Eng.,

pelo acompanhamento competente e profissionalismo

exemplar no trabalho de orientação;

Ao apoio significativo de meus pais,

sem os quais, seria impossível a realização

desta dissertação;

A todos que contribuíram de forma decisiva na

execução desta pesquisa.

“A adaptação é o dom da excelência do homem.”

(Pitágoras)

RESUMO

EING, Orlando Pamplona. **Adaptações na Metodologia Ágil de Desenvolvimento de Software XP (Extreme Programming)**. 2003. 101f. Florianópolis. Dissertação de Mestrado – Programa de Pós-Graduação em Ciência da Computação, UFSC.

A presente dissertação descreve regras e princípios, bem como fatores favoráveis e desfavoráveis sobre as metodologias ágeis que desenvolveram uma projeção mais acentuada no mercado. Relaciona-se em um comparativo de desempenho e aplicabilidade. Aborda os principais problemas encontrados em um projeto de desenvolvimento de software as metodologias: Programação Extrema (XP), SCRUM, Família Cristal, FDD, DSDM e ASD. Destaca-se a metodologia de programação extrema (XP), presente na maioria das equipes de desenvolvimento. São levantadas as principais adaptações necessárias para atender de maneira satisfatória os pontos críticos nos quais a XP falha ou atende parcialmente.

Palavras Chaves: Desenvolvimento, Adaptação, Qualidade, Tempo, Metodologias Ágeis.

ABSTRACT

EING, Orlando Pamplona. **Adaptations in the Agile Methodology for the Development of XP Software (Extreme Programming).** 2003. 101f. Florianópolis. Master's Degree Thesis – Post Graduation Program in Computer Science, UFSC.

The present essay describes the rules and principles, as well as the favorable and unfavorable factors about the agile methodologies that have developed a more accentuated projection in the market. Comparing them for performance and applicability. It approaches the main problems found in a given software project. The extreme programming methodology (XP) is highlighted, which is present in the majority of development teams. Questions about the necessary adaptations are raised in order to satisfactorily understand the critical points in which XP fails, or partially takes care of.

Keywords: Development, Adaptation, Quality, Time, Agile Methodologies.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIAÇÕES / EXPRESSÕES EM INGLES

ASD - Adaptive Software Development - Desenvolvimento De Software adaptável.

AM – Modelagem Ágil.

AssertTrue – Nome de uma função utilizada em um exemplo

Background – conjunto de objetos comuns compartilhados em testes semelhantes.

Constructor – Método que cria (instancia) um objeto.

Crystal Clear – Uma das metodologias integrantes da família Crystal.

Crystal family of methodologies – Conjunto de metodologias semelhantes

Crystal Web Orange – Uma das metodologias integrantes da família Crystal.

DSDM - Dynamic Systems Development Method – Método de desenvolvimento de sistemas dinâmico. É um framework para desenvolvimento de RAD (Rapid Application Development).

eBusiness - Qualquer forma de negócios ou transação administrativa ou de intercâmbio de informações que é executada utilizando qualquer tecnologia de informação ou comunicação.

eCommerce – Comércio eletrônico através da Internet.

EPG – Guia de processo eletrônico.

Extreme Programming – Metodologia ágil (Programação Extrema)

FDD – Feature Driven Development – Abordagem ágil e adaptável para sistemas em desenvolvimento.

Feedback – Resposta do cliente sobre a utilização do sistema

Fixture – Base conhecida de objetos (relacionado ao Background)

Framework – processo detalhado segundo certas regras.

Fraunhofer – Instituição localizada na Alemanha.

jUnit – Ferramenta para testes de aceitação de código.

MoneyBag – Nome de função utilizada em exemplo.

MoneyTest – Nome de função utilizada em exemplo.

PP – Programação em Pares.

Product Backlog – Lista de funções contendo exigências conhecidas do projeto.

RAD – Rapid Application Development, Projeto de desenvolvimento rápido.

Refactoring – Ato de refazer partes de código de forma a melhorar o desempenho.

Releases – Versão de software funcional que liberada em intervalos durante o projeto.

RunTest – Nome de função utilizada em exemplo.

Scrum – Um processo de desenvolvimento de produtos adaptável, rápido e auto-organizável.

SetUp – Nome de função utilizada em exemplo.

Spearmint – Uma ferramenta de modelagem de processo.

Sprints – São ciclos de repetição, onde a funcionalidade é desenvolvida ou aumentada para produzir novas repetições.

String – Tipo de dado que armazena texto

Suite – Conjunto de vários testes de aceitação.

Superclass – Superclasse, termo integrante da orientação a objetos

TBD (to be determined) – Caraterística de projetos para atender soluções que não possuem um cliente específico.

TearDown – Nome de função utilizada em exemplo.

TestCase – Nome de função utilizada em exemplo.

Timebox – Um ciclo de repetição deve ser realizado dentro de um tempo determinado.

Winifred - relatório de experiência de um projeto adotando as práticas da metodologia crystal orange.

1 INTRODUÇÃO

Problemas e riscos de fracasso são fatores geralmente presentes em projetos de desenvolvimento de software. Conforme o procedimento adotado pela empresa e as decisões tomadas ao longo do projeto, têm-se os sucessos ou os fracassos do mesmo. O conjunto de fatores como: código de baixa qualidade, mudança de requisitos e/ou saída de membros que possuem conhecimento exclusivo sobre os processos elaborados pela equipe de desenvolvimento, são fatores de risco, que podem resultar no fracasso de um projeto de software (BECK, 2000).

A promessa de sanar tais problemas é fornecida com o surgimento das metodologias ágeis, desenvolvidas com uma abordagem diferenciada das tradicionais técnicas de desenvolvimento. Essas trazem soluções que atendem a muitos fatores de risco, considerados problemas, que podem resultar em concluir um projeto ou não. Uma destas metodologias é denominada por programação extrema, ou simplesmente, XP. Essa tecnologia teve início em 1996 através de seu mentor Kent Beck, mostrando-se uma ferramenta para desenvolvimento com respostas rápidas e extremamente flexíveis aos requisitos. Como uma evolução natural da tecnologia, após usar durante algum tempo a XP, a tendência natural da necessidade de uma melhora da ferramenta, para atender a novas situações de problemas identificados ao longo de um projeto acabará surgindo (ASTELS, 2002). Esta pesquisa aborda adaptações a serem realizadas no processo de desenvolvimento XP, buscando uma maior eficiência ou adequando a uma nova condição de negócios.

1.1 Contextualização

As técnicas tradicionais de desenvolvimento de software são consideradas extremamente lentas, tanto para os desenvolvedores, como para o cliente final (ASTELS, 2002). Os altos custos de projetos fracassados por código de má qualidade, mudanças constantes de requisitos, cronogramas não cumpridos e demais fatores de risco, causados por gerentes de projeto de software com más escolhas de técnicas tradicionais, são uma realidade de muitas equipes de desenvolvimento.

Visando extinguir, ou amenizar tais fatores, surge a busca por processos ágeis e flexíveis a situações diversas, dentre os quais, destaca-se o XP (Programação Extrema), processo este, que vem sendo adaptado para atender melhor a certas regras de negócios, dentre as quais, algumas serão abordadas neste trabalho.

1.2 Problema da Pesquisa

Atrasos de cronograma e falhas em projeto de desenvolvimento de software são características não desejadas que, se não sofrerem algum tipo de controle podem levar ao fracasso de um projeto de software. Um dos primeiros obstáculos de grande parte das equipes de desenvolvimento que inicia um projeto é acabar enfrentando algum problema relacionado ao tempo para o desenvolvimento. Um outro fator, é a metodologia de desenvolvimento, que pode acabar prejudicando a evolução do projeto, da forma que foi desejada. Tais fatores já não eram bem vistos há alguns anos, porém, atualmente a competitividade é alta e fracassar em um projeto pode significar não apenas perder um cliente, mas, prejudicar a empresa ao ponto de uma falência.

A evolução das técnicas para desenvolvimento de software, buscando evitar este quadro, surge com as metodologias ágeis, responsáveis por ofertar uma significativa melhora nos processos e evitar os problemas comumente encontrados. Entretanto, apesar de serem uma evolução, as metodologias ágeis não resolvem totalmente as questões impostas pela mudança de requisitos. A programação extrema (XP), sendo a metodologia mais difundida e, portanto, a que atende com maior abrangência, oferecendo soluções viáveis, sofre os mesmos problemas de falhas em determinadas situações de mudança de requisitos. Para adequar a esta situação, adaptações são requeridas de forma a melhorar uma solução já proposta. Com isto se tornará ainda mais produtiva, abrindo assim, uma lacuna que possibilita uma pesquisa científica para documentar a estrutura destas adaptações, bem como a funcionalidade das mesmas, de forma a padronizar soluções para problemas comuns.

1.3 Objetivos

De forma a tornar mais didático a apresentação dos objetivos deste trabalho, os mesmos estão detalhados a seguir.

1.3.1 Objetivo geral

Levantar as principais adaptações referentes ao processo de desenvolvimento de software denominado XP (Programação Extrema), buscando soluções para problemas comuns.

1.3.2 Objetivos específicos

a) Detectar os principais problemas existentes em processos de desenvolvimento de software que utilizem metodologias ágeis;

b) Descrever as principais metodologias ágeis de forma a identificar o campo de abrangência de suas soluções;
c) Sugerir adaptações a serem aplicadas na metodologia da programação extrema.

1.4 Importância do Trabalho

É visível a dificuldade enfrentada por empresas de desenvolvimento de software que, não planejaram uma estrutura de desenvolvimento adequada à realidade imposta pelo mercado. Faz-se necessário, diante da constatação de um mercado com alta concorrência, que as empresas busquem esta estruturação de modo que se possam garantir alguma vantagem competitiva, atendendo a demanda de maneira eficiente e eficaz, cumprindo prazo e validando as soluções propostas. Para isto, é necessárias uma equipe de desenvolvimento, com profissionais treinados e motivados para atender imediatamente a necessidade de cada cliente, no tempo esperado e com custos viáveis. A produtividade nasce essencialmente da inovação, e a competitividade nasce da flexibilidade. A informática e a capacidade cultural de utilizá-la são essenciais para ambas (CASTELLS, 1996).

Considerando a realidade do cenário que está se moldando em torno do setor de TI (Tecnologia da Informação), não há mais espaço para o desenvolvimento mal planejado. A tolerância às falhas de projeto é pequena e tende a diminuir expressivamente, de acordo com o surgimento de novas soluções no mercado. O fato de acabar sofrendo algum fator de risco como: atrasos no cronograma, projetos cancelados devido a esses atrasos, sistemas obsoletos, mudança de requisitos, alta taxa de defeitos e saída de importantes membros da equipe de desenvolvimento que possuem conhecimento exclusivo, precisa ser algo muito distante do cotidiano das equipes de desenvolvimento. Isto não é apenas para os grandes fabricantes de

software, mas, sim para todos que querem tornar-se competitivos fazendo a diferença entre o sucesso e o fracasso.

A empresa que pretende atuar com competitividade dentro de um certo porte empresarial precisa de algum nível de processo (ASTELS, 2002). Estes são importantes para fixar os objetivos e os passos, para atingir os mesmos. O problema é manter a meta fixa durante todo o projeto e é justamente nesta situação que as metodologias ágeis podem auxiliar.

Os métodos ágeis de desenvolvimento têm-se demonstrado ser uma escolha viável para atender aos requisitos impostos pela nova realidade de mercado. Destaca-se entre eles o processo da XP (Extreme Programming), que vem sendo adotado por grandes e pequenas empresas que estão se confrontando com um mercado agitado por oportunidades de exportação, pela competitividade, exigência dos clientes ou o surgimento de novas regras de mercado. Entretanto, como a maioria das metodologias ágeis, a XP não é perfeita e passa por adaptações constantes na busca de uma melhor adequação as novas regras de negócio que devem ser consideradas constantemente. Estas adaptações acarretam em uma melhor produtividade na aplicação do processo. Na maioria das vezes ficam restritas ao conhecimento de poucas equipes de desenvolvimento, geralmente entre as que passaram por aquela necessidade de adaptação para resolver um determinado problema e por falta de interesse, ou regras internas, não procuram difundir tais técnicas entre demais desenvolvedores de software que utilizam o processo da XP. Justificando assim, um levantamento que agrupe de maneira estruturada e científica, estabelecendo claramente as soluções em relação aos problemas, que prejudicam o processo de desenvolvimento de software, sendo, que os mesmos já possuem uma solução funcional, evitando um retrabalho isolado nas equipes.

1.5 Delimitação do Trabalho

Como o assunto se refere às metodologias ágeis, as adaptações foram pesquisadas somente sobre a programação extrema, desconsiderando outras metodologias que não possuem uma documentação acadêmica, bem como comprovação prática documentadas suficientemente para validar sua real aplicabilidade em processos de desenvolvimento de software. A pesquisa foi feita com o levantamento dos problemas mais comumente encontrados não tencionando abranger todas as situações nas quais se fez necessária uma adaptação, pois, muitas situações são específicas de cada equipe e não são classificadas como um problema comum.

As adaptações têm por objetivo, melhorar algumas práticas da XP, entretanto, o escopo da metodologia não será alterado de maneira agressiva, sendo que, as regras impostas para sua utilização continuam ativas, prevalecendo à proposta original da XP, não estipulando que XP em conjunto com as adaptações levantadas se torne à solução para todos os projetos existentes.

1.6 Estrutura do Trabalho

Para uma apresentação mais didática, o trabalho está dividido em 6 partes, organizadas da seguinte forma:

O primeiro capítulo aborda a caracterização do problema, apresentação dos objetivos geral e específico, a limitação e a estrutura da pesquisa.

No capítulo dois, é conceituado o termo de metodologia ágil, apresentando as principais metodologias ágeis existentes, com as respectivas características que englobam cada processo.

No capítulo três é demonstrado os pontos de vantagens e desvantagens que cada metodologia ágil abordada no capítulo dois possui, destacando a programação extrema em um quadro comparativo.

No capítulo quatro são expostos os problemas comumente enfrentados no desenvolvimento de software através da metodologia ágil XP, suas causas e conseqüências, fornecendo o entendimento do problema, bem como, apresentando soluções de adaptação para os problemas levantados. Apresenta-se a forma de obter uma melhor implantação de certas adaptações, buscando técnicas que viabilizem a inclusão das adaptações nas equipes de desenvolvimento.

São demonstrados também, alguns exemplos da utilização de certas adaptações.

No capítulo cinco, são apresentadas conclusões e recomendações explanando os resultados obtidos com a pesquisa científica, bem como, sugerindo pontos não tratados na dissertação que podem ser necessários em uma mudança de requisitos imposta pelo mercado.

1.7 Metodologia

A classificação da pesquisa desenvolvida estabelece as seguintes maneiras de classificação: quanto aos objetivos, quanto à forma de abordagem, quanto à natureza e quanto aos procedimentos adotados. Gil (1996) afirma que, “com relação às pesquisas é usual a classificação com base em seus objetivos gerais, sendo possível classificá-las em três grupos: estudos exploratórios, descritivos e explicativos”.

Quanto aos objetivos o presente trabalho situa-se na categoria de Pesquisa Exploratória, devido ao assunto ser recente e pouco explorado, não obtendo muitas informações sobre as adaptações do processo de desenvolvimento XP (Extreme Programming). Quanto à forma de abordagem, além de ser uma opção do investigador, justifica-se, sobretudo, por ser uma forma adequada para entender a natureza de um fenômeno social, portanto o presente trabalho representa uma pesquisa qualitativa, pois suas características podem descrever a complexidade de determinado problema, analisar a interação de certas variáveis, compreender e classificar processos dinâmicos vividos por grupos sociais, contribuir no processo de mudança de determinado grupo e possibilitar, em maior nível de profundidade, o entendimento das particularidades do comportamento dos indivíduos.

2 FUNDAMENTAÇÃO TEÓRICA

Nos últimos 25 anos, um grande número de diferentes processos para desenvolvimento de software, foi introduzido no mercado, dos quais, poucos obtiveram sucesso e ainda são utilizados atualmente (ABRAHAMSSON, 2002).

O estudo de Nandhakumar e Avison, (1999), discute as tradicionais metodologias de desenvolvimento de um sistema de informação, destacando: “são tratadas principalmente como uma ficção necessária para apresentar uma imagem de controle ou prover um estado simbólico”.

Truex (2000), tem uma posição extrema afirmando que possivelmente, os métodos tradicionais são “ideais meramente inacessíveis e hipotéticos, um espantalho” que provê orientação normativa a situações de desenvolvimento utópicas”.

Como resultado, a indústria de desenvolvimento de software tende a adquirir um ceticismo quanto a novas soluções que são difíceis de entender e desta forma passam a não serem utilizadas (WIEGERS, 1998). Este é a base para o aparecimento dos métodos ágeis de desenvolvimento de software.

A Modelagem Ágil (AM), é uma metodologia baseada na prática para o modelo efetivo e a documentação dos sistemas baseados em software (ASTELS, 2002).

Segundo Scott Ambler (2002b):

> Modelagem ágil (AM) é uma metodologia baseada na prática para modelagem efetiva de sistemas baseados em software. A metodologia AM é uma coleção de práticas, guiadas por princípios e valores que podem ser aplicados por profissionais de software no dia a dia. AM não é um processo prescritivo, ela não define procedimentos detalhados de como criar um dado tipo de modelo, ao invés ela provê conselhos de como ser efetivo como modelador. Pense em AM como uma arte, não como uma ciência.

Um modelo ágil é um modelo bom o suficiente, nada mais, o que implica que ele exibe as seguintes características (AMBLER, 2002C):

1. Ele atende seu propósito;
2. Ele é inteligível;
3. Ele é suficientemente preciso;
4. Ele é suficientemente consistente;

5. Ele é suficientemente detalhado;
6. Ele provê um valor positivo;
7. Ele é tão simples quanto possível.

Segundo Astels (2002):

> Pode-se dizer que AM é uma atitude, não um processo prescritivo. AM é um suplemento aos métodos existentes, ele não é uma metodologia completa, é uma forma efetiva de se trabalhar em conjunto para atingir as necessidades das partes interessadas no projeto. AM é efetivo e é sobre ser efetivo, é algo que funciona na prática, não é teoria acadêmica. AM não é uma bala de prata. É para o desenvolvedor médio mas não é um substituto de pessoas competentes. AM não é um ataque à documentação, pelo contrário, AM aconselha a criação de documentos que têm valor. Não se caracterizando como um ataque às ferramentas ***CASE*** e principalmente, AM não é para todos.

Em resumo, a AM reúne uma série de valores e ações para modelagem de maneira rápida e de certa forma fácil, entretanto, não deve ser considerada menos trabalhosa que modelos tradicionais, ou seja, AM não quer dizer menos modelo (AMBLER, 2002A).

Apesar do crescente interesse no assunto, nenhum acordo claro foi alcançado em como distinguir o desenvolvimento de software ágil das mais tradicionais aproximações. Os limites, se é que existem, ainda não foram claramente estabelecidos, apesar disto, mostrou-se que certos métodos necessariamente não são satisfatórios para todos os indivíduos (NAUR, 1993) ou situações (BASKERVILLE, 1992). Por esta razão, ressalta-se o desenvolvimento de um processo pessoal para cada desenvolvedor de software (HUMPHREY, 1995). Mesmo considerando tais colocações dos autores, um estudo que foque a real aplicabilidade de métodos ágeis em relação a outros, de maneira científica, ainda não foi realizado. Em conseqüência disto, equipes de desenvolvimento acabam por ter uma certa dificuldade na procura de algum material que possa fornecer o embasamento correto para uma definição de que metodologia utilizar.

No período de novembro de 2002, até janeiro de 2003, uma empresa australiana denominada Shine Technologies (www.shinetech.com) fez uma pesquisa on-line sobre o uso de métodos ágeis. As principais conclusões são:

XP é o método, mais popular, usado por 59% dos participantes; em segundo vem SCRUM, com FDD em terceiro, conforme o gráfico que se segue:

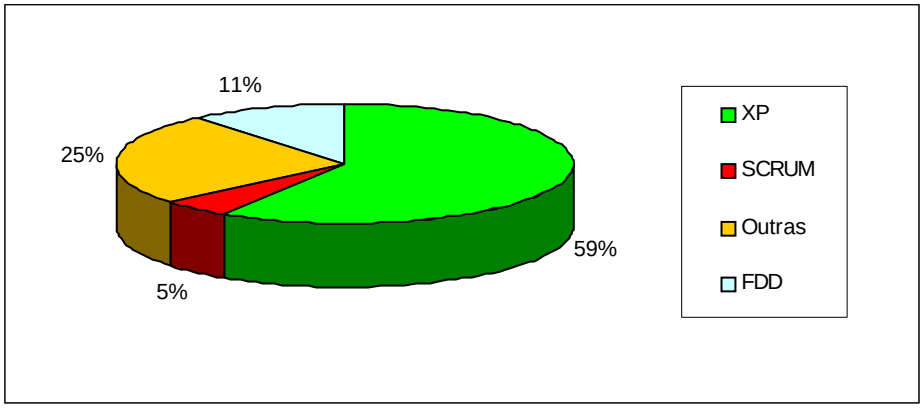

Figura 01 Metodologia ágil por preferência
Fonte: Shine Technologies

Após adotarem, um método ágil, 93% dos entrevistados responderam que tiveram aumento de produtividade, 88% afirmaram que a qualidade do software aumentou, 95% disseram que os custos de desenvolvimento caíram ou se mantiveram os mesmos e 83% garantiram que a satisfação dos usuários aumentou.

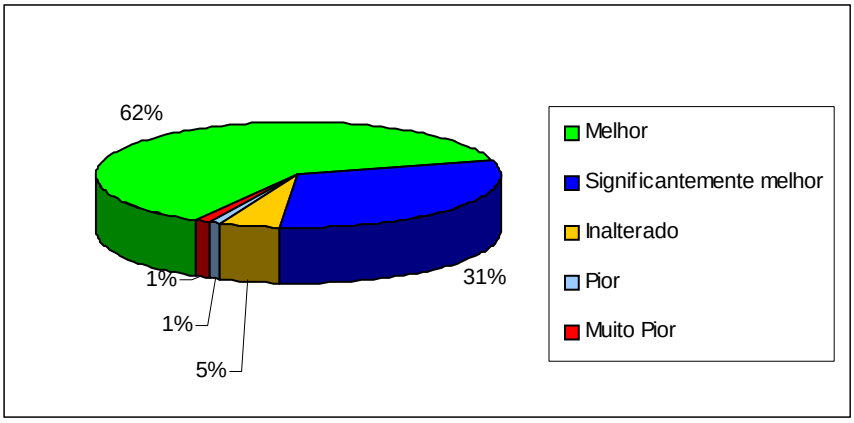

Figura 02 Aumento de produtividade
Fonte: Shine Technologies

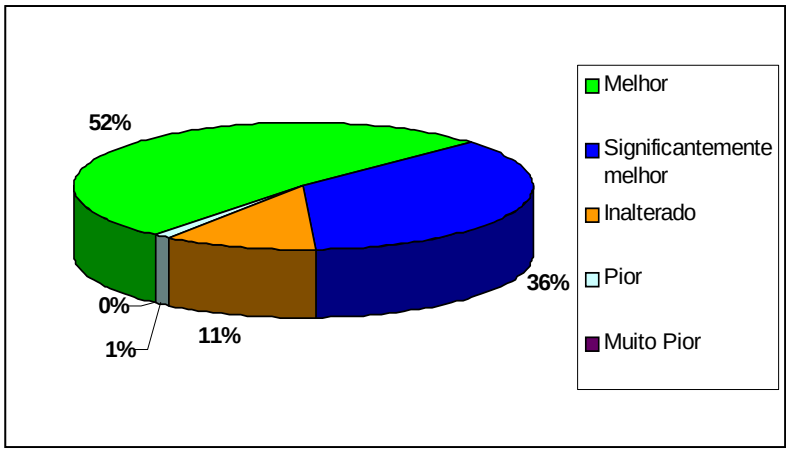

Figura 03 Qualidade do software gerado.
Fonte: Shine Technologies

Quanto aos pontos positivos, a maioria considerou, em primeiro lugar, "responder à mudança em vez de seguir um plano", e em segundo, "as pessoas acima dos processos"; Os pontos negativos foram "pouca documentação" e "falta de estrutura de projeto";

Quando questionados sobre a possibilidade de adotarem os métodos ágeis no ano de 2003, 94,7% dos respondentes pretendem continuar usando e apenas 16% dos respondentes acreditam que os métodos ágeis devam ser usados em todos os projetos, enquanto 50% acreditam que devam ser usados na maioria dos projetos.

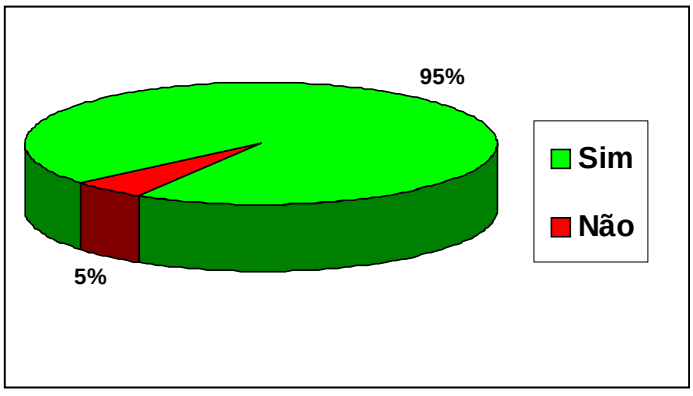

Figura 04 Sim ou não para a pergunta: Utilizará métodos ágeis em 2003?
Fonte: Shine Technologies

As práticas da modelagem ágil são englobadas pelo processo da XP além de suas práticas particulares. São elas: a comunicação, a simplicidade, o feedback, a humildade, a honestidade e principalmente a coragem (AMBLER, 2002a). Estas

práticas serão abordadas de uma maneira mais abrangente dentro das práticas da XP em parágrafos seguintes.

Os valores adotados pela AM são os desenvolvidos pela ***Agile Alliance*** (2001), sendo: os indivíduos e as interações em vez dos processos e das ferramentas; o software de trabalho em vez da documentação abrangente; a colaboração com o cliente em vez da negociação de contrato e finalmente, responder à mudança em vez de seguir um plano (ASTELS, 2002). Também foram definidos no manifesto ágil, nos eventos realizados de 11 a 13 de fevereiro de 2001, através de um seleto grupo de pesquisadores, destacando-se: Kent Beck, Ward Cunningham, Martin Fowler, Ron Jeffries e Robert C. Martin, os 12 princípios, que são considerados diretrizes gerais para um projeto de sucesso. Na seqüência tem-se a descrição de cada um dos 12 princípios (ALVIM, 2002):

Tabela 01 Princípios da Metodologia Ágil

Princípio	**Descrição**
Princípio 1	A mais alta prioridade é a satisfação do cliente através da liberação mais rápida e contínua de software de valor.
Princípio 2	Receba bem as mudanças de requerimentos, mesmo em estágios tardios do desenvolvimento. Processos ágeis devem admitir mudanças que trazem vantagens competitivas para o cliente.
Princípio 3	Libere software com a freqüência de duas semanas até dois meses, com preferência para a escala de tempo mais curta.
Princípio 4	Mantenha as pessoas dos negócios e os desenvolvedores trabalhando juntos a maior parte do tempo do projeto.
Princípio 5	Construa projetos com indivíduos motivados, dê a eles o ambiente e suporte que precisam e confie neles para ter o trabalho realizado.
Princípio 6	O método mais eficiente e efetivo de repassar informação entre uma equipe de desenvolvimento é através de conversação cara-a-cara.
Princípio 7	Software funcionando é a principal medida de progresso.
Princípio 8	Processos ágeis promovem desenvolvimento sustentado. Os patrocinadores, desenvolvedores e usuários devem ser capazes de manter conversação pacífica indefinidamente.
Princípio 9	A atenção contínua para a excelência técnica e um bom projeto aprimora a agilidade.
Princípio 10	Simplicidade, a arte de maximizar a quantidade de codificação ser realizada é essencial.
Princípio 11	As melhores arquiteturas, requerimentos e projetos emergem de equipes auto-organizadas.
Princípio 12	Em intervalos regulares, as equipes devem refletir sobre como se tornarem mais efetivas, e então refinarem e ajustarem seu comportamento de acordo.

Fonte: Agile Alliance

A melhor maneira ou ferramenta para aplicar a modelagem ágil, o melhor

processo para desenhar os diagramas é o lápis e o papel, ou o marcador e o quadro, de forma que agiliza o processo e possibilita o simples apagar, ou jogar fora ao término do processo (ASTELS, 2002).

2.1 Métodos ágeis existentes

Os métodos que são considerados nesta análise: ***Extreme Programming*** (BECK, 1999b), SCRUM (SCHWABER, 1995; SCHWABER and BEEDLE, 2002), ***Crystal family of methodologies*** (COCKBURN, 2002a), ***Feature Driven Development*** (PALMER and FELSING 2002), ***Dynamic Systems Development Method*** (Stapleton, 1997) e ***Adaptive Software Development*** (HIGHSMITH, 2000). Outros métodos foram desconsiderados por serem ainda muito recentes e não possuírem documentação suficiente de forma a permitir um referencial teórico satisfatório, bem como, não foram aplicados em nenhuma equipe como um trabalho acadêmico para uma avaliação prática do mesmo.

2.2 Programação Extrema

Focando novamente no processo de desenvolvimento de software utilizando a XP, o mesmo desperta interesse ao ouvir falar pela primeira vez, pois, visa atingir fatores de extrema importância na área de TI (Tecnologia da Informação), sendo: obter uma resposta rápida do desenvolvimento de forma a fornecer de maneira constante soluções para o cliente e ser flexível a ponto de modificar-se de acordo com as mudanças da regra de negócios durante o desenvolvimento do projeto. Para isto, o processo utiliza uma série de regras que serão vistos em um ciclo de produção pré-estabelecido, como apresentado na Figura 05.

Divide-se basicamente em 6 fases: Fase de exploração; Fase de planejamento; Repetições para a Fase de release; Fase de produção, Fase de manutenção e Fase final. Cada uma destas fases possui práticas distintas que devem ser executadas de forma a validar a passagem para a próxima fase. Em alguns momentos, como na Fase de repetições, existe uma espécie de “loop”, no qual o processo é executado até atingir um certo nível de qualidade e aplicabilidade na produção.

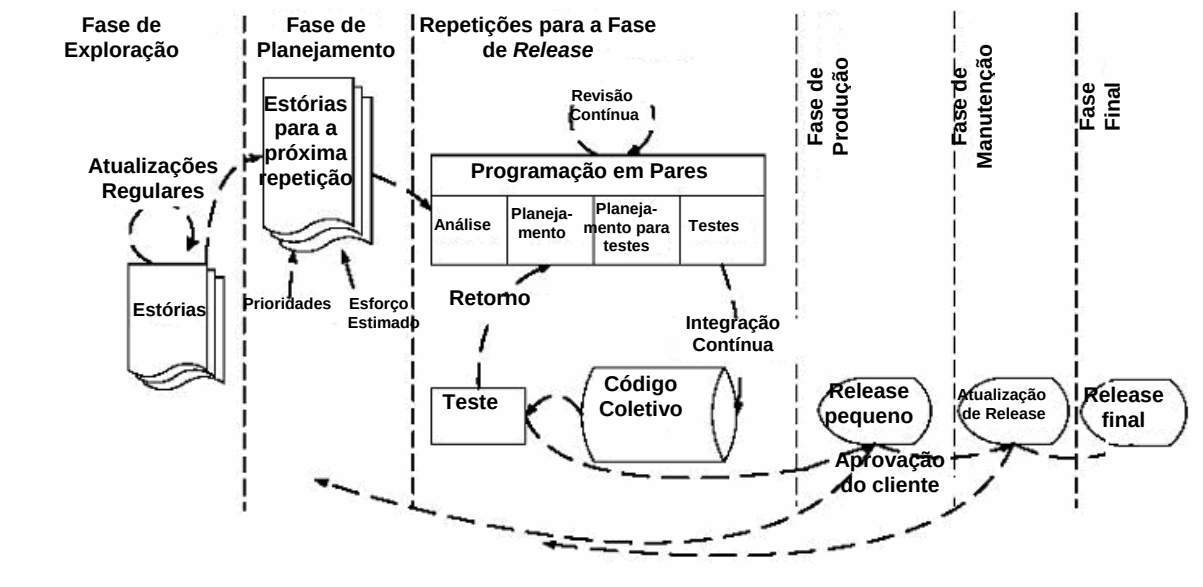

Figura 05 Ciclo de vida do processo XP
Fonte: Abrahamsson,2002

Apesar do nome e do conceito que alguns críticos possuem sobre XP, não se trata de uma forma desorganizada e sem projeto nenhum de desenvolvimento, ou seja, a idéia de sentar em frente à máquina e começar a gerar código de maneira inconseqüente não é verdadeira, pelo contrário, XP adota uma abordagem disciplinada e um sistema de projeto contínuo. Forma uma série de pequenos projetos durante todo o tempo, dedicando-se a uma tarefa de cada vez (ASTELS, 2002).

O jogo da XP é um processo no qual as atividades não são mais medidas em dias ou meses, mas sim em segundos, minutos e horas (ASTELS, 2002).

Para integrar-se ao jogo os participantes devem obedecer algumas regras e práticas, bem como seguir os princípios base da XP, entretanto, o final do jogo com a vitória, depende em muito de seus jogadores. Pode-se dizer que tudo termina bem se o produto final foi entregue atendendo a solução requisitada e a equipe atingiu a solução de uma maneira natural e de certa forma, o desenvolvimento transcorreu de maneira harmônica durante todo o processo.

Aborda-se na seqüência, os princípios que devem ser obedecidos dentro do processo de desenvolvimento de software denominado programação extrema, ou simplesmente XP.

2.2.1 Trabalhar com o Cliente

Talvez o mais extremo dos princípios da XP seja a sua insistência em fazer um cliente real trabalhar diretamente no projeto (ASTELS, 2002).

No jogo da XP existem duas equipes**,** a do cliente e a do desenvolvimento. A participação do cliente constantemente junto aos desenvolvedores é muito importante, pois, sem um cliente não existe motivo para se ter um sistema. A equipe do cliente desempenha um papel fundamental no processo da XP; os clientes não devem ser vistos como simples usuários do sistema e sim, como uma função muito mais ampla que isto, pois simples usuários não serão capazes de interagir da maneira correta.

Existem subdivisões de papéis na equipe do cliente. Podem ser os “contadores de histórias”, “os aceitantes”, “os donos do ouro” ou “os planejadores”. A divisão de quem fará qual função deve ser efetuada levando-se em consideração as habilidades específicas que cada uma das pessoas possui, observando-se uma melhor aptidão para representar um ou outro papel. Algo importante é não visualizar a situação como se fosse uma hierarquia de papéis, ou seja, não devem existir disputas para uma melhor ou pior função dentro do grupo, pois, todos trabalham para um objetivo comum e os papéis são fundamentais, independentemente da ação exercida por eles.

Segue-se uma descrição dos papéis desempenhados pela equipe do cliente:

2.2.1.1 Os contadores de histórias

São aquelas pessoas da equipe do cliente que tem a especialização no domínio, ou seja, que possuem o conhecimento dos requisitos que o sistema deverá cumprir. São as pessoas que irão orientar de forma que o ***release*** (versão do sistema que é liberada em tempos distintos) liberado atenda as necessidades de maneira satisfatória.

2.2.1.2 Os aceitantes

Podem ser também contadores de histórias ou ser um grupo à parte, entretanto, caso seja uma equipe distinta, isto não deve excluir a equipe dos contadores de história da responsabilidade de saber o que há em cada release em resumo, os aceitantes garantem que cada release tenha exatamente a solução para

as histórias dos contadores de história.

2.2.1.3 Os proprietários do ouro

São as pessoas que trabalham com o lado comercial do sistema; são eles que estipulam o valor comercial do sistema, bem como, são responsáveis pelos recursos que envolvem valor financeiro como o equipamento a ser utilizado e o financiamento do projeto em si.

2.2.1.4 Os planejadores

Possuem a responsabilidade de adequar da melhor maneira possível as distribuições dos ***releases***, de forma que se for necessário uma interrupção, que a mesma seja o mais breve possível. Também estudam o ciclo de negócios envolvido para estruturar a distribuição com os usuários que possuem maior envolvimento com as mudanças incluídas ou novas funcionalidades adicionadas.

2.2.1.5 O chefão

É a pessoa responsável por todo o sistema (BECK, 2000). Seu trabalho se desenvolve junto à equipe do cliente, bem como com a equipe de desenvolvimento. É o responsável para resolver qualquer problema que os caminhos organizacionais acabem impondo.

Um problema em projetos utilizando a XP é a de não existência da equipe cliente, casos comuns em projetos denominados TBD (***To Be Determined***), que são projetos para atender soluções que não possuem um cliente específico. Nesta situação sempre deve existir alguém com o conhecimento do domínio para realizar o papel do cliente e gerar as histórias, podendo ser inclusive, alguém da própria equipe de desenvolvimento. O risco maior nesta situação é: após a conclusão do projeto não existirem clientes interessados no sistema. Um fator importante é não utilizar a XP desta maneira enquanto existir a possibilidade de uma equipe real de clientes, pois, a produtividade é muito maior seguindo-se os passos do processo de maneira correta.

No que diz respeito à equipe de desenvolvimento, é responsável por estimar as tarefas, ajudar os clientes a8 ver as conseqüências de suas decisões, adaptar os seus processos de desenvolvimento e entregar o sistema (ASTELS, 2002).

Criar uma equipe de desenvolvimento é o processo de transformar um grupo de desenvolvedores com interesses diferentes e com experiência anterior em uma equipe integrada e efetiva em uma única tarefa (SHARIFABDI, 2002).

Da mesma forma que a equipe do cliente, a equipe dos desenvolvedores também possui papéis distintos a serem assumidos de acordo com a aptidão de cada desenvolvedor da equipe, os quais serão descritos na seqüência:

2.2.1.6 O técnico

Segundo Kent Beck (2000), traz a experiência para a mesa da equipe de desenvolvimento em XP e é responsável por garantir que o processo se desenrole normalmente. O papel do técnico é fundamental dentro da equipe de desenvolvimento, age como a pessoa de apoio e, por ter maior conhecimento que os demais, indica em muitas situações o caminho que será tomado.

2.2.1.7 O acompanhador

É responsável pelo processo de feedback das estimativas do sistema (BECK, 2000). Juntamente com o seu equivalente na equipe do cliente, o planejador, é responsável pela averiguação do tempo de desenvolvimento e alterações de programação.

O acompanhador não deve passar a imagem negativa de um mero avaliador sem conhecimento da situação, é importante que seu papel seja de uma pessoa sensata e confiável, e que a equipe não o veja como um gerenciador e sim como parte da equipe, sempre em busca de uma melhor produtividade.

2.2.1.8 O facilitador

É a pessoa responsável por estabelecer a melhor comunicação possível entre a equipe de desenvolvimento e a equipe do cliente. Possui um papel importante nas reuniões, garantindo que a compreensão das histórias contadas seja assimilada corretamente e que o grau de satisfação pessoal indique que cada elemento participante fez o máximo possível para a produtividade comum. Também é encarregado de resolver os conflitos que possam ocorrer entre as duas equipes.

2.2.1.9 O arquiteto

Possui duas funções dentro da equipe, inicialmente é responsável por escrever o menor número possível de testes requeridos para dar suporte ao que é necessário em uma determinada repetição (ciclo de desenvolvimento). Em segundo lugar, age como um observador de código, de forma a minimizar o ***refactoring***. O arquiteto não executa a otimização do código, e sim, detecta códigos de difícil observação por desenvolvedores normais, que devem ser reescritos para uma melhora significativa; o ***refactoring*** não é a sua função.

2.2.2 Uso de Metáforas para Descrição de Conceitos

Uma metáfora é uma forma poderosa de relacionar uma idéia difícil em uma área desconhecida (ASTELS, 2002). A metáfora tem a função de tornar o conhecimento do domínio de maneira comum a todos os integrantes das equipes, utilizando para isto uma representação simples, utilizando um exemplo geralmente da realidade do cliente para descrever uma determinada situação no sistema. Deste modo, o cliente tem a capacidade de entender a situação e saber se a resposta do sistema, ou seja, a solução atende determinado requisito.

Podem ser utilizadas tanto pela equipe do cliente, bem como pelos desenvolvedores. Quando utilizado pela equipe de desenvolvimento podem descrever uma condição da organização, o projeto ou o código ou os algoritmos e padrões. Para se ter uma idéia de metáfora, Martin Fowler (1999), classifica uma condição de código mal escrito como “código que cheira”.

Já utilizada pela equipe do cliente, pode representar a conceitualização de uma determinada prática do sistema, no qual o cliente não tenha o conhecimento técnico para descrever com uma linguagem de desenvolvedor; utiliza, portanto, uma metáfora como o exemplo: “O sistema de software que estou precisando deve funcionar como uma consulta à lista telefônica, de forma que com o nome eu possa achar o telefone relacionando à pessoa”.

As metáforas são extremamente úteis para a comunicação e troca de idéias dentro de uma equipe de desenvolvimento (ASTELS, 2002).

2.2.3 Planejamento

Os projetos de desenvolvimento de software de qualquer tamanho

significativo precisam ser planejados. Um certo grau de planejamento é requerido pelo processo da XP; o diferencial está exatamente em quanto tempo a equipe deve dedicar-se ao planejamento. O que comumente acontece em projetos com uma carga excessiva de planejamento é de que ao atingirem um determinado estágio de desenvolvimento os requisitos acabam mudando, conseqüentemente, o planejamento efetuado anteriormente de nada adiantou.

Deve ser estabelecido por pessoas que tenham o conhecimento necessário das prioridades e requisitos do sistema, de forma a estabelecer um escopo plausível para cada liberação de software (***release***).

O planejamento é dividido em duas partes, ficando uma parte a cargo da equipe de desenvolvimento e outra parte para a equipe do cliente, cada uma com decisões específicas , sendo (BECK, 2000):

1) Decisões do pessoal de negócios, ou equipe do cliente: O escopo, definindo o que o sistema precisa fazer. A prioridade, para elaborar o escopo de forma que não se torne muito abrangente, decidindo assim, o que pode ser excluído ou retardado. Datas das liberações, decisão sobre quando serão empregados para utilização, bem como realizar este processo.
2) No que diz respeito às decisões do pessoal técnico ou equipe de desenvolvimento têm-se:

a) As estimativas, o cálculo realizado para ter uma previsão do tempo de implementação de um determinado código.
b) As conseqüências verificam como uma forma de alerta para a equipe do cliente os resultados de determinadas decisões, como: ferramentas a ser utilizado, hardware, a existência ou não de subsistema entre outras coisas.
c) O processo engloba o todo, são decisões que consideram fatores de como será o desenvolvimento do sistema em relação a recursos gerais, bem como outros itens que digam respeito à equipe.
d) A programação detalhada estima quando determinado código será desenvolvido dentro de um ***release***. Geralmente segue uma ordem de menor para maior risco, para minimizar assim, o risco geral do sistema.

3) O planejamento deve ser elaborado de maneira muito bem estruturada, de forma que não são aceitáveis promessas de liberação de software,

com tempo de desenvolvimento extremamente pequeno, sejam apenas para agradar o cliente e que podem acarretar em desapontamento se o prazo estipulado não for devidamente obedecido.

2.2.4 Reuniões Curtas

Devem ser realizadas com o objetivo focado na comunicação. Não devem ser utilizadas para discussões ou detalhamentos; geralmente são realizadas no início de cada manhã. Cada desenvolvedor descreve o que fez no dia anterior e o que pretende fazer durante o dia de produção que está se iniciando.

Caso haja a necessidade de uma discussão, ela deve ser realizada em particular, somente com as partes envolvidas e não durante uma reunião padrão.

2.2.5 Testes

Os testes são a garantia de que o código desenvolvido não apresenta falhas na resolução de um determinado problema. Quando todos os testes são realizados e obtém-se um resultado positivo, tem-se a certeza de que o código produzido está atendendo os requisitos e desta forma, pode-se iniciar o desenvolvimento de outro código, incorporando o código já testado e aprovado ao ***release***.

Os testes são executados em vários níveis dentro do processo, um dos níveis é o de unidade, que possui a função de avaliar uma pequena parte de código. Geralmente existe um grande número de testes de unidade para validar uma tarefa.

Os testes de unidade devem ser escritos antes do código destinado a ser testado pelo mesmo, formando assim um ciclo com a seqüência: teste, código, teste, código e assim por diante. O conteúdo dos testes, ou seja, a função de testes que executam deve considerar o seguinte: “teste tudo o que possa quebrar” (ASTELS, 2002). Assim sendo, o pacote de teste criado deve contemplar todas as situações possíveis de erros, de forma que se os testes forem executados retornando 100% de aprovação, têm-se a garantia de que o código criado está totalmente pronto. Ao fato dos testes serem escritos antes de código, pode-se adicionar a vantagem de que o desenvolvedor não irá escrever mais código do que o necessário para satisfazer as condições dos testes, evitando assim, que o foco de uma determinada tarefa não fique abrangente a outras situações.

Quanto à execução dos testes, além de executá-los na forma de testes de

unidade, devem ser executados constantemente durante o projeto, de maneira a garantir que nenhum código seja alterado e conseqüentemente danificado quanto à sua função inicial.

2.2.6 Simplicidade

Esta prática atende ao fato de projetar e codificar o que realmente é necessário em um certo momento. Vai justamente contra a prática tradicional de um projeto generalizado, ou seja, projetar todo o sistema para só então iniciar a codificação. O risco de utilizar a prática de um projeto generalizado e no momento que atingir uma determinada codificação descobrir que não há mais motivo para realizá-la, pois, mudaram os requisitos, é eliminado com a prática da simplicidade.

Pode-se classificar um projeto simples com as características em ordem de importância: executar todos os testes, revelar todas as intenções, não envolver duplicação e conter o menor número de classes ou métodos (BECK, 2000).

Uma das vantagens de manter um projeto simples é ganhar tempo de codificação, de forma que se implementa apenas o que realmente é necessário, não perdendo tempo com requisitos que poderão ser necessários em um futuro próximo. Se realmente for constatado algum requisito futuro, ele será codificado no momento em que se mostrar imprescindível.

Na dúvida é melhor não codificar e deixar para acertos finais no momento do ***refactoring*** (processo de alterar um sistema de software) para manter uma simplicidade no projeto é interessante não ficar muito preso a ele (NEWKIRK, 2001).

2.2.7 Programação em pares

Trata-se de uma das práticas mais interessantes do processo da XP, muitas questões surgem em torno da programação em pares e será mais bem descrita em suas adaptações no capítulo 3. A programação em pares busca juntar duas cabeças pensantes em um mesmo problema, de forma que o desenvolvimento de código seja realizado por um par de desenvolvedores, conforme demonstra a Figura 06.

Se o parceiro tiver alguma idéia para implementar determinado código, nada impede que os papéis sejam trocados. Geralmente utiliza-se algo como: “Faça você um pouco” e “Posso codificar?”, ou algo semelhante para a realizar a inversão, isto depende muito do par que está codificando e do relacionamento estabelecido entre

a dupla.

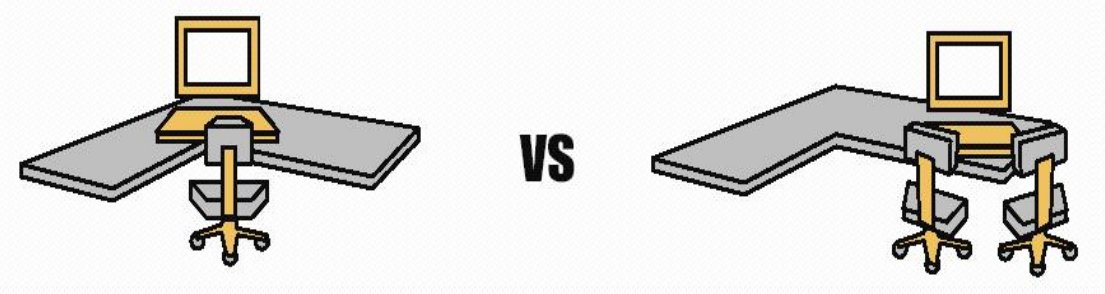

Figura 06 Par de programadores
Fonte: Willians, 2000

A técnica de programação em pares utiliza-se de alguns agentes facilitadores para obter sucesso. Dentre estes agentes um dos mais significativos é o ambiente físico de trabalho. Segundo 96% dos programadores, o layout do espaço correto é um fator crítico na importância de sucesso de um projeto (WILLIANS, 2000).

O ambiente precisa disponibilizar aos desenvolvedores um certo conforto, deve-se ficar sentados lado-a-lado e ambos ter acesso ao teclado e ao mouse, com a visualização perfeita do monitor. A sala deve ser ampla o suficiente para conter todos os pares de programadores e não deve existir cubículos que isolem as duplas pois, muitas vezes uma dupla escuta determinada conversa da dupla ao lado, fato que pode contribuir em determinadas tarefas (WILLIANS, 2000).

A escolha de um parceiro geralmente é realizada logo após a curta reunião matinal. Os desenvolvedores trocam informações, sugerindo um determinado interesse em fazer algum código específico, as pessoas que tiverem um interesse comum, formam um par e vão para uma estação de trabalho. Os pares não devem ser sempre os mesmos; um revezamento é interessante para aproximar todo o grupo, de forma a não estabelecer uma dupla que queira sempre estar junta.

O processo ocorre naturalmente, alternando as posições de piloto (o desenvolvedor que tem controle do mouse e teclado) e do co-piloto (o desenvolvedor que auxilia o piloto, ou navegador). O código escrito pelo piloto é testado mentalmente pelo navegador, que sugere alterações quando achar necessário. O profissionalismo da dupla é fundamental, de forma que ambos trabalhem dentro de um padrão e que sugestões jamais se tornem discussões.

Desta forma o código é concluído e a integração é realizada, ficando a dupla livre para formar pares com outros desenvolvedores e iniciar o processo novamente.

2.2.8 Codificação dentro de padrões

Um padrão de desenvolvimento é altamente requerido dentro da XP: indiferente do padrão adotado é necessário que todos os membros da equipe o adotem. Caso algum membro discorde de algo poderá sugerir ao grupo que todos mudem determinado ponto dentro do padrão, mas jamais codificar de maneira diferente sem comunicação prévia ao grupo e com o devido consentimento de todos.

Existem padrões diversos, por exemplo, se a equipe estiver desenvolvendo em Java, o padrão da Sun Microsystem (1999) é uma boa opção.

Algumas práticas podem ser citadas, segundo David Astels (2002):

> Você pode programar aos pares porque ambos os parceiros trabalham no mesmo formato. Todos podem ter a propriedade do código porque todos escreveram o código com o mesmo estilo. Você pode fazer o refactoring porque entende o formato do código que está alterando e os outros entenderão o formato do seu trabalho. Finalmente, você pode trabalhar com velocidade total porque não precisa reformatar o código durante o trabalho, nem precisa estar constantemente alternando de um estilo para outro, ou tentando entender a formatação e o estilo, bem como o próprio código.

2.2.9 Propriedade coletiva de código

Quando se faz uma divisão de funções, estabelecendo determinadas classes para um ou outro desenvolvedor, forma-se uma situação improdutiva, pois, se alguém precisar que uma determinada alteração seja realizada nesta classe, deverá requisitar para o proprietário da mesma a alteração, ou adotar uma outra solução.

O desenvolvedor que requer a alteração terá que esperar que o proprietário da classe tenha tempo para realizar a mudança de código. Isto pode levar muito tempo e atrasar o processo, ou poderá ele mesmo realizar as alterações em uma cópia da classe, descobrir que a alteração tem problemas e alterar o código novamente para o correto. O proprietário da classe poderá ignorar o conhecimento deste fato e realizar a alteração solicitada da maneira incorreta, isto acarretará duas classes diferentes e o projeto passa a ter seu nível de risco elevado.

Para evitar este tipo de acontecimento em XP, todo o código é de propriedade coletiva, de forma que o desenvolvedor que sentir a necessidade de alteração, tem todo o direito de executá-la, independentemente de quem tenha escrito o código em questão.

Para auxiliar no processo e tornar uma alteração confiável, uma ferramenta

de gerenciamento de projetos é requerida, ela deve disponibilizar acesso direto a qualquer código do projeto para todos os membros da equipe, fornecendo assim, um controle de fontes seguro e funcional.

2.2.10 Integração contínua

O processo de integração nada mais é do que adicionar ao projeto o código gerado e testado. Após a dupla criar um determinado código e o testar exaustivamente, de forma a minimizar o máximo possível a possibilidade de conflitos com algum código já existente no projeto, a tarefa é considerada concluída. O código final então é integrado ao projeto maior e testado novamente para verificar se a integração alterou algo que já estava funcionado.

A integração deve ser realizada pelo menos uma vez ao dia, porém, é aconselhável que a cada tarefa concluída seja realizada a integração. Após um certo número de integrações que expandem as funcionalidades do projeto, obtêm-se um novo ***release*** (versão com novas funções) e o mesmo pode ser distribuído para o cliente.

2.2.11 Refactoring

De acordo com Martin Fowler (1999):

> ***Refactoring*** é o processo de alterar um sistema de software de tal forma que ele não altere o comportamento externo do código e melhore a sua estrutura interna. Essa é uma forma disciplinada de limpar o código que minimiza as chances de introdução de “bugs”.

O ***refactoring*** é realizado constantemente: durante a programação em pares o co-piloto muitas vezes sugere ***refactoring*** no código que acabou de ser criado. Pode também ser realizado após o código estar pronto e devidamente testado com a finalidade de melhorar alguma funcionalidade do mesmo.

O ***refactoring*** pode ser pequeno, alterando-se poucas linhas de código, ou, abrangente, mudando a lógica de uma determinada tarefa para otimizar a mesma e aumentar a produtividade.

Ron Jeffries (2001), declara o seguinte sobre o ***refactoring***:

> Existem dois momentos em que se considera o ***refactoring*** como parte da execução de uma tarefa: antes de a tarefa começar, para “criar espaço”

> para o código novo, e após a tarefa ser implementada, para tornar o código novo e antigo o mais claros e simples possível.

2.2.12 ***Releases*** pequenos

O intervalo entre a distribuição dos ***releases*** deve ser o menor possível. Geralmente a faixa de tempo fica em torno de um a seis meses. Seis meses é um tempo indesejável, por ser muito longo. Um dos fatores beneficiados com ***releases*** pequenos é o ***feedback*** (parecer sobre a utilização do código) do cliente, de forma que se um caminho errado estiver sendo percorrido, a sua detecção será a mais rápida possível.

Entretanto, as tarefas integradas a cada ***release*** devem estar sempre completas. A liberação de um ***release*** com determinadas funções incompletas não propicia benefício para nenhuma das equipes.

2.2.13 Trabalho de 40 horas por semana

Um dos problemas enfrentados pelos desenvolvedores é o estresse constante da pressão para o desenvolvimento de determinados projetos. O cansaço absorvido por noites de codificação em busca de cumprir cronogramas atinge muitas equipes de desenvolvimento (ASTELS, 2002).

No processo de desenvolvimento utilizando a XP isto não acontece, o número de horas semanais de trabalho não pode ultrapassar de 40, sendo, 8 horas diárias de fundamental valor para uma boa produtividade. Uma quantidade razoável de descontração durante o trabalho também ajuda a evitar que a equipe tenha estafa.

O planejamento tende a evitar a necessidade de um número maior de horas trabalhadas, evitando assim, a pressão geralmente presente em técnicas tradicionais de desenvolvimento.

2.2.14 Alterações

As alterações de requisitos, geralmente mal vistas em projetos, na XP são totalmente viáveis e funcionais.

Projeto flexível é uma das características que torna a XP tão atraente. No mundo real, a alteração acontece continuamente. Não há como o cliente estabelecer seus requisitos no início do projeto e manter os mesmos até o término do mesmo.

Diante de uma instabilidade de mercado, as regras de negócio tendem a mudar de acordo com as necessidades que surgem a cada dia. Desta forma, o sistema deve contemplar as situações conforme surgem no decorrer do projeto.

Segundo Astels (2002), “mantendo o código limpo e simples, você garante que o código seja flexível e permite que ele se dobre quando for confrontado com a mudança nos requisitos”.

Pode-se definir o conjunto das práticas citadas como um ciclo repetitivo em alguns momentos que, caso seguido de maneira fiel, implicará na utilização do processo da XP de maneira exata como definido inicialmente, ou seja, sem adaptações, criando-se um ciclo conforme observado a estrutura na Figura 07 a seguir:

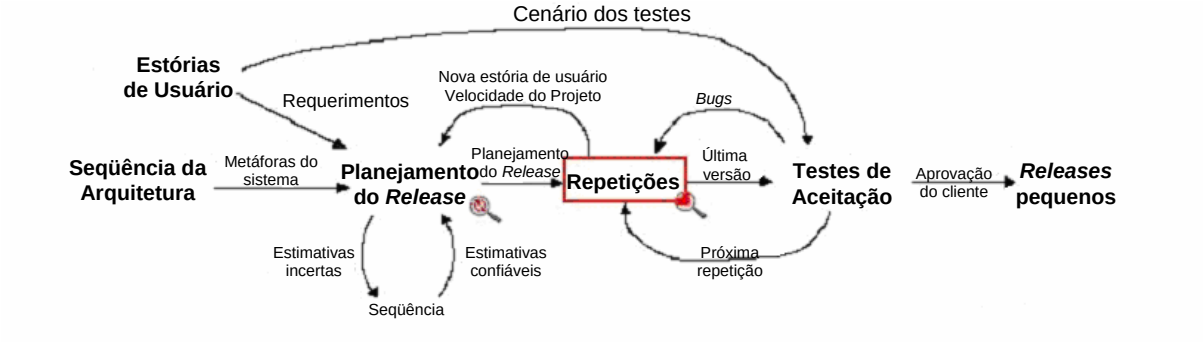

Figura 07 Projeto da Extreme Programming
Fonte: Wells, 2003

2.3 SCRUM

O termo SCRUM surge originalmente do Japão, citado em 1986 por Takeuchi e Nonaka, sendo apresentado com um processo de desenvolvimento de produtos adaptável, rápido e auto-organizável (SCHWABER e BEEDLE, 2002). Visa alcançar um objetivo através do trabalho em grupo, onde o foco é aplicar o controle do processo industrial à teoria de desenvolvimento de sistemas, através de idéias de flexibilidade, adaptabilidade e produtividade. Não define nenhuma técnica de desenvolvimento de software específica para a fase de implementação. SCRUM se concentra em como os envolvidos no projeto devem trabalhar para produzir a flexibilidade do sistema dentro de uma constante variável do ambiente.

A idéia principal de SCRUM está no desenvolvimento de sistemas que envolvem um contingente de variáveis de ambiente e técnicas (por exemplo exigências, prazo, recursos, e tecnologia), sendo provável que este cenário tenha

alterações durante o processo. Isto faz o processo de desenvolvimento imprevisível e complexo, requerendo flexibilidade para responder às mudanças. Como resultado do processo de desenvolvimento, tem-se um sistema funcional para ser fornecido ao cliente (SCHWABER 1995).

SCRUM auxilia a melhorar as práticas de engenharia existentes (por exemplo, testando práticas) em uma organização. Para isto envolve atividades de administração freqüentes que apontam e identificam constantemente. Qualquer deficiência ou impedimentos no processo de desenvolvimento, como também, as práticas que são usadas no mesmo.

O ciclo SCRUM é composto por três fases, sendo: Pré-jogo, Desenvolvimento e Pós-jogo, como demonstra a Figura 08.

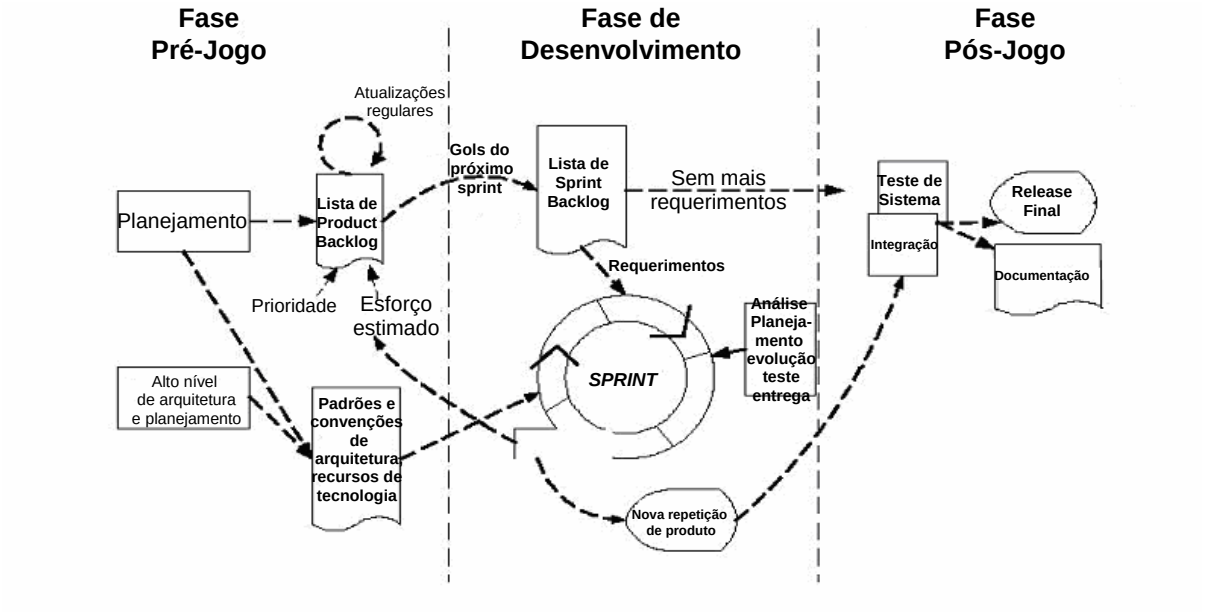

Figura 08 Processo SCRUM
Fonte: Abrahamsson,2002

A fase de pré-jogo é composta por duas sub-fases: Planejamento e Arquitetura em nível de planejamento (***planejamento***).

O planejamento trata a definição do início do sistema a ser desenvolvido. Um ***Product Backlog*** (lista de funções) é criado, contendo todas as exigências que são atualmente conhecidas. As exigências podem originar do cliente, vendas, marketing, apoio de cliente ou os desenvolvedores do software.

Os requerimentos são priorizados de acordo com as exigências de cada um e

então, é realizado um cálculo para estimar o custo da implementação. A lista de reserva de produtos é constantemente atualizada com novos artigos, a cada momento com um número maior de detalhes, possibilitando assim, novos cálculos para estimações mais precisas e alterações de prioridade. A fase de planejamento inclui a definição da equipe de projeto, ferramentas que serão utilizadas e demais recursos, bem como uma avaliação de riscos. Na fase de arquitetura, o alto nível de planejamento do sistema inclusive, a arquitetura é planejada baseada nos artigos atuais da lista de reserva de produto. Em caso de mudanças que elevem os custos do projeto de um sistema existente, as mesmas são identificadas junto à lista dos artigos de Reserva de forma a evidenciar os problemas que podem ser causados. Uma reunião de revisão de planejamento é realizada de forma a focar as propostas para as chamadas tomadas de decisão que serão a base para uma nova implementação.

A fase de desenvolvimento é a parte ágil da SCRUM. Esta fase é tratada como uma "caixa preta" onde o imprevisível é esperado (ABRAHAMSSON, 2002). As variáveis ambientais e técnicas diferentes (qualidade, exigências, recursos, tecnologias de implementação e ferramentas) identificados em SCRUM, podem mudar durante o processo. São observadas e controladas por várias práticas SCRUM durante as Sprints da fase de desenvolvimento. Ao invés de considerar estes fatores apenas no início do projeto de desenvolvimento, SCRUM foca constantemente em uma procura de controle através de uma flexibilidade que se adapte às mudanças.

Na fase de desenvolvimento o sistema é desenvolvido em s***prints*** (tarefas de curta duração ou reserva de produto). ***Sprints*** são ciclos de repetição, onde a funcionalidade é desenvolvida ou aumentada para produzir novas repetições. Cada ***sprint*** inclui as fases tradicionais de desenvolvimento de software: exigências, análise, planejamento, evolução e entrega. A arquitetura e o planejamento do sistema evoluem durante o desenvolvimento de um ***sprint***. É planejado para ter sua duração entre uma semana a um mês. Por exemplo, pode haver três a oito ***sprints*** em um processo de desenvolvimento de sistemas, antes de o sistema estar pronto para a distribuição. Também pode haver mais de uma equipe participando de uma repetição (ABRAHAMSSON, 2002).

A fase de pós-jogo contém o encerramento da liberação. Esta fase inicia-se quando um acordo foi feito, indicando que as variáveis ambientais estão com suas

exigências completadas. Neste momento, artigos ou assuntos novos não podem ser criados, a lista permanece vazia. O sistema está pronto para a liberação, incluindo as tarefas como a integração, prova de sistema e documentação.

SCRUM não requer nenhuma prática de engenharia específica. Pode ser adotado para administrar qualquer prática de engenharia utilizada em uma organização (SCHWABER e BEEDLE, 2002). Porém, SCRUM pode mudar consideravelmente as descrições dos cargos presentes na equipe de projeto. Por exemplo, o gerente de projeto, já não organiza a equipe, mas a equipe se auto-organiza e toma decisões de interesse comum. Então, o gerente trabalha para remover os impedimentos do processo. É responsável pela validação diária das decisões com a administração; o papel dele agora é de um treinador em lugar de um gerente no projeto.

Schwaber e Beedle (2002) identificam dois tipos de situações nas quais SCRUM pode seja adotado: um projeto existente e um projeto novo. Estes são descritos dentro do seguinte: um caso típico de adotar SCRUM em um projeto existente é uma situação onde o ambiente de desenvolvimento e a tecnologia a serem utilizados são conhecidos, mas a equipe de projeto está lutando com problemas relacionados a exigências variáveis e tecnologia complexa. Neste caso, a introdução de SCRUM é iniciada diariamente. A meta do primeiro ***sprint*** deveria ser "demonstrar qualquer parte da funcionalidade do usuário na tecnologia selecionada" (SCHWABER e BEEDLE, 2002). Isto ajudará a equipe a acreditar em si mesma, e o cliente a acreditar na equipe. Durante os primeiros ***sprints***, são identificados os impedimentos do projeto e removidos para habilitar a progressão da equipe. Ao término do primeiro ***sprint***, o cliente e a equipe, junto com o “mestre de SCRUM”, realizam uma revisão de ***sprint*** e juntos decidem o próximo passo. No caso de continuar com o projeto já existente, é realizado um ***sprint*** que irá planejar possibilidades garantindo as decisões, metas e exigências contempladas nos projetos seguintes.

No caso de adotar SCRUM em um projeto novo, Schwaber e Beedle (2002) sugerem que o primeiro trabalho, seja realizado com a equipe e o cliente, durante vários dias, de forma a construírem uma reserva de produto inicial. Neste momento, a reserva de produto pode consistir em funcionalidades empresariais e exigências de tecnologia. A meta do primeiro ***sprint*** é então "demonstrar uma parte fundamental de funcionalidade do usuário na tecnologia selecionada" (SCHWABER e BEEDLE

2002).

O ***sprint*** para criar a lista de reserva inicial, também inclui as tarefas de montar a equipe e distribuir os papéis de SCRUM, construindo práticas de administração, além das tarefas atuais de implementar a demonstração.

Rising e Janof (2000), em um relatório de experiências prósperas no uso de SCRUM em três projetos de desenvolvimento de software, sugerem que:

> Claramente, SCRUM não é aplicável para estruturas de equipes grandes e complexas, porém, consideramos que equipes isoladas em um projeto grande poderiam fazer uso de alguns elementos de SCRUM. Esta é verdadeira diversidade de processo.

Decidiram que duas a três vezes por semana são suficientes para manter as sprints dentro dos prazos. Por exemplo, outras experiências positivas incluem voluntarismo (membro(s) da equipe que assume(m) o compromisso de uma dedicação maior em relação ao tempo, buscando agilizar o processo), aumentado dentro da equipe a resolução de problema de uma maneira mais eficiente.

SCRUM é um método satisfatório para equipes pequenas, com menos de 10 engenheiros. Schwaber e Beedle (2002), sugerem para a equipe incluir cinco a nove sócios de projeto.

Recentemente, podem ser achados esforços para integrar XP e SCRUM juntas. SCRUM é vista para prover o vigamento de administração de projeto pelo qual é apoiado a prática XP, formando um pacote integrado para equipes de desenvolvimento de software. Os autores reivindicam que isto aumenta a escalabilidade de XP a projetos maiores. Porém, ainda não existem estudos que apóiem estes argumentos.

2.4 *Crystal Family*

As metodologias que englobam a ***crystal family*** (grupo de metodologias com características em comum) são diversas, de forma a satisfazer projetos individuais, de acordo com os requisitos apresentados, através de princípios que atendem circunstâncias variadas de diferentes projetos.

Cada metodologia integrante é associada a uma cor que indica o grau de complexidade da mesma, ou seja, quanto mais escura a cor, mais complexa a metodologia, conforme demonstrado na Figura 15.

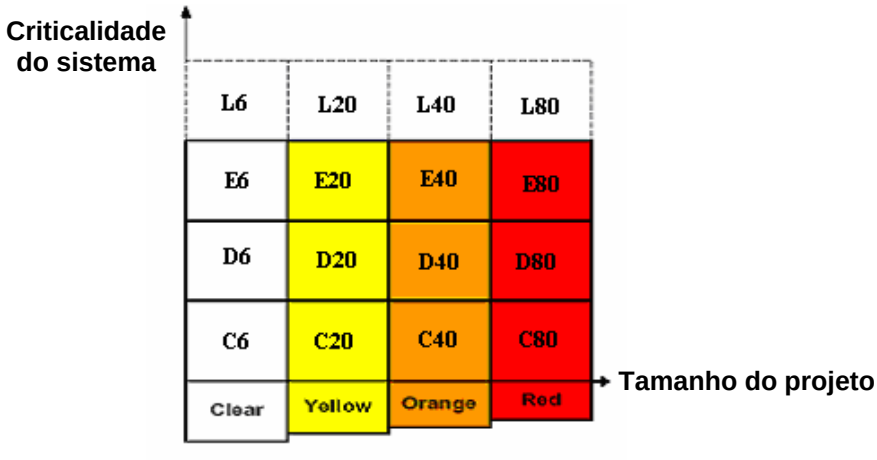

Figura 09 Dimensões das metodologias da crystal family
Fonte: Abrahamsson,2002

Há três principais metodologias ***crystal*** atualmente: ***Crystal Clear, Crystal Orange*** e ***Cristal Web Orange*** (COCKBURN 2002A).

Crystal sugere que a escolha da metodologia deve ser apropriada, considerando fatores como o tamanho do projeto e o quanto pode ser considerado um projeto crítico.

Há certas regras, características e valores que são comuns a todos os métodos na ***crystal family***. Em primeiro lugar, os projetos sempre usam um ciclo de desenvolvimento com um tempo máximo de quatro meses, mas, preferivelmente entre um e três meses (COCKBURN 2002A). A ênfase é focada na comunicação e cooperação das pessoas. Metodologias ***crystal*** não limitam qualquer prática de desenvolvimento, ferramentas ou produtos de trabalho, permitindo por exemplo: a adoção de práticas como XP e SCRUM. (COCKBURN 2002A).

Todas as metodologias da ***crystal family*** provêem diretrizes padrão de política, produtos de trabalho, "assuntos locais", ferramentas, padrões e papéis a serem seguidos no processo de desenvolvimento. ***Crystal clear*** e ***crystal orange*** são os dois membros da ***crystal family*** que foram criados e realmente usados (COCKBURN 1998; COCKBURN 2002A), merecendo um melhor detalhamento de sua estrutura, com seus conceitos, semelhanças e diferenças.

Crystal clear é metodologia para projetos muito pequenos (um projeto D6, segundo a categoria de projetos), incluindo até seis desenvolvedores, porém, com alguma extensão para comunicação e testes, também pode ser aplicada a projetos das categorias E8 e D10. Uma equipe usando ***crystal clear*** pode ser localizada em

um ambiente de trabalho compartilhado, devido as limitações em sua estrutura de comunicação (COCKBURN, 2002A).

Crystal orange é metodologia para projetos de tamanho médio, com um total de 10 a 40, membros no projeto (categoria D40), e com uma duração de projeto de um a dois anos. Com alterações no processo de verificação/teste, também pode ser aplicada a projetos da categoria E50. O projeto é dividido em várias equipes, sendo que os grupos possuem suas funcionalidades cruzadas, usando a estratégia de diversidade holística. Por enquanto, este método não suporta o ambiente de desenvolvimento distribuído. Uma das características presentes na metodologia é considerar o tempo de mercado, ou seja, estar sempre pronta para adaptações, de forma a dar suporte às equipes em uma mudança de mercado, sem perder a funcionalidade do projeto.

Conforme demonstrado na Figura 10, observa-se o ciclo em ***crystal orange***.

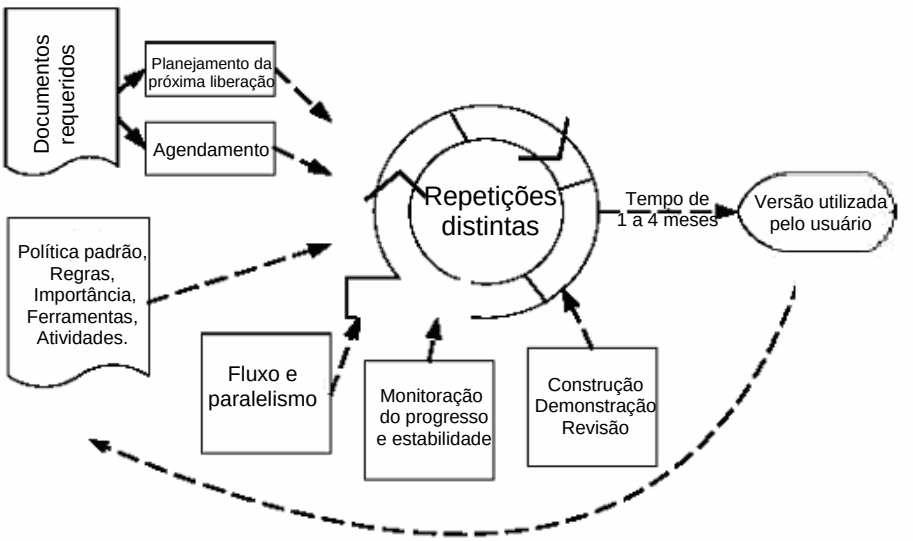

Figura 10 Uma repetição em crystal orange
Fonte: Abrahamsson,2002

Descreve-se na seqüência, um relatório de experiência de um projeto adotando as práticas da metodologia ***crystal orange*** (COCKBURN, 1998), denominado de ***Winifred***. Englobando uma duração de dois anos, o “projeto ***Winifred***”, é considerado um projeto médio e teve sua equipe composta entre 20 a 40 membros. Foi desenvolvido com a estratégia de repetição de ter uma meta de aproximação orientada a objetos para reescrever um sistema legado de mainframe.

As metodologias da ***crystal family*** não cobrem projetos com um ciclo de vida crítico como os ilustrados na Figura 09 (COCKBURN 2002a). Outra restrição identificada por Cockburn (2002A) é que: só equipes que estejam inseridas em um mesmo ambiente, estruturadas para uma troca de informações cruzadas podem ser endereçadas por estas metodologias.

Cockburn (2002a) também identificou limitações relativas aos indivíduos que utilizam as metodologias da ***crystal family***. Por exemplo, ***crystal clear*** tem uma estrutura de comunicação relativamente restringida e só é satisfatória para uma única equipe localizada em um único ambiente. Além disso, ***crystal clear*** exibe problemas claros em elementos de validação de sistemas e não é satisfatória para sistemas com ciclo de vida críticos.

Crystal orange não possui suporte para uma equipe substituta, sendo assim, é satisfatória para projetos que envolvem até 40 pessoas. Também não é considerada muito competente quanto ao planejamento de projeto e atividades de verificação de código.

Se conclui que, das quatro metodologias existentes na ***crystal family***, apenas duas são aplicáveis e dentre estas, existem muitas falhas que podem ocasionar problemas em desenvolvimento de projetos.

2.5 *Feature Driven Development* (FDD)

Feature Driven Development (FDD) é uma abordagem ágil e adaptável para sistemas em desenvolvimento. A abordagem de FDD não abrange todo o processo de desenvolvimento de software, foca no planejamento e fases de construção. Porém, foi projetada para trabalhar com outras atividades de um projeto de desenvolvimento de software (PALMER e FELSING, 2002) e não exige a utilização de nenhum modelo de processo específico. A abordagem de FDD abrange o desenvolvimento de repetições com as melhores práticas encontradas na indústria de desenvolvimento de software, sendo aspectos com ênfase na qualidade ao longo do processo e entregas freqüentes e tangíveis de código, aliadas a um monitoramento preciso do progresso do projeto.

FDD consiste em cinco processos seqüenciais e provê os métodos, técnicas, e diretrizes necessárias para que os membros do projeto possam entregar um sistema funcional. Além disso, FDD inclui os papéis, regras, metas e fases de desenvolvimento bem definidas, os quais, são necessários dentro de um projeto

(PALMER e FELSING 2002). Ao contrário de algumas outras metodologias ágeis, FDD se mostra satisfatório para o desenvolvimento de sistemas críticos (PALMER e FELSING 2002).

O processo da FDD, como mencionado anteriormente, consiste em cinco processos seqüenciais, durante os quais, o planejamento e o desenvolvimento do sistema, são executados conforme Figura 11. As repetições, seguindo o esquema de planejamento e desenvolvimento, fornecem apoio ao desenvolvimento ágil, com adaptações rápidas, para mudanças recentes, em exigências e necessidades de negócio. Tipicamente, uma repetição de uma característica envolve um período entre uma a três semanas de trabalho para a equipe.

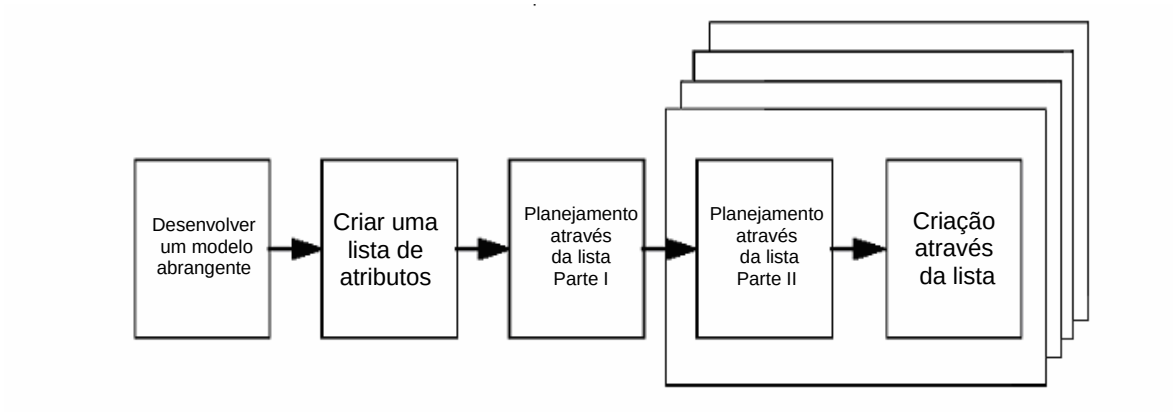

Figura 11 Processo da FDD (segundo Palmer e Felsing, 2002)
Fonte: Abrahamsson,2002

Um grupo pequeno de características é selecionado do conjunto e equipes que se adeqüam com estas características selecionadas são formadas pelos gerentes de classe. O planejamento e desenvolvimento através de características, resulta em procedimentos de repetição, durante os quais, as características selecionadas são produzidas. Uma repetição deve levar entre alguns dias ou no máximo três semanas. Pode haver equipes com um conjunto de características múltiplas, projetando simultaneamente e desenvolvendo seu próprio conjunto de características. Este processo de repetição inclui tarefas como revisão de planejamento durante a codificação, unidades de teste, integração e revisão de código (ABRAHAMSSON, 2002).

Depois de uma repetição próspera, as características completadas são passadas para o desenvolvimento principal, enquanto novas repetições com planejamento e desenvolvimento, começam com um novo grupo de características.

A estrutura pode ser observada conforme demonstrado na Figura 12.

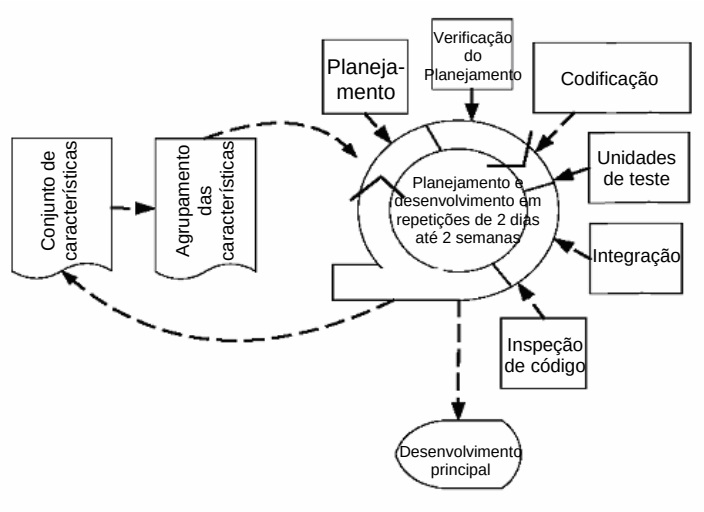

Figura 12 Planejamento e desenvolvimento através de características (FDD)
Fonte: Abrahamsson, 2002

De acordo com Palmer e Felsing (2002), FDD é satisfatório para projetos novos, projetos que aumentam, código já existente (atualizações) e projetos com a tarefa de criação de uma segunda versão de uma aplicação existente. Os autores também sugerem que as organizações deveriam adotar o método gradualmente, em partes pequenas e finalmente em sua totalidade.

FDD é considerado uma metodologia séria por qualquer organização de desenvolvimento de software que precisa entregar qualidade em sistemas críticos e principalmente com um tempo de entrega que não pode ser desconsiderado (PALMER e FELSING 2002). Relatórios de experiência seguros dificilmente são encontrados. Geralmente o material consultado trata-se de documentação criada por empresas que utilizem a metodologia, fornecendo assim, histórias de sucesso não comprovadas, ou, descaracterizadas cientificamente. Por outro lado, não se pode afirmar que se trata de uma metodologia que tende ao fracasso, pois, qualquer pesquisa, discussões e comparações com este foco, seriam de interesse a acadêmicos e usuários pesquisadores da metodologia, distante do foco comercial visto pelo mercado. Isto acaba não fornecendo uma resposta precisa para distinguir quando é melhor a implementação através da FDD ou não, ficando cada caso específico a ser avaliado para tomar tal decisão em relação a outro método ágil (ABRAHAMSSON, 2002).

Como o método é relativamente novo, está em uma crescente evolução. Ferramentas que forneçam um apoio para a metodologia estarão provavelmente

disponíveis em pouco tempo e uma pesquisa mais abrangente sobre suas vantagens ou desvantagens provavelmente será realizada em breve. Entretanto, FDD se mostra uma metodologia interessante e promissora.

2.6 *Dynamic Systems Development Method* (DSDM)

Desde sua origem em 1994, o Método de Desenvolvimento de Sistemas Dinâmicos (DSDM), se tornou gradualmente o primeiro em aplicações de desenvolvimento rápido no Reino Unido (STAPLETON, 1997). DSDM é um ***framework*** para desenvolvimento de RAD (***Rapid Application Development***, ou seja, projetos de desenvolvimento rápido) mantido por um consórcio (www.dsdm.org), que não visa lucro. A idéia fundamental atrás de DSDM é abranger a maior quantia possível de funcionalidades em um produto, e ir ajustando tempo e recursos para alcançar essa funcionalidade.

DSDM é dividido em cinco fases: estudo de viabilidade, estudo empresarial, repetição funcional do modelo, repetições de planejamento/desenvolvimento e implementação, conforme Figura 13. As duas primeiras fases são seqüenciais e realizadas uma única vez. As demais fases são executadas durante todo trabalho de desenvolvimento. As repetições são como uma ***timebox***, ou seja, o ciclo de repetição deve ser realizado dentro de um tempo determinado. O tempo permitido é variável e deve ser planejado anteriormente para cada repetição, o conjunto de resultados com as repetições realizadas são a garantia de produção do projeto. Em DSDM, uma duração de ***timebox*** típica é de alguns dias a algumas semanas.

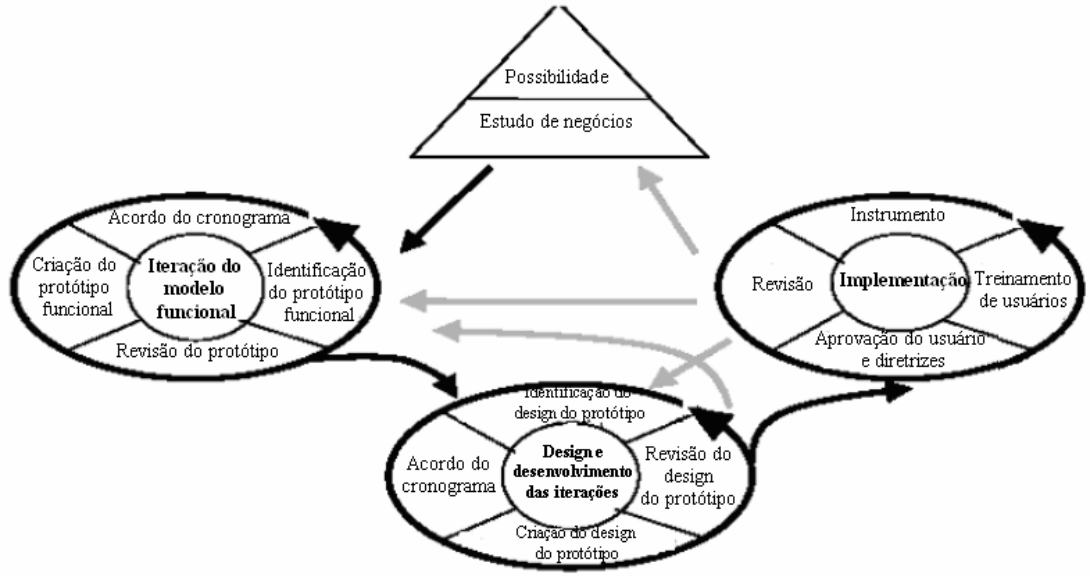

Figura 13 Diagrama do processo DSDM (STAPLETON,1997)
Fonte: Abrahamsson, 2002

DSDM teve seu uso difundido no Reino Unido desde aproximadamente a segunda metade da década de 90. Oito casos de estudos foram documentados dentro deste período (STAPLETON, 1997) e as experiências demonstram claramente que se trata de uma alternativa viável para RAD. Para facilitar a adoção de método e esclarecer dúvidas eventuais, o consórcio responsável por DSDM, publicou um filtro de conveniências do método, no qual três áreas estão cobertas: negócio, sistemas, e área técnica. Demonstram-se as perguntas principais na Tabela 2.

Tabela 02 Dúvidas sobre DSDM

Questão	Resposta
As funcionalidades são razoavelmente visíveis na interface do usuário?	Aos usuários realizarem a prototipação precisam obter uma verificação que o software produzido possua as ações corretas sem o usuário ter que entender os detalhes técnicos.
É possível identificar claramente todas as classes de usuários finais?	Isto é importante para poder identificar e envolver todo grupo de usuários pertinentes, de forma que o produto do software possa abranger todas as exigências.
A computacionalidade da aplicação é complexa?	Há um risco aparente na complexidade do desenvolvimento das funcionalidades pelo uso de DSDM. São obtidos resultados melhores se alguns dos blocos desenvolvidos já estiverem disponíveis para a equipe de desenvolvimento.
A aplicação é potencialmente grande? Caso positivo, seria possível uma divisão em componentes funcionais menores?	DSDM é utilizado para o desenvolvimento de sistemas grandes. Alguns destes projetos tiveram um período de desenvolvimento de 2 a 3 anos. Porém, a funcionalidade é desenvolvida através de pequenos blocos de desenvolvimento.
O tempo de projeto realmente é necessário?	O envolvimento ativo do usuário é de suprema importância no método de DSDM. Se os usuários não participam do planejamento do desenvolvimento (devido ao fato que o planejamento não seja obrigatório para os mesmos), o desenvolvedor pode começar a fazer suposições sobre o que é preferencial no sistema para atingir os prazos finais.
Os requerimentos são flexíveis e só especificados em alto nível?	Se os requerimentos forem detalhados e fixados antes do desenvolvimento, não serão alcançados os benefícios da DSDM.

Fonte: Abrahamsson, 2002

As equipes de DSDM podem variar entre dois a seis integrantes e podem haver muitas equipes em um único projeto. O mínimo de duas pessoas envolvidas vem do fato que cada time tem que ter um usuário e um desenvolvedor pelo menos. O máximo de seis é um valor encontrado através de experiências práticas. DSDM foi aplicado em pequenos e grandes projetos semelhantes. A condição prévia para utilizar a metodologia em sistemas grandes é que o sistema possa ser dividido em componentes que possam ser desenvolvidos em equipes pequenas (ABRAHAMSSON, 2002).

Ao considerar o domínio de aplicação, Stapleton (1997) sugestiona que DSDM é melhor e mais facilmente aplicada a sistemas empresariais em contrapartida a sistemas com uma aplicação científica.

Foi desenvolvida originalmente e continua sendo mantido por um consórcio

que consiste em várias companhias associadas. Fora do consórcio não há nenhuma pesquisa identificada, enquanto dentro do consórcio o método continua evoluindo. Como um exemplo de sua evolução, em 2001 uma versão do método denominada e-DSDM foi criada para atender o mercado de ***eBusiness*** e ***eCommerce*** (HIGHSMITH, 2002).

2.7 *Adaptive Software Development* (ASD)

Desenvolvimento de Software adaptável, ou simplesmente ASD, foi desenvolvida por James A. Highsmith III, e publicada em Highsmith (2000). Muitos dos princípios de ASD se originaram da pesquisa de Highsmith em métodos de desenvolvimento com repetições. O antecessor de ASD, “Desenvolvimento de Software Radical”, desenvolvido através de um trabalho de Highsmith e S. Bayer, publicado em Bayer e Highsmith (1994), acabou por estimular a criação da metodologia.

ASD possui seu foco de atuação principalmente nos problemas de sistemas complexos, para grandes desenvolvimentos. O método estimula fortemente o desenvolvimento com repetições e uma constante prototipação.

Um projeto de ASD é dividido em ciclos compostos de três fases. As fases dos ciclos são Especulação, Colaboração, e Aprendizado, conforme a Figura 14.

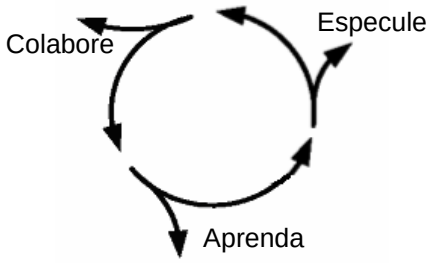

Figura 14 Ciclo ASD (HIGHSMITH,2000)
Fonte: : Abrahamsson, 2002

As fases são nomeadas para enfatizar de certo modo o papel de mudança no processo.

1. Especulação é utilizada no lugar de planejamento, pois, um plano geralmente é visto como algo onde incerteza é uma fraqueza, e no qual divergências indicam fracasso (ABRAHAMSSON, 2002).

2. A colaboração realça a importância de trabalho de equipe como o meio de sistemas de automudança em desenvolvimento.
3. Aprendizagem, devido à necessidade para reconhecer e reagir a decisões erradas e o fato que os requisitos podem mudar durante desenvolvimento.

A Figura 21 ilustra os ciclos de desenvolvimento adaptáveis com maiores detalhes. A fase de projeto inicial define as bases do projeto e é iniciada através da definição da missão do mesmo. A missão fixa uma linha de trabalho para o produto final e todo o desenvolvimento é guiado de forma que a missão será atingida.

A missão está definida em três itens: a visão do projeto caracterizada, a lista dos dados do projeto, e um esboço de especificação de produto. A fase de Iniciação de Projeto fixa uma agenda de trabalho, definindo o cronograma com datas e objetivos para o desenvolvimento dos ciclos, que possuem tipicamente a duração entre quatro e oito semanas.

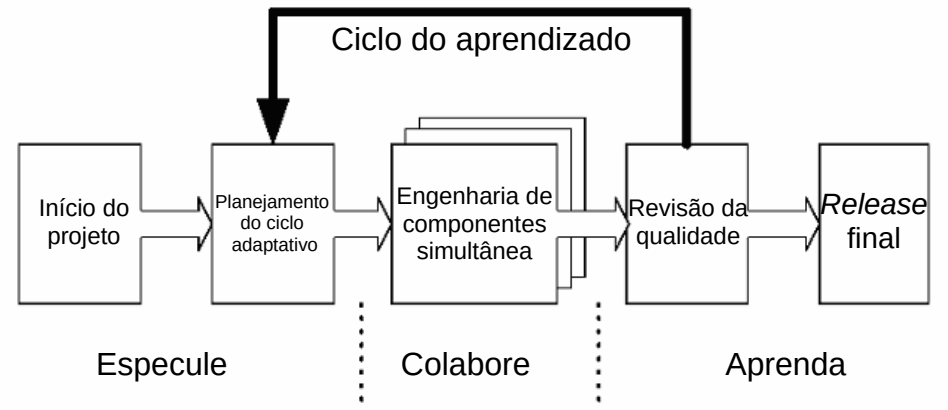

Figura 15 Fases do ciclo de vida do DSD (HIGHSMITH, 2000)
Fonte: Abrahamsson,2002

Como um resumo, a natureza adaptável de desenvolvimento em ASD é caracterizada pelas seguintes propriedades, conforme Tabela 3:

Tabela 03 Propriedades de ADS

Característica	Descrição
Missão do projeto	As atividades de cada ciclo de desenvolvimento devem ser justificadas em relação à missão global do projeto. A missão pode ser ajustada, de acordo com o desenvolvimento do projeto.
Baseado em componentes	As atividades de desenvolvimento não devem ser orientadas a tarefas, mas, focadas nas funcionalidades do desenvolvimento do software, isto é, construindo o sistema em várias partes, atendendo uma de cada vez.
Interatividade	Um método em cascata só trabalha dentro de um ambiente bem definido e compreendido. A maioria dos desenvolvimentos é turbulenta, e o esforço de desenvolvimento deveria ser focalizado em refazer. Ao invés de fazer isto, corrija na primeira codificação.
Caixa de tempo	A ambigüidade em projetos de software complexos pode ser amenizada por prazos finais tangíveis, se fixando em uma base regular. Um projeto que adote o esquema de tempo, para a administração, força os participantes a anteciparem decisões difíceis, porém inevitáveis durante o projeto.
Tolerância a mudanças	Mudanças são freqüentes em desenvolvimento de software. Então, é mais importante poder se adaptar as mudanças ao invés de tentar controlar as mesmas. Para construir um sistema tolerante a mudanças, é necessário que os desenvolvedores avaliem constantemente se os componentes que estão sendo construídos possuem a possibilidade de mudar.
Risco controlado	O desenvolvimento de itens de alto risco (por exemplo, não tão bem conhecido, ou mais crítico a mudanças) deve iniciar o mais cedo possível.

Fonte: Abrahamsson,2002

Os princípios e as idéias subjacentes atrás de ASD são razoáveis, mas poucas diretrizes são determinadas para colocar o método em uso. O livro (HIGHSMITH, 2000) demonstra em geral várias perspicácias em desenvolvimento de software, e fornece uma leitura muito interessante. Segundo Abrahamsson (2002):

> A filosofia adotada pelo autor é em grande parte sobre construir uma cultura organizacional adaptável, não sobre os particulares. Como uma conseqüência, o método oferece uma pequena base para adotar ASD de fato em uma organização.

ASD não tem limitações embutidas para sua aplicação. Uma característica interessante na extensão de usar ASD é que não obriga o uso de equipes dentro de um mesmo ambiente como outros métodos de desenvolvimento de software ágeis. Freqüentemente as equipes precisam trabalhar por espaço, tempo e limites organizacionais. Highsmith (2000) destaca pontos afirmando que a maioria das dificuldades em desenvolvimento distribuído é relacionado à reunião social, cultural e

habilidades da equipe. Conseqüentemente, ASD oferece técnicas para aumentar a colaboração e o entrosamento entre a equipe, sugerindo informações que compartilhem estratégias, uso de ferramentas de comunicação e modos de gradualmente introduzir um rigor de trabalho no projeto para apoiar o desenvolvimento distribuído.

Ainda não foi realizada nenhuma pesquisa significante em ASD. Poucas experiências foram documentadas sobre o retorno da utilização de ASD, porém, os princípios subjacentes do desenvolvimento de software adaptável não estão desaparecendo devido a este fato. Seus idealizadores estão tentando avançar com o tema, focando nos componentes centrais dos ambientes de desenvolvimento de software ágil, tentando esclarecer as incertezas presentes em relação às pessoas (ABRAHAMSSON, 2002).

3 COMPARANDO MÉTODOS ÁGEIS

A tarefa de comparar objetivamente qualquer metodologia com outra é difícil e o resultado é freqüentemente baseado nas experiências subjetivas de usuários e considerações dos autores (SONG e OSTERWEIL, 1991). Duas aproximações alternativas existem: informal e comparação semiformal (SONG e OSTERWEIL, 1992).

A comparação semiformal tenta superar as limitações subjetivas de uma técnica de comparação informal. De acordo com Sol (1983) podem ser checadas comparações semiformais de cinco modos diferentes:

a) Descrevendo um método idealizado e avaliando outros métodos contra isto.
b) Destilando um conjunto de características indutivas importantes de vários métodos e comparando cada método contra isto.
c) Formulando a priori, hipóteses sobre os requerimentos dos métodos e derivando um framework da evidência empírica em vários métodos.
d) Definindo uma metalinguagem como um veículo de comunicação e um quadro de referência contra o qual se descreve muitos métodos.
e) Usando uma aproximação contingente e tentando relacionar as características de cada método para problemas específicos.

Song e Osterweil (1992) argumentam que o segundo e o quarto tipo de aproximação são mais próximas ao método científico clássico usado para propósitos de comparação de métodos. Justamente o quarto modo de comparação, isto é, detalhando uma metalinguagem como um veículo de comunicação, foi adotado de forma a tentar aproximar ao máximo as definições e conceitos envolvidos em cada método descrito.

DSDM e SCRUM foram introduzidos no início e meio dos anos 90 respectivamente, porém, pode-se dizer que SCRUM tem seu estado em uma fase de crescente evolução desde que estudos que citam o uso do método ainda estão escassos.

Outros métodos também reconhecidos amplamente são FDD, ***Crystal*** e ASD, porém, ainda menos se conhece sobre a atual utilização dos mesmos. Assim, eles também podem ser considerados em fase de construção.

Já outros métodos propostos começaram a despontar em meados do ano 2000. São considerados em estado “nascente” e ainda não possuem sequer um estudo ou documentação suficiente. XP, e DSDM, por outro lado, são métodos ou aproximações que foram bem documentadas, os quais possuem vários títulos de literatura e muitas experiências documentadas disponíveis. Além disso, cada método gerou sua própria pesquisa ativa e comunidades de usuários. Assim, eles podem ser classificado como “ativos”.

Não se pode afirmar que qualquer um dos métodos citados está fadado ao esquecimento ou abandono de pesquisas, embora, por exemplo, é considerado que o uso da terminologia de DSDM como prototipação é antiquada (ABRAHAMSSON, 2002). A classificação das fases em relação aos métodos pode ser vista na Tabela 4.

Tabela 04 Classificação das fases em relação aos métodos

Situação em agosto/2002	**Descrição**	**Método**
Nascent (métodos novos)	Métodos muito recentes. Nenhuma pesquisa identificável existe, nenhuma experiência informada	AM
Building up (métodos em estado de evolução)	Método é reconhecido amplamente, relatórios de experiência em crescimento, ativa comunidade trabalhando sobre o mesmo, pesquisas relativas ao método são identificáveis.	FDD, Criystal, Scrum, ASD.
Active (métodos em utilização com documentação)	Método detalhado em vários lugares, amplas experiências, relatórios facilmente encontrados, pesquisa altamente ativa, grande comunidade pesquisando e utilizando.	XP, DSDM

Fonte: Abrahamsson,2002

Pode-se observar que cada método atinge os problemas enfrentados em software através de um ângulo diferente. A Tabela 5 resume cada método utilizando três aspectos selecionados: pontos chave, características especiais e falhas identificadas. “Pontos chave” detalham os métodos.

Tabela 05 Resumo dos métodos através de aspectos

Nome do método	Pontos-chave	Características especiais	Falhas detectadas
ASD	Cultura adaptável, colaboração, base de funcionamento dos componentes é focada nas repetições do desenvolvimento.	Organizações são vistas como sistemas adaptáveis. Criando uma ordem emergente, independente da teia de interconexão dos indivíduos.	ASD é mais uma abordagem de conceitos e cultura do que a prática realmente de desenvolvimento de software.
AM	Aplicando princípios ágeis para modelar: Cultura ágil, trabalho organizado para suportar a comunicação, simplicidade.	Pensamentos ágeis também são aplicados para a modelagem.	Este é um ótimo adicional para a filosofia dos profissionais de modelagem, porém, só trabalha dentro de outros métodos.
CRYSTAL	Família de métodos. Cada um tem os mesmos valores e princípios. Técnicas, papéis, ferramentas, e padrões variam.	Método de planejamento a princípios. Habilidade para selecionar o método mais satisfatório, baseado no tamanho do projeto e no fator de quanto o projeto é considerado crítico.	Métodos muito recentes para se estimar a produtividade: Apenas dois dos quatro métodos sugeridos existem realmente na prática.
DSDM	Aplicação de controle para RAD, uso de timeboxing (espécie de cronograma para controlar o tempo dedicado em cada ação), consórcio para guiar o desenvolvimento do método.	É o primeiro método realmente ágil de desenvolvimento de software, uso de prototipação, Usuário com vários papéis: “o embaixador”, “visionário” e “o aconselhador.”	Enquanto o método é avaliado e de certa forma aprimorado, apenas sócios do consórcio que o mantém, possuem acesso aos artigos e procedimentos de documentos que contém a evolução do método.
XP	O desenvolvimento é voltado para o Cliente, equipes pequenas, liberação de software diária.	Refactoring contínuo, ou seja, a reconstrução do sistema para melhorar o seu desempenho e uma forte responsabilidade para adotar estas mudanças.	Práticas individuais são satisfatórias para muitas situações, visão global e práticas de administração são determinadas com menos atenção.
FDD	Processo de cinco passos, componentes orientados a objetos, baseado no desenvolvimento, repetições muito curtas: de horas a 2 semanas.	Simplicidade de método, planejamento, implementação do sistema através de modelagem de características do objeto.	FDD só focaliza em planejamento e implementação. Precisa de uma outra abordagem para apoiar sua utilização.
SCRUM	Independente, pequeno, auto-organizado equipes de desenvolvimento, Ciclos de liberação de 30 dias.	Reforça o paradigma de troca do “definido e repetível” para a “nova visão de desenvolvimento de produto através de SRUM.”	Enquanto SCRUM detalha em um conhecimento específico como administrar a liberação com um ciclo de 30 dias, a integração e os testes de aceitação não são detalhados.

Fonte: Abrahamsson, 2002

Segundo a Tabela 5 constatamos que, ASD é o método mais abstrato do ponto de vista de desenvolvimento de software. Embora muito atraente, sua meta fundamental, “criar uma ordem emergente fora de uma teia de indivíduos interconectados”, é difícil de atingir. Modelagem ágil (AM), XP, representam pontos

de vista em relação a uma prática orientada. Todos eles contém várias práticas empiricamente validadas e consideradas muito úteis pelos usuários. Como tal, são métodos muito valiosos.

As metodologias da ***crystal family*** são as únicas para sugestionar princípios de planejamento de método, explicitamente para permitir um desenvolvimento que depende de tamanho de projeto e de quanto é considerado crítico. Isto é um importante aspecto desde que escalabilidade do método é um dos principais tópicos que a comunidade ágil precisa se dirigir.

DSDM é diferenciado dos outros métodos por seu uso de prototipação, também atinge alguns papéis que outros não consideraram, como o “embaixador”, o “visionário” e o “aconselhador”. Estes papéis de usuário representam pontos de vista de clientes diferentes. A desvantagem de DSDM é a exigência de pertencer ao consórcio de DSDM para ganhar acesso aos artigos que discutem aspectos diferentes do método. FDD não tenta prover toda a solução em uma única metodologia de desenvolvimento de software, mas foca em uma abordagem de cinco passos simples que são baseados em identificar, projetar e implementar as características. Pressupõe que algum trabalho para o projeto já foi concluído e devido a este fator, não abrange as fases iniciais de desenvolvimento. SCRUM é uma abordagem de administração de projeto confiável em auto-organizar equipes independentes de desenvolvimento implementando, um projeto de software em ciclos de 30 dias chamados de ***sprints*** (corridas de curta distância).

Produtos de software só rendem benefícios empresariais se eles forem usados. Semelhantemente, os benefícios associados com métodos de desenvolvimento de software ágeis só são alcançáveis se estes métodos são usados no processo de produção. A adoção de uma nova tecnologia de processo deveria ser relativamente fácil de ser executada, desde que, as organizações não podem se dispor a reduzir a velocidade ou parar a produção para reorganizar ou aprender métodos novos (KRUEGER, 2002). Nandhakumar e Avison (1999) colocam que, atualmente métodos de desenvolvimento de software são usados “como uma ficção necessária para apresentar uma imagem de controle ou prover um estado simbólico”. Sugerindo que “abordagens alternativas que reconheçam o caráter particular de trabalho em tais ambientes são requeridas”. O caráter particular de trabalho a que se referem é o turbulento ambiente empresarial ou, como denominada por analistas, “a economia dirigida a mudança” (HIGHSMITH, 2002).

Neste ambiente, são esperadas mudanças no projeto, até mesmo nas fases iniciais. Nandhakumar e Avison (1999) consideram os desenvolvedores que trabalham com estas práticas, como sendo o melhor qualificado, possuindo uma capacidade de improvisação.

Podem-se caracterizar todos os métodos ágeis revisados como tendentes a ter uma certa ênfase nos seguintes aspectos (HIGHSMITH, 2002):

a) entrega de um produto útil;

b) confiança em pessoas;

c) colaboração encorajadora;

d) estimulação de uma técnica em excelência;

e) realização do trabalho da forma mais simples possível;

f) ser continuamente adaptável.

Nandhakumar e Avison (1999) argumentam que o corrente desenvolvimento de software é diferente da visão metodológica popular. A Tabela 6 demonstra que o desenvolvimento de software ágil aborda muitas práticas de desenvolvimento de software atuais, tratando o desenvolvimento de software pelo ponto de vista dos desenvolvedores. Assim, se explica em parte por que programação extrema recebeu muita atenção nos últimos anos.

Tabela 06 Método em relação às tarefas pelo ponto de vista dos desenvolvedores.

	Visão metodológica	**Desenvolvimento de software na prática**	**A visão defendida pelos métodos ágeis**
Atividades	Tarefas Discretas	Projetos pessoais relacionados	Projetos pessoais relacionados
	Duração previsível	Conclusão imprevisível	Conclusão da próxima liberação previsível
	Repetível	Dependente de contexto	Freqüentemente dependente de contexto
	Seguro	Depende de condições do contexto	Desempenho ligado a uma liberação pequena, se tornando seguro.
Performance do processo	Interações específicas	Interatividade por herança	Interações especificadas que promovem uma natureza de herança interativa
	Tarefas únicas em seqüência	Muitas tarefas entrelaçadas	Tarefas simples são planejadas freqüentemente devido a sua natureza dependente.
Esforço dos desenvolvedores	Dedicado a projetos de software	Comum a todas as atividades (por exemplo, projeto, pessoal, rotinas)	Desenvolvedores calculam o esforço de trabalho requerido.
	Indiferenciáveis	Específico a indivíduos	Específico a indivíduos
	Completamente avaliado	Completamente utilizado	Completamente utilizado
Controle do trabalho	Regularidade	Oportunismo, improvisação e interrupção.	Só controle de acordo mútuo existem, respeito para com o trabalho de outros.
	Controle de planejamento e gerenciamento	Preferência individual e negociação mútua	Preferência individual e negociação mútua

Fonte: Abrahamsson, 2002

A Figura 16 demonstra as fases de desenvolvimento de software contempladas por diferentes métodos ágeis, cada método é dividido em três blocos. O primeiro bloco indica se um método provê apoio para a administração de projeto. O segundo bloco identifica-se a existência de um processo para tal situação é descrito dentro do método. Finalmente o terceiro bloco, indica se o método descreve as práticas, atividades e produtos de trabalho que poderiam ser seguidos e utilizados sob circunstâncias variadas.

A cor cinza em um bloco indica que o método cobre a fase de ciclo de vida e a cor branca indica que o método não provê informação detalhada sobre uma das três áreas avaliadas, isto é, administração de projeto, processo e práticas/atividades/produtos de trabalho.

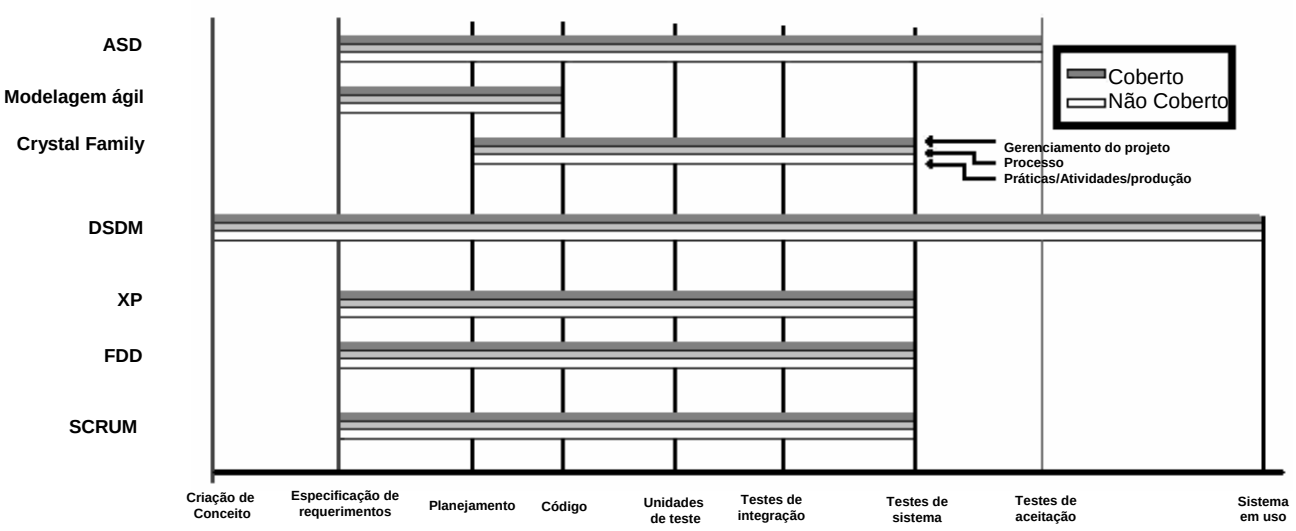

Figura 16 Ciclo de vida suportado no desenvolvimento de software
Fonte: Abrahamsson, 2002

Na figura 16 se percebe que métodos ágeis são focados em aspectos diferentes do ciclo de vida de desenvolvimento de software. Além disso, alguns são focados mais em práticas (XP), enquanto outros se preocupam em administrar o projeto de software (SCRUM). Contudo, há abordagens que cobrem todo o ciclo de vida de desenvolvimento (DSDM), enquanto que a maioria satisfaz a fase de especificação de exigências (FDD). Assim, há uma diferença clara entre os vários métodos ágeis de desenvolvimento.

Considerando que DSDM não precisa de abordagens complementares para apoiar o desenvolvimento de software, as demais, possuem um grau variado, porém, DSDM só está disponível para os sócios do Consórcio de DSDM, tornando o acesso às informações extremamente restrito.

Levando em consideração a adoção dos métodos, o tamanho da equipe de desenvolvimento é um dos principais assuntos decisivos. XP e SCRUM são focados em equipes pequenas, preferivelmente menos de 10 desenvolvedores de software. ***Crystal Family***, FDD, ASD e DSDM prometem a capacidade de até 100 desenvolvedores na mesma equipe (ABRAHAMSSON, 2002).

Porém, proponentes de métodos ágeis concordam que, quando o tamanho de uma equipe de desenvolvimento se torna muito grande, a quantia de documentação para que todos fiquem a par da situação, tende a aumentar proporcionalmente ao tamanho da equipe, fazendo o projeto “menos ágil” (ABRAHAMSSON, 2002). Cada método contém várias sugestões como organizar canais de comunicação dentro de

um grupo pequeno de desenvolvedores.

A aproximação ágil aparece particularmente satisfatória, em uma situação onde as exigências futuras, não são conhecidas (AMBLER, 2002B & HIGHSMITH, 2002).

Custo de adoção e dificuldades são difíceis de avaliar considerando que um número muito limitado de estudos sobre o assunto foi publicado, porém, o paradigma de apoio a adoção entre o modo tradicional de desenvolvimento de software e o desenvolvimento de software ágil é forte, sendo provável que desenvolvedores de organizações adotem os métodos ágeis em se uso cotidiano em um ritmo de adoção cada vez mais acelerado. Esta troca de paradigma foca outros assuntos que os processos tradicionais não tratam de uma maneira eficiente, como planejamento, preparação para o futuro, documentação e negociação de contrato, unindo esta estrutura com processos e ferramentas. Entretanto, não é provável que organizações façam uma migração completa das práticas de desenvolvimento de software existentes nas metodologias ágeis e sim, uma troca progressiva e em pequenos passos das práticas disponíveis (ABRAHAMSSON, 2002).

A maioria dos métodos ágeis autoriza o cliente e a equipe de desenvolvimento a tomar decisões importantes relativas ao processo de desenvolvimento. Assim, adotar uma metodologia ágil, também acaba por requerer uma troca cultural, especialmente na administração de cúpula. Muitas das formas de controle tradicionais são trocadas por outros mecanismos, como informes periódicos. Por um lado, isto requer uma mudança no modo de funcionamento da organização. Por outro lado, desata uma maior confiança na habilidade e competência da equipe de desenvolvimento.

Todas as aproximações ágeis propõem o desenvolvimento de características de software dentro de ciclos com pequenas repetições, em torno de 2 a 6 semanas de duração. Então, tornam o risco minimizado, pois, qualquer problema decorrente de decisões ou foco errados, serão visíveis em pouco tempo.

A adoção de uma nova tecnologia é uma área bem documentada de estudo (SULTAN e CHAN, 2000), porém, as tecnologias de desenvolvimento de software ágil são tão radicalmente diferentes de outras tecnologias de software, que impedem o valor da pesquisa dentro da área em relação aos estudos já realizados, dificultando assim a sua difusão. Outros importantes fatores que afetam a adoção de uma nova tecnologia são: estabilidade organizacional, conhecimento técnico já

adquirido e perceptível insegurança no trabalho (ABRAHAMSSON, 2002).

Estudos mostram que o desenvolvimento através de metodologias tradicionais não é utilizado na prática de acordo com sua exata definição, pois, são mecanismos que devem ser utilizados dentro dos detalhes e particularidades disponíveis no método respectivo. Como resultado, os desenvolvedores de software ficam céticos em relação a novas soluções que são aparentemente difíceis de serem compreendidas e forçam uma mudança radical com a promessa de solução para os fatores comumente encontrados em desenvolvimento de software.

Os métodos ágeis são efetivos e satisfatórios para muitas situações e ambientes, porém, poucos estudos validados podem ser encontrados para apoiar estas reivindicações. A evidência existente consiste principalmente em histórias de sucesso de profissionais que utilizam tais metodologias. Além disso, o freqüente surgimento de novos métodos ágeis provoca mais confusão ao invés de esclarecimento, pois, cada método usa seu próprio vocabulário e terminologia.

Um fator comum para as metodologias ágeis é o foco em pessoas como uma fonte importante de vantagem competitiva, pois, as estratégias humanas são difíceis de serem copiadas (PFEFFER, 1998 e MOLEIRO & LEE, 2001). Esta, porém, não é uma realização nova: desde 1990 idéias sobre o assunto já são levantadas, mostrando que o melhor modo para melhorar produtividade de software e qualidade é focalizar em pessoas. Assim, as idéias que métodos ágeis provocam não são novas, nem os fomentadores dos métodos reivindicam este direito (ABRAHAMSSON, 2002).

4 ADAPTAÇÕES

Programação extrema demonstra-se uma solução satisfatória em muitas situações enfrentadas pelas equipes de desenvolvimento, entretanto, como a maioria das novas tecnologias existentes, não pode ser considerada perfeita e por este motivo encontra-se em constante aprimoramento de suas regras, prático e princípio.

Um dos grandes problemas enfrentados em projetos que adotam XP é a resistência do fator humano. A correta e eficiente aplicabilidade da metodologia, depende consideravelmente das pessoas que compõem o projeto, sem as quais, nenhum resultado satisfatório pode ser obtido. Os problemas surgem tanto na equipe do cliente, como na dos desenvolvedores, entretanto, a equipe do cliente, apesar de resultar em alguns problemas, não torna inviável a aplicação da metodologia. Já, quando os problemas surgem com a equipe de desenvolvimento, de acordo com a gravidade do mesmo, a situação pode atingir a inviabilidade. Problemas como desenvolvedores que não aceitam as regras da XP, não concordam com a programação em pares, não aceitam a forma de planejamento, ou, não buscam um relacionamento bom com o grupo entre outros fatores, ou seja, não são capazes de abandonar o comportamento tradicional para o desenvolvimento de software, geralmente por estarem acomodados com uma situação crítica, porém conveniente.

Um dos fatores que apresenta problemas com conseqüências significativas devido ao fator humano deficiente é a programação em pares. A produtividade gerada pela programação em pares é surpreendente, porém, para isto, é necessário que os desenvolvedores que compõem a dupla estejam devidamente sincronizados com um único foco. Desenvolvedores dispersos ou desmotivados que são comumente encontrados em equipes de desenvolvimento tendem a ser um problema para o desenvolvimento de software, pois, inviabilizam uma das principais regras da XP, ou seja, a programação em pares. Iniciativas que abordem uma possível solução para este problema serão levantadas na seqüência do capítulo.

Outro fator que não atinge uma inviabilização do desenvolvimento, mas, pode influenciar em atrasos no cronograma, trata-se da situação de longas discussões

sobre o projeto, geralmente causadas todas as manhãs durante a reunião matinal que ocorre. Segundo outra regra da XP, tal reunião é apenas para colocar toda a equipe a par da situação do projeto e jamais para reuniões que se estendam consideravelmente, envolvendo discussões sobre assuntos diversos que acabam consumindo boa parte da manhã, que deveria estar sendo utilizada para a codificação. O tempo total gasto no projeto neste tipo de reunião pode ser significantemente negativo. Uma medida simples para resolver tal problema é citada no decorrer do capítulo.

Abrangendo outra regra da XP temos o código compartilhado. Geralmente isto é realizado através de alguma ferramenta que proporcione suporte para tal função, porém, pequenas equipes de desenvolvimento que estão iniciando sua produção de software através de uma metodologia ágil, tendem a criar certas desconfianças sobre o investimento em algum software neste sentido, pois não conseguem um apoio tecnológico que motive tal investimento. Ferramentas gratuitas no mercado costumam não atingir todos os pontos necessários, ou não fornecem uma estrutura de suporte confiável, não sendo raras as situações em que se perde muito tempo na tentativa de implantação de uma ferramenta como esta, deixando a equipe em uma situação complicada de como realizar tal prática sem um investimento inicial. Porém, obtendo o funcionamento correto exigido na situação, este é outro fator que através de uma pequena adaptação pode ser contemplado durante o tempo inicial de implantação do projeto.

Outro fator que implica em problemas seria a prática do ***refactoring***, ou seja, da reescrita de código de forma a melhorar seu funcionamento, que por vezes, acarreta em situações de desavença entre os desenvolvedores. Apesar do princípio de código coletivo, o desenvolvedor acaba adquirindo uma certa familiaridade com o código gerado pelo mesmo. A situação em que um desenvolvedor acessa um código anteriormente escrito por ele para adicionar alguma funcionalidade e acaba se deparando com um código totalmente diferente do original, incluindo até mesmo outra lógica de funcionamento, acaba conduzindo à necessidade de uma leitura atenta do código para a interpretação do mesmo e possível alteração. O tempo gasto e a insatisfação de ver algo alterado de sua codificação original pode gerar descontentamento e tempo perdido para interpretar novamente a funcionalidade do

código.

É evidente a necessidade de algum fator que otimize o entendimento deste novo código, pois, se o código é alterado, mas, a compreensão do mesmo pelo programador original é rápida, o mesmo acaba se conscientizando que tais mudanças eram realmente necessárias para otimizar o mesmo, adicionando a funcionalidade desejada rapidamente, de forma a não perder tempo ou sentir-se agredido com tal alteração.

Quanto tempo um desenvolvedor pode dedicar-se à construção de testes de código, até que este tempo passe a ser prejudicial ao cronograma do projeto? Esta é uma questão muito discutida. Outro detalhe seria a dúvida sobre o que deve ser testado, ou até que ponto deve ser testado. Unidades de teste é uma solução para melhorar o desempenho da codificação de testes de forma que não tornam os testes um obstáculo para o projeto e sim, algo extremamente produtivo e objetivo, tornando o processo muito mais fácil.

O questionamento de XP gerar ou não documentação é tomado como arma para desenvolvedores que não apóiam a iniciativa de metodologias ágeis. Entretanto, a afirmação de que XP não possui documentação é falsa, apenas, a documentação é gerada somente no momento em que se torna necessária. Porém, projetos podem ser adaptados para atingirem um certo grau de documentação de forma a facilitar o entendimento geral da equipe. Desenvolveu-se um modelo no qual XP pode ser aplicada de forma a não perder sua funcionalidade e paralelamente satisfazer aos elementos que precisam de alguma documentação complementar no projeto. Os passos para este desenvolvimento podem ser mais bem vistos na continuidade do capítulo.

Como se percebe, foram citados alguns pontos problemáticos da XP, alguns podem inviabilizar o projeto e outros apenas prejudicar a qualidade e o tempo do desenvolvimento do projeto. Adaptações já testadas e comprovadamente funcionais podem ser aplicadas nestas situações para evitar toda a problemática que algumas equipes utilizando a metodologia da XP acabam sofrendo.

Uma característica comum de usuários da XP seria uma tendência a adaptar o processo após adquirir um certo grau de conhecimento sobre o mesmo. Esta necessidade deve-se às diferentes características entre os projetos que a utilizam,

bem como, aos distintos pensamentos dentro de uma equipe de desenvolvimento.

A adaptação pode ocorrer em dois níveis:

a) os níveis individuais, que estabelece mudanças de micronível dentro de uma determinada tarefa, podendo não ser mais utilizada no decorrer do projeto, atinge a um desenvolvedor ou, ao par que estiver envolvido na codificação; ou

b) as adaptações no nível de grupo, consideradas de macronível, devem ser discutidas com todo o grupo e se considerada positiva, é incorporada à prática do projeto.

Algum nível de adaptação costuma ocorrer em projetos de XP, porém, a adaptação não implica diretamente nos princípios básico da XP, os mesmos formam a base do processo e continuam sendo utilizados dentro da adaptação.

Na seqüência, apresentam-se algumas das principais adaptações classificadas como macronível, que possuem, a característica de abordarem soluções, para os problemas que costumam surgir em grande parte das equipes, que iniciam o processo de desenvolvimento através da metodologia ágil XP.

4.1 Adaptação em Pair Programming (Programação em pares)

Uma das mais polêmicas práticas da XP é a programação em pares. Opiniões dividem-se no que diz respeito às vantagens e desvantagens de se programar em pares. Geralmente, as pessoas tendem a formular a idéia de que a produtividade de código diminui nesta situação, pois, dois programadores fazem o serviço de um e se estivessem em máquinas separadas desempenhariam o dobro da produção de código. Fato este, que não é verdade, pois, já foi estatisticamente comprovado um ganho de 15% no tempo de desenvolvimento na programação em pares (WILLIANS, 2000). Várias atividades são executadas paralelamente, sendo, a análise, o projeto, o ***refactoring***, o teste, a revisão do código, entre outras coisas, causando assim, uma produtividade muito maior em comparação a dois programadores separados.

Uma das maiores dificuldades de se implantar a PP (Programação em Pares), é a forte resistência apresentada pela equipe de desenvolvimento. Convencer desenvolvedores que já possuem um conhecimento previamente adquirido e que já trabalharam durante alguns tempo no desenvolvimento de software de maneira tradicional é uma tarefa que nem sempre obtém sucesso e portanto, umas das

causas que levam ao fracasso da implantação da programação em pares. A correta escolha das pessoas, com uma personalidade maleável é um fator importante no momento de integrar a equipe de desenvolvimento e quatro fatores devem ser considerados no momento da seleção, sendo: comunicação, bom relacinamento entre os participantes, confiança e concordância.

O fator comunicação é a característica de personalidade mais óbvia e essencial para o sucesso da programação em pares. Comunicação é claramente valiosa em qualquer ambiente de desenvolvimento, mas em um ambiente de programação em pares, a habilidade para comunicar é crucial. O par deve ser capaz de se comunicar efetivamente, corretamente e em ordem para uma ponderada análise dos méritos ou problemas de decisões diferentes, discutindo estratégias, testes a serem realizados, e erros identificados principalmente pelo navegante(co-piloto). Em essência, uma falta de comunicação entre o piloto e co-piloto, diminuirá o potencial para o par trabalhar em harmonia.

Quanto ao fator de bom relacionamento, pares que não estão confortáveis entre si, serão relutantes em oferecer sugestões, devido à possibilidade de ser ridicularizado. O medo de parecer ignorante perante o parceiro, faz com que diminua por parte do co-piloto, o número de propostas corajosas e idéias que são elementos essenciais em XP. Pares que estão confortáveis entre si, exploram sugestões intrigantes e estratégias interessantes com o conhecimento que possuem e compartilham de maneira natural e produtiva. Outro fator relacionado à sensação de bom relacionamento entre os pares diz respeito ao incômodo com indivíduos que têm padrões de trabalho diferentes, etiqueta moral e profissional (por exemplo: higiene pessoal).

Em relação à confiança, os desenvolvedores devem ser confiantes nas suas habilidades e dos respectivos parceiros. XP requer que os desenvolvedores inovem soluções ao longo da base de código de produção. O par deve ser confiante dentro de suas habilidades para somar funcionalidades novas prosperando dentro de uma reciprocidade. Pares nos quais falta confiança, tendem a focar o objetivo em resolver um problema de um código ruim, ao invés de abandonarem tal solução e criarem algo novo para sanar tal situação encontrada, influenciando diretamente numa má produtividade no que tange ao ***refactoring*** de código.

E por último, o fator concordância, a habilidade para chegar a um acordo, completa o quarteto das características de personalidade exigidas para a prática da programação em pares. A dificuldade de chegar a um acordo, mesmo entre um par confiante, com argumentos que possam convencer para a realização de uma determinada codificação, é alta nos pares de programadores. O propósito primário de trabalhar em pares é obter o melhor código possível, independente de quem tenha originado tal código. Bons pares de desenvolvimento devem discutir sugestões sem preconceito e principalmente sem interesse por sua origem e focar somente nos méritos da própria sugestão.

Uma equipe de desenvolvimento que possui estes quatro valores tem um prospecto de sucesso nas ações que realizam muito maior. Não se pode esquecer de outros valores que agregam uma melhor produtividade, como: criatividade ou atenção aos detalhes.

Ações que visam estimular uma aceitação por parte da equipe de desenvolvimento são necessárias.

Para obter-se uma satisfatória produtividade através da utilização das práticas da XP, torna-se necessário uma sinergia do conjunto de todas as práticas. Retirando-se alguma das práticas, o processo acaba sendo prejudicado. Entretanto, existem práticas que podem prejudicar de maneira pouco intensa, bem como, práticas que podem inviabilizar a produtividade da XP.

A programação em pares é uma das práticas de grande importância dentro da XP. Em contrapartida, para que se possa atingir todo o potencial no processo, é necessário uma conscientização de todos os elementos que compõem a equipe de desenvolvimento, sem a colaboração dos quais, é totalmente impossível executar a programação em pares.

Dentre as possíveis práticas que podem ser utilizadas para, de alguma forma, buscar um melhor relacionamento entre as duplas de programadores, estaria alguma atividade fora do ambiente de trabalho. Atividades recreativas que envolvam a equipe em um ambiente descontraído, também reforçam os laços entre as pessoas, facilitando assim, um melhor entrosamento entre desenvolvedores. Geralmente, atitudes como esta acabam resolvendo o problema de bons programadores, que se tratam de pessoas muito introspectivas, ou seja, de uma socialização difícil. O que

ocorre em muitas situações é o receio de ofender o colega, ou, de ser ridicularizado por alguma colocação. Porém, não é suficiente um bom relacionamento entre a equipe, conflitos no momento da prática da programação em pares ainda são um problema.

Algo que é muito comentado entre as pessoas que tentaram exercer a programação em pares é o fato de que um programador executa toda a codificação, enquanto outro apenas fica olhando sem demonstrar qualquer interesse no que está acontecendo. Para evitar este comportamento existem determinadas ações que podem ser implantadas no processo, sendo:

Implantando uma política, na qual o programador que possui menor experiência seja o piloto, ou seja, a pessoa que tem o controle do teclado e mouse para a codificação. Desta forma, evita-se que o programador experiente faça tudo sozinho. Também implica em ganho de tempo para treinamento, pois diante da necessidade, o programador menos experiente acaba fazendo um curso paralelo na situação, capacitando-se sem a necessidade de parar a produtividade para isto.

A troca entre piloto e co-piloto também dever ser um comportamento adotado. Desta forma, o co-piloto sente-se na obrigação de prestar atenção na codificação, pois, a qualquer momento ele poderá assumir o controle para continuar o processo de desenvolvimento e seria uma situação constrangedora não saber continuar a seqüência do que estava sendo realizado. Outra prática neste sentido, seria de promover reuniões diárias, nas quais os co-pilotos de cada dupla vão explanar sobre o que está acontecendo em cada equipe, caso ele não tenha se interessado, não terá como saber o que está acontecendo.

Algo comum que acontece nas equipes é a existência de um programador que julga ter todo o conhecimento necessário para desenvolvimento de código, que é contra qualquer mudança antes mesmo de saber qual seria e que, principalmente, aprendeu a programar já há alguns anos e não admite ter que mudar sua forma de programação. É importante não forçar ninguém a realizar algo que não quer, então nesta situação a prática da programação em pares pode ser suspensa para que este programador trabalhe de acordo como considere melhor. Provavelmente, o mesmo irá começar a comparar o desempenho das duplas de programadores com o seu desempenho, constatando assim, que sua produtividade é inferior trabalhando

sozinho na codificação e aos poucos, adotando programação em par.

4.2 Motivação da equipe de desenvolvimento

Geralmente, o início de um projeto está repleto de motivação, as pessoas encontram-se numa fase de entusiasmo, tudo ainda é novidade e as promessas são sempre otimistas. Entretanto, com o passar do tempo, uma forte tendência ao marasmo e desânimo começa a tomar conta do ambiente. Basta um membro da equipe começar a reclamar da situação para que o baixo astral se espalhe e conseqüentemente, a produtividade baixe de maneira significativa.

Uma pessoa desmotivada não irá obedecer às regras da XP e de certa forma, tentará expressar atitudes contrárias para demonstrar sua insatisfação, como uma forma inconsciente de chamar a atenção para o momento que está passando, tentando desta forma encontrar uma solução para seu problema.

Sabe-se que se as regras podem ser adaptadas, ou, até mesmo abolidas, de acordo com o grau de importância que possuem dentro da XP, entretanto, um membro trabalhando totalmente contra todas ou a maioria das práticas acaba implicando em um fracasso da aplicação do processo da XP.

Um membro da equipe que esteja desmotivado torna-se um grande problema, pois, além de não ter mais o rendimento e a dinâmica necessária na equipe, tende a forçar uma situação que “contagie” os demais para a mesma situação que ele. Retirar este membro da equipe não é exatamente uma boa saída, pois, uma eventual demissão pode causar má impressão no resto do grupo, o que gera comentários e possivelmente desencadeia um processo de descontentamento, insegurança e inevitavelmente a desmotivação.

A melhor maneira de evitar a desmotivação que pode prejudicar tanto uma equipe é não deixar a mesma aparecer em nenhum momento, tentar de alguma forma manter o alto astral generalizado do início do projeto durante todo o decorrer do mesmo.

Buscando exatamente a execução desta prática, podem-se destacar algumas ações a serem realizadas que tem por característica motivar a equipe, sendo:

Um planejamento salarial devidamente elaborado e passado para o conhecimento da equipe logo no início do projeto é fundamental. Se a equipe

trabalha sem saber o que acontecerá no dia de amanhã, uma insegurança vai sendo criada a cada dia, um descontentamento por estar muito tempo com o mesmo salário e não ter uma previsão de aumento é um problema. Muitas vezes, a posição do membro da equipe não dá liberdade para iniciar um diálogo sobre este assunto com o pessoal que está financiando o projeto, acarretando comentários dentro da equipe que contaminam com uma carga negativa todos que se encontram na mesma situação.

Já, se a equipe possui datas e valores de reajuste que irão ser executados, por mais baixos que sejam os reajustes, desde que constantes, não geram motivos para reclamações e a motivação é mantida.

Outro fator que mantém os ânimos elevados é a presença constante de alimentos no ambiente de desenvolvimento. Café, bolachas, salgadinhos e até mesmo uma pizza com a equipe ao final de toda jornada de 40 horas semanais, faz com que o pessoal tenha um melhor entrosamento, além de estimular a todos. É impressionante o resultado que causa um café entre uma codificação e outra. O stress gerado durante o tempo de criação é consideravelmente diluído durante o lanche rápido.

Um projeto da PNUD/ONU – Programa das Nações Unidas para o Desenvolvimento com a Secretaria da Fazenda do Estado de São Paulo, o qual desenvolve um projeto de modernização tributária através da web, utilizou amplamente o recurso da comida presente a todo o momento, obtendo resultados satisfatórios de moral alto dentro da equipe (MULIANA, 2002).

Um dos tópicos levantados é que a motivação fornece uma capacidade imensa de previsibilidade no projeto, ou seja, se conseguir atingir uma motivação de 100% em toda a equipe e manter esta motivação durante o andamento do projeto, a estimativa de datas iniciais para conclusão das etapas podem ser cumpridas sem problemas, ao passo que quanto mais desmotivada, o cronograma tende a ser prejudicado com atrasos significativos. Infelizmente, a motivação nestes níveis dificilmente pode ser atingida, porém, o cronograma sofre pequenas alterações com uma motivação elevada.

4.3 Reuniões curtas

Esta é outra prática que possui uma dificuldade de ser realizada da maneira correta, o que, acaba tornando-a improdutiva dentro do processo da XP.

Uma sugestão é tornar o ambiente das reuniões o menos confortável possível. Jamais utilizar a própria sala de desenvolvimento para esta reunião, pois, a tendência é que tenha um prolongamento indesejado.

Retirar todas as cadeiras da sala e manter apenas uma mesa alta com café sobre ela é uma excelente escolha para acomodar uma reunião matinal da XP. Estando em pé, a tendência é ser o mais breve possível, o os desenvolvedores irão pensar muito bem antes de iniciar qualquer comentário que leve todos a ficar muito tempo nesta situação desconfortável. A regra é a seguinte: se você está sentindo necessidade de sentar, provavelmente a reunião deve estar no momento certo de ser finalizada.

4.4 Código compartilhado sem ferramenta

Outro aspecto relevante está relacionado à dificuldade que as equipes encontram em definir um gerenciador de código, para exercer a prática de código compartilhado, ou seja, cada desenvolvedor tem direito a alterar o código produzido por outro, com o propósito de melhorar o mesmo, retirar redundâncias, ou adaptar novas funcionalidades, desde que não altere a função inicial abordada pelo mesmo.

Geralmente são utilizados softwares como o CVS (Sistema de Controle de Versões) para esta tarefa, entretanto, o custo para aquisição de softwares que desempenhem tal tarefa costuma ser elevado, e as ferramentas consideradas *freeware* (gratuitas), disponíveis no mercado, não costumam atender a todas as necessidades que o projeto requer, ou então, não dispõem de recurso algum de suporte em caso de algum problema acontecer.

Ao iniciar o projeto, não é cabível uma interrupção por problemas nas escolhas das ferramentas utilizadas e isto gera atrasos que vão contra o processo da XP.

Equipes pequenas e que geralmente estão iniciando no processo da XP, tendem a não arriscar em grandes investimentos para a prática da mesma, tentando assim, diminuir o risco do custo em caso de um fracasso do projeto. Para tanto,

pode-se aplicar uma estrutura de compartilhamento um pouco mais trabalhosa que a gerada através de um software gerenciador, mas, que é funcional o suficiente para os primeiros meses de teste do projeto.

O acompanhador, ou um dos desenvolvedores, que passa a assumir um papel importante, ficando a seu cargo, o gerenciamento do código compartilhado, deverá estar sempre muito atento às situações e a cada final de dia, o mesmo é incubido de realizar a combinação do código produzido naquele dia e fixar no quadro anotações que descrevem o que foi feito, de maneira resumida. É importante que não falhe este processo nenhum dia, pois, no dia seguinte, antes de formular os pares para o desenvolvimento, convém que todos observem as alterações realizadas no dia anterior e caso necessário alterem algum código já produzido.

Esta tarefa toma um grande tempo, portanto, é preferível que o acompanhador seja o responsável, já que de qualquer forma, terá que saber o que está acorrendo em todo o projeto. Caso um programador fique responsável por tal tarefa, é interessante ressaltar que o desenvolvimento relacionado a este membro ficará sempre debilitado.

4.5 *Refactoring*

O ***refactoring*** é certamente uma das práticas que beneficiam em muito o processo da XP. É o responsável por uma constante melhora do código produzido, otimizando assim, a estrutura geral do projeto.

O ***refactoring*** pode ser executado durante a própria codificação. Na estrutura de pares, o co-piloto pode visualizar determinados fatores que passaram despercebidos pelo piloto, sugerindo assim, uma alteração e conseqüentemente o ***refactoring*** do código em questão. Entretanto, existe a possibilidade de ***refactoring***, já que se trabalha com o código compartilhado, de um outro par de desenvolvedores. Isto acarreta em uma certa confusão caso o desenvolvedor original resolva voltar àquele código para realizar algo: perde-se um certo tempo para compreender as alterações realizadas e de acordo com o número de situações como esta, a soma do tempo perdido para a nova compreensão do código passa a ser um fator não desejado na equipe.

Uma solução para esta situação, que aparentemente implica em um maior

tempo, mas que, porém, em longo prazo contribui para uma melhor situação, seria a necessidade de adicionar comentários em todo código alterado, mantendo as linhas sobre forma de comentário e adicionando algum texto que justifique ou explique determinada alteração. Desta forma, o desenvolvedor terá uma rápida compreensão ao ler o código escrito, ficando muito mais rápido para alterar, bem como, fornecendo uma confiança de não estar alterando nada de forma errada.

O código pode ser limpo destes comentários antes de ser liberado um release, ou, pode-se manter o mesmo caso seja necessária uma nova alteração após os testes do cliente.

4.6 Testes Automatizados

Kent Beck compara XP com um motorista dirigindo um carro, ou seja, o motorista precisa guiar e fazer correções constantes para se manter na estrada (BECK, 2000). Alguém precisa delinear o curso, estabelecer os marcos, manter uma estrutura de progressão, e talvez até mesmo consultar, em certos momentos a direção correta. Infelizmente, isto se torna muito complicado devido a necessidade da velocidade que deve ser obedecido. Testadores de XP têm que dirigir na pista com esta velocidade. São necessários testes automatizados de peso leve, bem como ferramentas de teste que agilizem este processo. A adaptação aqui sugerida para este problema, surgiu de experiências obtidas com automatização de testes de aceitação durante um período de 10 meses em uma companhia de ***Java outsourcing*** que usa XP para todo o desenvolvimento de software (CRISPIN, 2002). Foi realizada com uma equipe de até nove desenvolvedores e um testador. Os fatores destacados durante este período foram:

• Por que automatizar testes de aceitação;

• Como projetar testes automatizados para uma baixa-manutenção e autoverificação;

• Como codificar e implementar testes automatizados rápidos e eficientes;

• Selecionar ferramentas de desenvolvimento para auxiliar em testes com automatização;

• Como aplicar os valores de XP para testes de automatização;

No que diz respeito ao porque automatizar, simplesmente não se pode dispor

desta funcionalidade, embora a automatização de testes de unidade seja citada em XP, o caso para automatização de teste de aceitação, não é tratado de maneira eficiente. A inerente “desordem” de testes de aceitação e a dificuldade de automatização por parte dos usuários provavelmente é a principal razão para isto.

Uma das razões dos testes de aceitação estarem deficientes é porque a taxa de testes aprovados esperada não é de 100% até o fim da repetição, quando todas as histórias já foram implementadas. Antes disso, é necessário julgar seu progresso baseado em taxas parciais de aprovação dos testes.

Quanto à necessidade de automatizar os testes de aceitação, a idéia dos testes é uma constante execução dos mesmos sobre o código gerado a cada situação de compilação do mesmo, isto requer um certo tempo para ser realizado e os resultados geralmente não são satisfatórios logo nas primeiras tentativas, levando o desenvolvedor a um ato repetitivo de testes durante sua jornada diária de trabalho. A soma de todos estes tempos de verificação acaba sendo significativa no total do projeto, induzindo o desenvolvedor a executar tais testes, somente ao término de cada repetição.

No que consta a implantação de uma automação de testes citada no capítulo 3, desde que corretamente aplicada, garante um tempo extra para a codificação. Provê uma forma de resolver problemas não cobertos pelos testes de unidade durante a semana de 40 horas, ao invés de um impacto mais drástico ao término de cada repetição. Também possibilita tempo para criar cuidadosamente os testes que não podem ser automatizados.

Respondendo a quem automatiza, os testes de aceitação são responsabilidade de toda a equipe que compõem o jogo da XP, incluindo o lado do cliente. A automatização pode ser feita teoricamente por um desenvolvedor ou pelo cliente, porém, a prática demonstra que é melhor uma pessoa dedicada exclusivamente para esta função, sendo a mesma da equipe de desenvolvimento.

Já no que diz respeito a quais ferramentas utilizar, depende diretamente de dois fatores, o primeiro seria de quanto tempo a equipe dispõe para o desenvolvimento, e o segundo de quanto está disposta a investir, ou seja, qual é o orçamento disponível para o projeto. Isto influencia diretamente na escolha de adquirir um produto já existente no mercado, ou a própria equipe desenvolver sua

ferramenta de automatização de testes.

Uma ferramenta já existente e adotada com certo sucesso em equipes de desenvolvimento seria a jUnit que está disponível gratuitamente no endereço http://www.junit.org. A mesma realiza um excelente trabalho com testes de unidade. Um ponto desfavorável é que a mesma implica na obrigação de desenvolvimento em Java, porém, pouco se conhece sobre equipes utilizando XP que não estejam com seu desenvolvimento baseado na linguagem Java.

Focando no que pode ser automatizado, esta situação deve ser avaliada de acordo com o tamanho do projeto, pois, projetos grandes, considerados de alto risco, devem ter sua estrutura de testes automatizados totalmente coberta, pois, existe tempo e necessidade de se proceder desta forma. Já, ao contrário em pequenos projetos, não é interessante perder tempo automatizando e testando continuamente em detalhes a estrutura de código, pois, se acaba dedicando mais tempo para esta ação, do que, ao projeto propriamente dito.

Em certos momentos se deve deixar o cliente decidir a prioridade, de acordo com o tempo disponível para a conclusão do projeto, em relação à seguridade que os testes podem proporcionar para o código final.

Conceituando o que são os testes de unidade, pode-se descrevê-los por pequenas ações que não exigem a codificação de um teste de aceitação. Por exemplo, em um caso de cadastramento, no qual a única mudança seria no banco de dados, o desenvolvedor pode ir diretamente consultar o banco de dados para verificar as mudanças, sem precisar codificar um teste para realizar tal tarefa. Isto tende a otimizar certos processos considerados de uma complexidade de verificação menor.

- ✓ Princípios de XP aplicados aos testes de automatização.

Alguns dos princípios da XP podem ser aplicados sobre os testes de automatização, vejamos alguns:

Tabela 07 Princípios aplicados sobre os testes de automatização.

Princípio	Descrição
Princípio 1	Planejamento dos testes de forma que não contenham nenhum código duplicado, otimizando ao máximo possível a existência dos mesmos.
Princípio 2	Separação do código do projeto do código dos testes.
Princípio 3	Verificação apenas dos critérios mínimos para obter a resposta, considerando que mínimo não significa insuficiente: o teste deve satisfazer todo o conjunto de fatores que implicam em uma possível falha.
Princípio 4	Reescrita contínua dos testes automatizados, combinando, dividindo, somando módulos, interfaces de módulo variáveis ou comportamentos, evitando sempre que possível qualquer duplicação.
Princípio 5	Desenvolvimento e revisão dos testes utilizando a programação em pares.

A automatização de testes não é algo de custo baixo, porém, o desenvolvimento dos próprios testes de aceitação pode acarretar em um tempo de projeto muito maior. A utilização de ferramentas gratuitas ou não, otimiza a codificação. Implantar a automatização de testes em fatores de risco críticos e só então, ir gradativamente abrangendo os demais, também garante uma melhor performance de cronograma. É importante estar sempre melhorando os testes, reescrevendo de forma a tornar seu impacto o menor possível no projeto.

No que se refere aos testes de aceitação, pode-se observar o exemplo seguinte, desenvolvido por Kent Beck e Erich Gamma, que demonstra de maneira detalhada o processo de testes através da JUnit. Exemplificando como escrever um teste de aceitação, o modo mais simples é como uma expressão em um depurador. Pode-se mudar expressões sem recompilar, bem como, optar por aguardar o que escrever após observar os objetos funcionando. Também se pode escrever expressões de teste como declarações que imprimem ao fluxo de produção padrão. Ambos os estilos de testes estão limitados porque exigem julgamento humano para analisar os resultados obtidos.

Testes de JUnit não exigem julgamento ou interpretação humana, e são fáceis de executar vários ao mesmo tempo. Diante da necessidade de testar algo, o procedimento seria:

1- Criar uma instância de ***TestCase***;

2- Criar um ***constructor*** que aceita uma ***string*** como um parâmetro e passa isto para a superclasse;

3- Anule método de ***runTest*** ( );

4- Quando você quiser conferir um valor, chame ***assertTrue*** () e passe um

boolean que terá valor verdadeiro se o teste tiver sucesso.

Por exemplo, testar que a soma de dois valores com a mesma moeda corrente contém um valor que é a soma dos valores dos dois valores respectivos, escreva:

```
public void testSimpleAdd() {
    Money m12CHF= new Money(12, "CHF");
    Money m14CHF= new Money(14, "CHF");
    Money expected= new Money(26, "CHF");
    Money result= m12CHF.add(m14CHF);
    assertTrue(expected.equals(result));
}
```

Caso seja necessário escrever um teste semelhante para um já escrito, ao invés disto, escreva uma ***fixture***. Quando for necessário executar vários testes, a criação de uma ***suite*** de testes é o mais recomendado. Estes processos são mais bem definidos na seqüência:

Na situação de dois ou mais testes que operam nos mesmos grupos ou em grupos semelhantes de objetos, os testes são executados com um ***background*** conhecido de objetos. Esta base conhecida de objetos é denominada ***fixture***. Quando se escreve testes, freqüentemente observa-se que o gasto com o tempo de codificação é muito maior em relação ao fato de realmente testar os valores de fato. Até certo ponto, pode-se fazer a escrita do código de ***fixture*** de maneira mais fácil, prestando uma atenção cuidadosa aos ***constructors*** que estão sendo escritos. Freqüentemente, a utilização de uma mesma fixture para vários testes poderá ser utilizada. Cada caso enviará mensagens ligeiramente diferentes ou parâmetros para a ***fixture*** e irá conferir os diferentes resultados obtidos.

Os passos para a criação de uma fixture comum são descritos por:

1- Criar uma subdivisão de classe de ***TestCase***;

2- Criar um constructor que aceita uma ***string*** como um parâmetro e passe isto para a ***superclass***;

3- Somar uma variável de exemplo para cada parte da instalação;

4- Ativar ***setUp***( ) para inicializar as variáveis;

5- Ativar ***tearDown*** ( ) para libertar qualquer recurso permanente alocado em

setUp ( );

Por exemplo, escrever vários casos de teste que trabalhem com combinações diferentes de 12 Francos suíços, 14 Francos suíços, e 28 Dólares de EUA. Primeiro criar uma ***fixture***:

```
public class MoneyTest extends TestCase {
   private Money f12CHF;
   private Money f14CHF;
   private Money f28USD;
   protected void setUp() {
      f12CHF= new Money(12, "CHF");
      f14CHF= new Money(14, "CHF");
      f28USD= new Money(28, "USD");
   }
}
```

Após a conclusão da ***fixture***, podem-se escrever tantos casos de teste quantos necessários.

Escrever um caso de teste sem uma fixture é simples - ativando ***runTest*** () em uma subdivisão de classe anônima de ***TestCase***. Escrevem-se casos de teste para uma ***fixture*** do mesmo modo, compondo uma subdivisão de classe de ***TestCase*** para seu grupo de código e fazendo subdivisões de classe anônimas, para os casos de teste individuais.

JUnit provê um modo mais conciso para escrever um teste através de uma ***fixture***, sendo:

1- Escrever o método do caso de teste na classe de ***fixture***. Considerando que deve ser declarado como público, ou não pode ser invocado por reflexão.

2- Criar uma instância da classe ***TestCase*** e passar o nome do método de caso de teste para o constructor.

Por exemplo, testar a adição de um valor em dinheiro em uma ***MoneyBag***:

```
public void testMoneyMoneyBag() {
   // [12 CHF] + [14 CHF] + [28 USD] == {[26 CHF][28 USD]}
   Money bag[]= { f26CHF, f28USD };
   MoneyBag expected= new MoneyBag(bag);
```

assertEquals(expected, f12CHF.add(f28USD.add(f14CHF)));

}

Criar uma instância de ***MoneyTest*** que executará este caso de teste da seguinte forma:

new MoneyTest("testMoneyMoneyBag")

Quando o teste é executado, o nome do teste é utilizado para observar o método que está rodando. Uma vez que se possui vários testes, a melhor forma de trabalho é a organização através de uma suíte. Para suprir o problema de testes rodando juntos pode-se observar o código para executar um teste a cada momento, em relação ao código de testes simultâneos:

Para rodar um teste único:

TestResult result= (new MoneyTest("testMoneyMoneyBag")).run();

Para criar uma ***suite*** com dois testes rodando juntos:

TestSuite suite= new TestSuite();

suite.addTest(new MoneyTest("testMoneyEquals"));

suite.addTest(new MoneyTest("testSimpleAdd"));

TestResult result= suite.run();

Outro modo é deixar JUnit extrair uma suite de um ***TestCase***. Deste modo é passado a classe do ***TestCase*** para o ***constructor*** de ***TestSuite***.

TestSuite suite= new TestSuite(MoneyTest.class);

TestResult result= suite.run();

TestSuites não têm que conter apenas ***TestCases***. Podem conter qualquer objeto que implementa a interface de Teste. Por exemplo, pode-se criar dois ***TestSuite*** distintos e depois executar os mesmo juntos através de uma ***TestSuite*** que contém ambos:

TestSuite suite= new TestSuite();

suite.addTest(Kent.suite());

suite.addTest(Erich.suite());

TestResult result= suite.run();

Concluído o fato de possuir uma suite de testes, o passo seguinte é utilizar esta estrutura. ***JUnit*** provê ferramentas para definir a suite a ser executada e exibir seus resultados, o fato é tornar a suite acessível para uma ferramenta de

TestRunner com uma suite de método estático que retorna uma suite de testes.

Por exemplo, para fazer uma suite de ***MoneyTest*** disponível para um ***TestRunner***, acrescente o código seguinte a ***MoneyTest***:

```
public static Test suite() {
    TestSuite suite= new TestSuite();
    suite.addTest(new MoneyTest("testMoneyEquals"));
    suite.addTest(new MoneyTest("testSimpleAdd"));
    return suite;
}
```

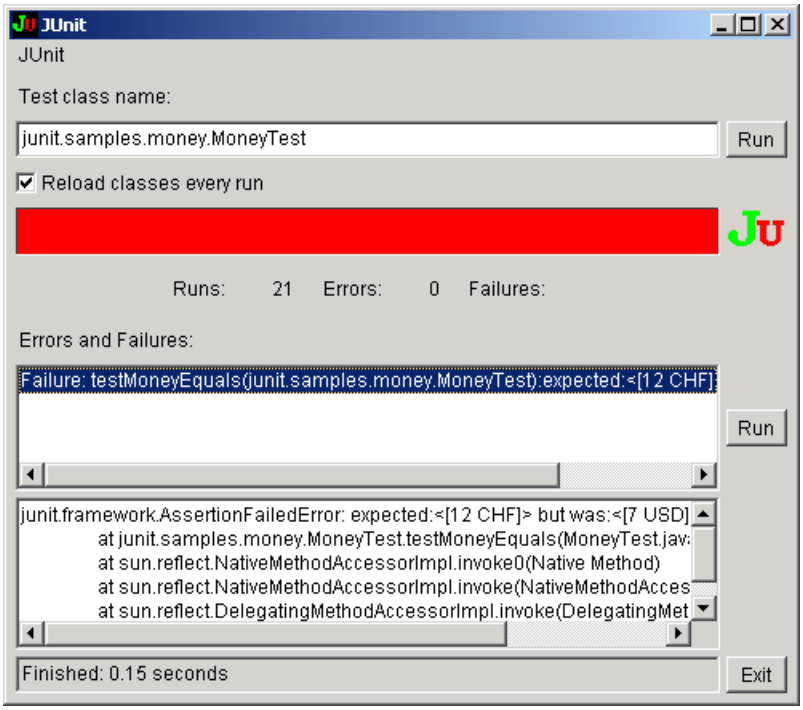

Figura 17 Ferramentas gráficas para testes em JUnit
Fonte: JUnit Web Site

Existem algumas ferramentas gráficas para executar o processo de testes. JUnit possui as seguintes estruturas para isto, ***junit.awtui.TestRunner*** ou ***junit.swingui.TestRunner.***

4.7 Adaptando Modelos de Processo para XP (documentação)

Uma das questões freqüentemente levantadas por críticos em relação a XP, seria a não geração de uma documentação como a dos métodos tradicionais, ou

seja, uma descrição de regras e atividades e demais fatores, de forma a permitir uma adaptação fácil para organizações diferentes.

Este problema pode ser sanado através de uma adaptação, executando o projeto através de uma estrutura produzida em Fraunhofer, IESE que focaliza este problema provendo uma sistemática e definindo facilmente um processo adaptável de geração de documentação para um modelo XP. Este modelo é então usado como base para o geração de um guia de processo eletrônico acessível via web (EPG).

SPEARMINT é uma ferramenta de modelagem de processo que tem sido desenvolvido por ***Fraunhofer*** IESE nos últimos 8 anos (BUNSER, 2002). Sua intenção é fazer processos complexos de software serem facilmente compreensíveis, através de conceitos naturais e uma notação gráfica semelhante a UML, provendo orientação de processo para participantes de projeto. SPEARMINT, pode gerar um Guia de Processo Eletrônico (EPG) de qualquer modelo de processo. Baseado na web, dedica-se a prover meios dedicados a encontrar qualquer informação pertinente ao processo da maneira mais rápida possível.

No projeto desenvolvido através da ***Fraunhofer*** IESE, SPEARMINT foi usado para modelar o XP, gerando um processo de XP de forma a obter um guia eletrônico de todo o processo.

A idéia é de fornecer uma completa visualização do processo da XP, de forma que sirva como base de consulta geral para equipes de desenvolvimento entenderem melhor como todo o processo funciona. A estrutura criada foi baseada na série de livros sobre XP, além de consultas a vários sites na internet, bem como de muitas conversas em listas de discussão. Também, contou com o auxílio de pesquisa realizada dentro da própria instituição ***Fraunhofer*** IESE localizada na Alemanha. O processo originado considera a estrutura base da XP, podendo ser adaptado de acordo com a necessidade encontrada por determinada equipe de desenvolvimento. O exemplo completo do processo pode ser observado na seqüência.

Em relação à adaptação citada no capítulo 3 sobre a origem de uma documentação mais abrangente, pode-se observar um exemplo gerado através do processo, utilizando como objeto de estudo a própria XP. O primeiro nível de

abstração é mostrado na Figura 18, que é uma representação vívida de atividades, artefatos, e fluxo de produto entre eles. Tudo começa com duas atividades: o cliente descreve o sistema, e as regras do jogo estão definidas.

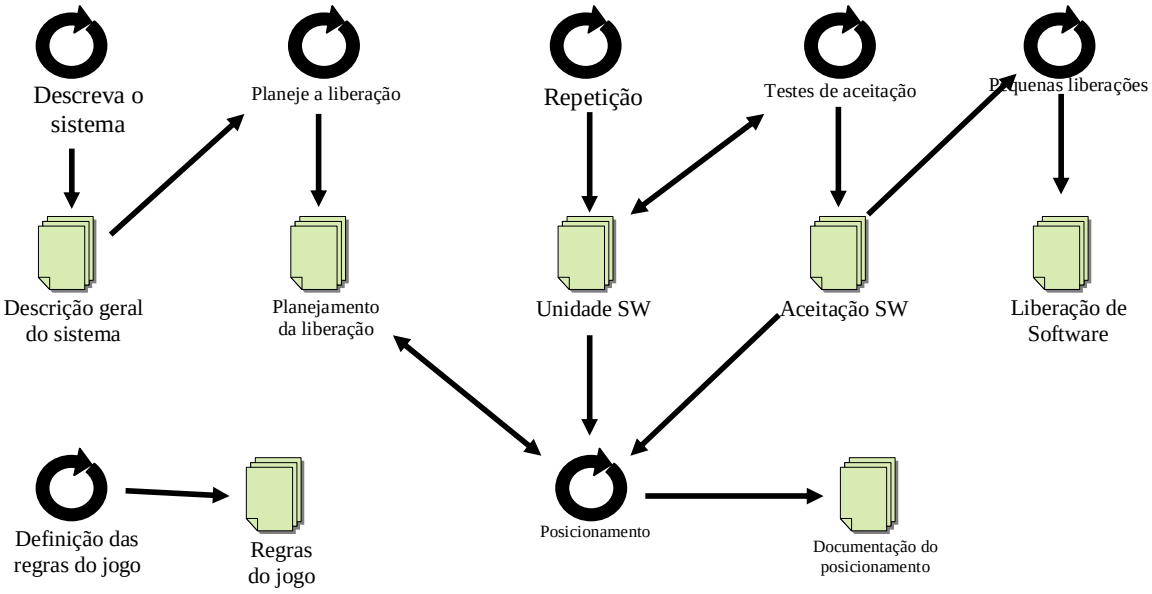

Figura 18 Primeiro nível de abstração

Enquanto o cliente está descrevendo o sistema, os desenvolvedores, escrevem “histórias de usuário”. Se por qualquer razão uma história de usuário parece ser muito grande, deve ser dividida em duas ou mais história de usuário.

Depois que o sistema é descrito, o acompanhador tem que manter conhecimento do nível de progresso em relação ao plano original e notificar o gerente. Deste modo, o gerente com certeza contará com informações seguras a toda hora para tomar decisões.

O segundo passo é planejar a liberação do software pelo processo: de acordo com as exigências do sistema, ou seja, os ***releases***. A equipe de desenvolvimento poderá dar uma estimativa do tempo e recursos que levará para desenvolver as histórias de usuários. Se o cliente concorda com os prazos e os requisitos apresentados, a equipe inicia o desenvolvimento do software; caso contrário, eles têm que negociar, quais características serão desenvolvidas para o primeiro ***release***, ou liberação do sistema.

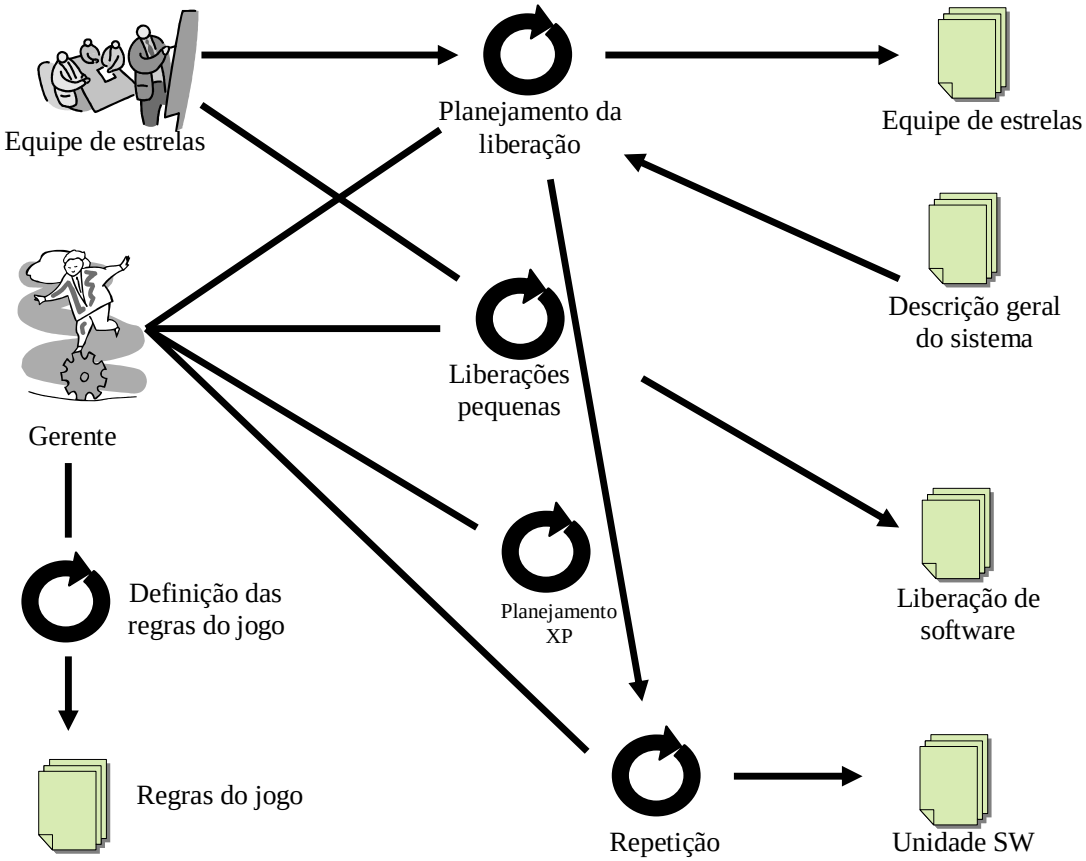

Figura 19 Segundo nível de abstração

Então, ocorrem quebras realizadas pela equipe de desenvolvimento, das histórias de usuário em “tarefas”, que são unidades para codificar. As tarefas podem ser divididas futuramente em mais tarefas ou combinações. Este passo corresponde ao próximo nível de abstração, como parte, da atividade chamada repetições.

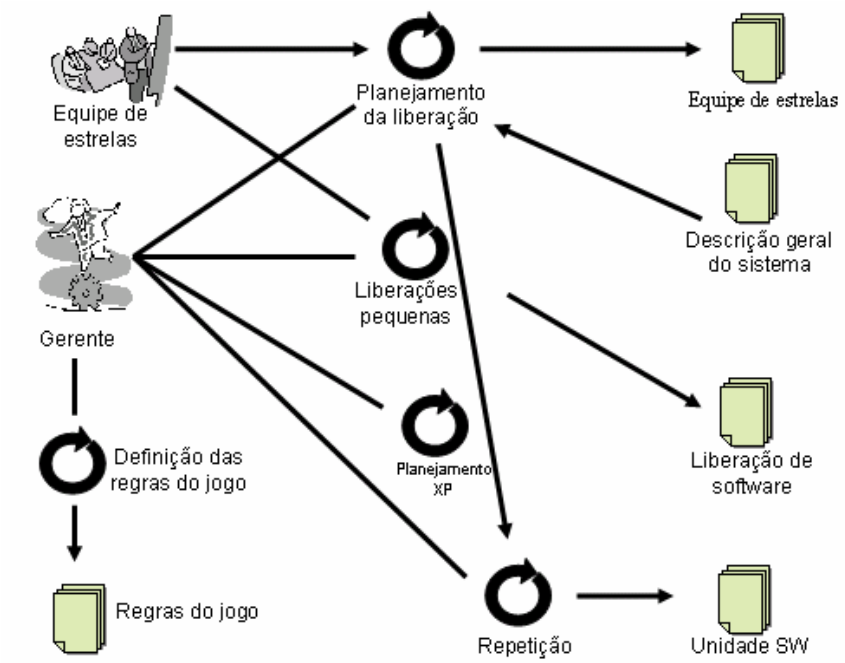

Figura 20 Terceiro nível de abstração

Uma vez que as histórias de usuário são implementadas e devidamente testadas, o cliente executa os testes de aceitação. Caso seja aprovado pelo cliente, é feita uma liberação pequena e a equipe está pronta para ir pela segunda liberação, onde é esperado que o cliente inclua todas as características que não puderam ser incluídas na liberação atual. Este mesmo nível de abstração, do ponto de vista do gerente seria como na Figura 19 e como perspectiva do cliente como na Figura 20.

As regras do jogo são a essência da XP. Estas regras têm, porém, sido por muito tempo mudadas para práticas mais elaboradas, as quais, conduziram à solução de alguns dos principais problemas que atingem a indústria de software hoje, como a incapacidade de prover o cliente com tempo seguro e estimativas de custo. Alguns dos artefatos produzidos por esta atividade estão contidos na Figura 21.

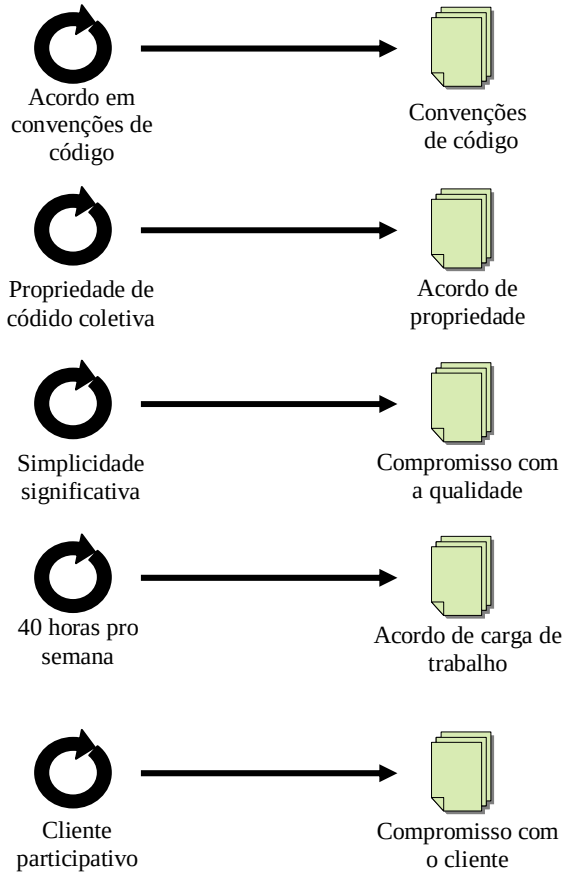

Figura 21 Quarto nível de abstração

A equipe de desenvolvimento tem que concordar nas convenções de codificação a serem usadas, como nomear protocolos. Os desenvolvedores possuem todo o código: eles são livres para fazer mudanças em qualquer parte do código para obter uma melhora, como também simplificar qualquer coisa que parecer redundante.

Um das características que fazem SPEARMINT uma poderosa ferramenta, é a flexibilidade para deixar o usuário ter diferentes visões do processo: uma visão por papéis dentro do processo, além da visão geral do mesmo. Desde que os papéis executem atividades diferentes, não há nenhuma necessidade para mostrar todas as atividades que um papel particular não está executando. O usuário também adquire a versão gráfica do modelo de processo no EPG (Guia de processo eletrônico) é possível ver as interconexões das atividades e os artefatos produzidos por ele.

Certamente o processo da XP é muito mais complexo do que o apresentado

aqui, entretanto, para se ter uma idéia melhor, observe a Figura 22, na qual é demonstrada a impressão do que o guia de processo da XP, isto é, o EPG, pode ser reproduzido. No exemplo, a orientação em atividade de repetições é mostrada na figura 22.

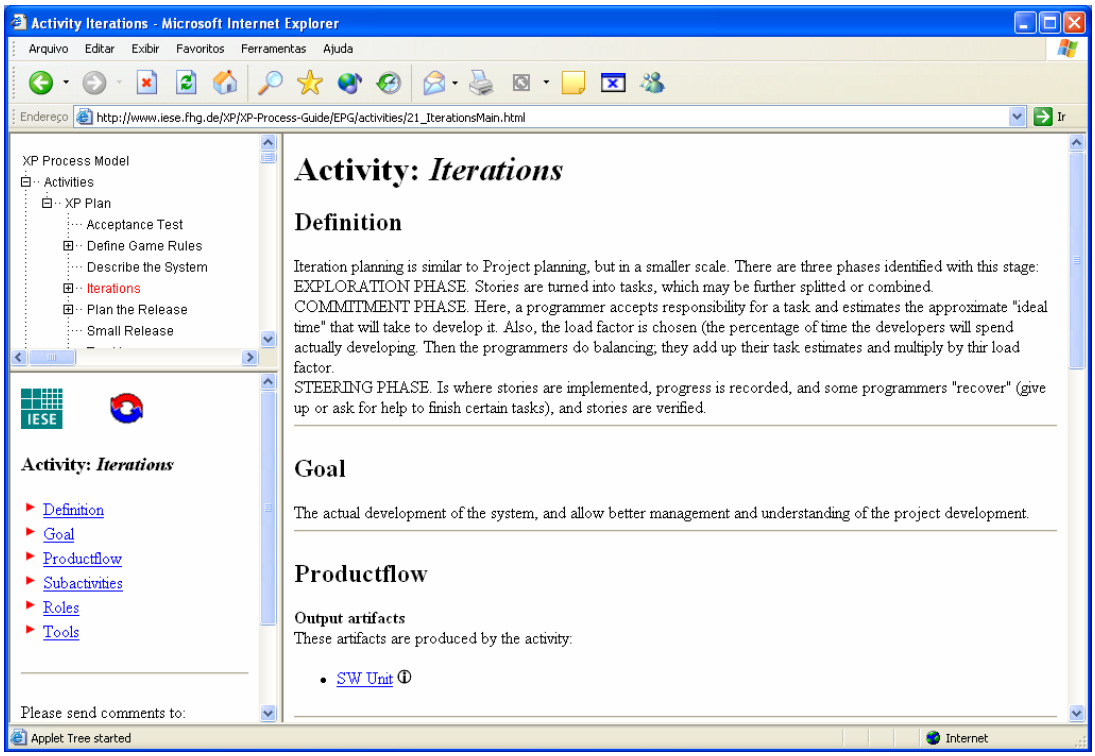

Figura 22 Exemplo do EPG

Outra vantagem desta aproximação é que o modelo gerado pelo SPEARMINT pode ser adaptado facilmente para as necessidades de organizações diferentes. Assim, usando as EPG de criação automática de SPEARMINT, o guia de processo sempre reflete a versão atual do processo, sendo uma ajuda ao desenvolvedor, a ser informado sobre a versão atual.

O SPEARMINT, neste caso foi utilizado para modelar o próprio processo da XP, entretanto, poderá ser utilizado para gerar a mesma documentação sobre projetos que estejam sendo desenvolvidos através da XP.

5 CONCLUSÕES E RECOMENDAÇÕES

Os processos de desenvolvimento de software, em menor ou maior utilização, são necessários para projetos que possuem um porte profissional. O desenvolvimento comercial sem regras ou práticas bem definidas é inviável e tende ao fracasso.

O mercado de TI (Tecnologia da Informação), na área de desenvolvimento de software está passando por uma crescente evolução e dentro deste contexto, tecnologias de desenvolvimento que não foquem em um retorno rápido dos investimentos, considerando a qualidade deste retorno e não simplesmente o tempo de resposta, estão, de certa forma, sendo abolidas dos projetos de desenvolvimento de software.

As adaptações apresentadas demonstram soluções para problemas que geralmente ocorrem nas equipes de produção que optaram pela utilização de metodologias ágeis, mais especificamente programação extrema. Tais problemas, de certa forma, acabam contribuindo para o atraso do cronograma. Perde-se muito tempo pensando em como solucionar esta ou aquela dificuldade encontrada, quando, ao invés disto, se poderia estar produzindo código. A leitura de um material bibliográfico que apresenta soluções já implantadas por outra equipe com sucesso, fornece um ganho de tempo muito significativo.

Como foi verificado ao longo da pesquisa, a metodologia XP não é aplicável em todas as situações e freqüentemente requer algum nível de adaptação, sendo específica em certas situações. Problemas semelhantes podem ser sanados de maneiras totalmente distintas, ficando ao cargo de um consenso comum a melhor prática a ser adotada em uma determinada situação, ou, até mesmo um conjunto de práticas para resolver de maneira mais abrangente o problema proposto. Algumas adaptações sugeridas possuem flexões na sua execução, bem como outras, são exclusivamente soluções únicas a serem aplicadas.

Como aspectos positivos das adaptações em relação à metodologia aplicada em sua estrutura original destaca-se a possibilidade de execução de todas as regras da XP nas equipes de desenvolvimento, já que, na sua estrutura base, não permite flexibilidade na execução de suas práticas, dificultando assim, a implantação da

metodologia em equipes que possuem restrições.

Do ponto de vista prático, as equipes de desenvolvimento geralmente possuem restrições que acabam por inviabilizar uma ou outra prática, fato que se for considerado de acordo com a rigidez da estrutura da XP, praticamente proíbe estas equipes de trabalharem em um projeto de software com a metodologia ágil XP.

De acordo com os objetivos específicos, observa-se no capítulo 2 a descrição das metodologias ágeis consideradas de maior destaque na área, definindo suas características funcionais em relação aos pontos negativos, detectando de maneira generalizada os problemas característicos provenientes da utilização de metodologias ágeis, intercalando sugestões no capítulo 3, que tendem a sanar tais fatores.

Em relação ao objetivo geral de levantar soluções funcionais para problemas comumente encontrados em projetos de desenvolvimento de software, podem-se dizer que seguindo a estrutura de adaptações possíveis, as quais, foram testadas por equipes e se demonstraram eficientes, a programação extrema pode ser adotada por um número muito maior de desenvolvedores. Sendo assim, a pesquisa se constitui numa contribuição bibliográfica para equipes de desenvolvimento que buscam respostas sobre qual metodologia utilizar e as possíveis soluções encontradas para problemas que provavelmente terão em um certo momento do projeto.

Foca-se, não apenas a facilitação da resolução dos problemas que comumente são enfrentados dentro dos projetos, bem como, estimular desta forma, a utilização de metodologias ágeis de desenvolvimento, em específico da XP, de forma a contribuir com um melhoramento da produtividade, tanto em relação à qualidade do código produzido, bem como, caracterizando-se como uma contribuição para os cronogramas de desenvolvimento.

5.1 Sugestões para Trabalhos Futuros

As metodologias ágeis apontam para um crescimento expressivo e rápido. O mercado demonstra sinais de uma absorção quase que inevitável destas metodologias, dentre as quais, a programação extrema tem sofrido um destaque considerável em relação as demais.

Baseado nesta idéia, pode-se estipular que ainda há muito que ser realizado para contribuir em um melhoramento das metodologias já existentes. Sendo mais específico sobre XP, pesquisas futuras poderiam apontar para uma pesquisa sobre outras metodologias ágeis, já que possivelmente em pouco tempo outras metodologias irão ganhar destaque no mercado, se apresentando possivelmente melhores que XP. Uma absorção de práticas e regras seguidas por outras metodologias poderia ser realizada e incorporada na XP, de forma a reunir o melhor contido em cada uma delas.

Entretanto, o foco não deve ficar apenas em pesquisa sobre a metodologia ágil da programação extrema. Trabalhos que busquem instigar um maior conhecimento no universo de outras metodologias vistas aqui, que não possuem provas acadêmicas formais da real aplicabilidade, como já realizado com XP, são requeridos para fornecer base a um maior estudo sobre metodologias ágeis, ou mesmo, gerar uma nova metodologia com as melhores regras que práticas que envolvem cada uma delas.

REFERÊNCIAS BIBLIOGRÁFICAS

Accounting, Management and Information Technology 10: 53–79. Wall, D. **Using Open Source for a Profitable Startup**. Computer December: 158–160, 2001.

ABRAHAMSSON, PEKKA E SALO, OUTI & RONKAINEN, JUSSI**. Agile software development methods Review and analysis,** University of Oulu, ESPOO 2002.

AGILE, ALLIANCE, **http://www.agilealliance.org** – acessado em 26/10/2002.

ALVIM, PAULO, **Gestão Ágil de Projetos – SCRUM na Prática**, Extreme Programming Brasil – Primeiro Congresso Brasileiro de Metodologias Ágeis de Software. São Paulo: 04-06 Dezembro, 2002.

AMBLER, SCOTT W. **"Agile Modeling: Effective Practices for Extreme Programming and the Unified Process":** John Wiley & Sons, 2002a.

_____, SCOTT W. **The Agile Data Method and Agile Database Techniques**, Extreme Programming Brasil – Primeiro Congresso Brasileiro de Metodologias Ágeis de Software. São Paulo: 04-06 Dezembro, 2002b.

ASTELS, DAVID. e MILLER, GRANVILLE e NOVAK, MIROSLAV. **Extreme Programming; guia prático**. Rio de Janeiro: Campus, 2002.

BECK, KENT, MARTIN FOWLER. **Extreme Programming Explained**: **Embrace Change**. Addison Wesley Longman, 2000.

BECKER, KORNSTAEDT, HAMANN, U., KEMPKENS, D., ZETTEL, R., **Support for the Process Engineer: The Spearmint Approach to Software Process Definition and Porcess Guidance**, Proceedings of the 11th Conference on Advanced Information Systems (CAISE'99)*,* 1999A.

BECKER, KORNSTAEDT, U., VERLAGE, M., **The V-Model Guide: Experience with a Web-Based Approach for Process Support**, Proceedings of Software Technology and Engineering Practices, 1999B.

BUNSER, C. , PELAYO, M., ZETTEL, J., **Out of the Dark: Adaptable Process Models for XP**: The third International Conference on eXtreme Programming and Agile Processes in Software Engineering, XP2002, Alghero, Sardinia, Italy, 2002.

CASTELLS, MANUEL. **The Information Age, Vol 1, The Rise of the Network Society**: Blackwell, 1996.

CAPRA, FRITJOF. **As Conexões Ocultas: Ciência para uma Vida Sustentável**. São Paulo: Cultrix, 2002.

COCKBURN, A. AND L. WILLIAMS. **"The Costs and Benefits of Pair Programming."** In Proceedings of the First International Conference on Extreme Programming and Flexible Processes in Software Engineering. Italy, 2000.

COLLINS, C.T., Miller, R.W **Adaption: XP Style**. Proc. XP2001

CRISPIN, L. HOUSE TIP: **Testing in the Fast Lane: Automating Acceptance Testing in an Extreme Programming Environment**, The third International Conference on eXtreme Programming and Agile Processes in Software Engineering, XP2002, Alghero, Sardinia, Italy, 2002.

CVS web site, http://www.cyclic.com - acessado em 20/11/2002.

DICK, ANDREW J., BRYAN ZARNETT, **Paired Programming & Personality Traits**, The third International Conference on eXtreme Programming and Agile Processes in Software Engineering, XP2002, Alghero, Sardinia, Italy, 2002.

FOWLER, MARTIN. **Is Planejamento Dead?.** Software Development, v9 i4 p42 April 2001.

FOWLER, MARTIN. **"Refactoring: Improving the Desing of Existing Code":** Addison-Wesley Longman, 1999.

GAMMA, E., KENT BECK, **Test infected: Programmers love writing tests**, http://www.junit.org, 1998.

GIL, ANTONIO C. **Como elaborar: Projetos de Pesquisa**. São paulo: Atlas,1996.

HODGETTS, PAUL, DENISE PHILLIPS, OSCAR CHICO. **Extreme Development with the Java 2 Platform**, Enterprise Edition (J2EESM), Presentation at JavaOneSM Conference, San Francisco, CA, June 2001.

JEFFRIES, RON. e ANDERSON, ANN. e HENDRICKSON, CHET. **“Extreme Programming Installed”:** Addison Wesley Longman, 2001.

JEFFERIES, RON, **“Are We Doing XP?”, Invited Talk, eXtreme Programming and Flexible Processes in Software Engineering** - XP2001, 2001B.

JUNIT WEB SITE: **http://www.junit.org -** acessado em 18/12/2002.

KERIEVSKY, J**. Patterns and XP. in Extreme Programming Examined**, G. Succi, M. Marchesi ed., Addison - Wesley, 2001

LANGR J.: **Evolution of Test and Code via Test-First Planejamento**. At http://www.objectmentor.com, 2001.

LARMAN, CRAIG. **Utilizando UML e padrões: uma introdução à análise e ao projeto orientados a objetos** – Bookman, 2000.

MACKINNON, T., **"Endo-Testing: Unit Testing with Mock Objects", eXtreme Programming and Flexible Processes in Software Engineering** - XP2000, 2000.

MULIANA, R. **Pontes Seguras – Uma Experiência de Extreme Programming em um Projeto Web**, Extreme Programming Brasil – Primeiro Congresso Brasileiro de Metodologias Ágeis de Software. São Paulo: 04-06 Dezembro, 2002.

NANDHAKUMAR, J. E AVISON, J. **The fiction of methodological development: a field study of information systems development**. Information Technology & People 12(2): 176–191, 1999.

NAUR, P. **Understanding Turing's universal machine: Personal style in program description**. The Computer Journal 36(4): 351–372, 1993.

NEWKIRK, JAMES e MARTIN, ROBERT C. **Extreme Programming in Practice**: Addison Wesley, 2001.

O'REILLY, T. **Lessons from Open Source Software Development. Communications of the ACM** Vol. 42(No. 4): 32–37, 1999.

PALMER, S. R. E FELSING, J. M. **A Practical Guide to Feature-Driven Development**. Upper Saddle River, NJ, Prentice-Hall. 2002

PARNAS, D. L. E CLEMENTS, P. C. **A rational planejamento process: How and why to fake it. IEEE Transactions on Software Engineering** 12(2): 251–257, 1986.

PFEFFER, J. **The human equation : building profits by putting people first. Boston, MA, Harvard Business School Press**. 1998.

PRESSMAN, ROGER S. **Engenharia de Software**. São Paulo: Makron Books, 1995.

RICHARDSON, ROBERTO J. **Pesquisa Social: Métodos e Técnicas**. São Paulo: Atlas, 1999.

RISING, L. AND JANOFF, N. S. **The Scrum software development process for small teams**. IEEE Software 17(4): 26–32. 2000

SCHUH, P. **Recovery, Redemption, and Extreme Programming**. IEEE Software 18(6): 34–41, 2001.

SCHWABER, K. **Scrum Development Process**. OOPSLA'95 Workshop on Business Object Planejamento and Implementation. Springer-Verlag. 1995.

SCHWABER, K. E BEEDLE, M. **Agile Software Development With Scrum**. Upper Saddle River, NJ, Prentice-Hall. 2002.

SHARMA, S., SUGUMARAN, V. E RAJAGOPALAN, B. **A framework for creating hybrid-open source software communities**. Information Systems Journal 12(1): 7–25. 2002.

SMITH, J. **A Comparison of RUP and XP**. Rational Software White Paper. http://www.rational.com/media/whitepapers/TP167.pdf. 2001.

SOL, H. G. **A feature analysis of information systems planejamento methodologies: Methodological considerations**. In: Olle, T. W., Sol, H. G. and Tully, C. J. (eds.). Information systems planejamento methodologies: A feature analysis. Amsterdam, Elsevier: 1–8. 1983.

SOMMERVILLE, I. (1996). Software engineering. New York, Addison-Wesley. Song, X. and Osterweil, L. J. **Comparing planejamento methodologies through process modeling**. 1st International Conference on Software Process, Los Alamitos, Calif., IEEE CS Press. 1991.

SONG, X. E OSTERWEIL, L. J. **Toward objective, systematic planejamentomethod comparisons**. IEEE Software 9(3): 43–53. 1992.

STAPLETON, J. **Dynamic systems development method – The method in practice**. Addison Wesley, 1997.

SUCCI, G. E MARCHESI, M. **Extreme Programming Examined: Selected Papers from the XP 2000 Conference**. XP 2000 Conference, Cagliari, Italy, Addison-Wesley, 2000.

SULTAN, F. E CHAN, L. **The adoption of new technology: The case of object-oriented computing in software companies**. IEEE Transactions on Engineering Management 47(1): 106–126, 2000.

SCHUH, P. **Recovery, Redemption, and Extreme Programming**. IEEE Software 18(6): 34–41, 2001.

SCHWABER, K. **Scrum Development Process**. OOPSLA'95 Workshop on Business Object Planejamento and Implementation. Springer-Verlag, 1995.

SCHWABER, K. E BEEDLE, M. **Agile Software Development With Scrum**. Upper Saddle River, NJ, Prentice-Hall, 2002

SHARMA, S., SUGUMARAN, V. E RAJAGOPALAN, B. **A framework for creating hybrid-open source software communities**. Information Systems Journal 12(1): 7–25, 2002

SHINE TECHNOLOGIES: **AGILE METHODOLOGIES Survey Results**, http://www.shinetech.com/agile_survey_results.jsp - acessado em 23/01/2003.

SMITH, J. **A Comparison of RUP and XP**. Rational Software White Paper. http://www.rational.com/media/whitepapers/TP167.pdf, 2001.

SOL, H. G. **A feature analysis of information systems planejamento methodologies: Methodological considerations**. In: Olle, T. W., Sol, H. G. and Tully, C. J. (eds.). Information systems planejamento methodologies: A feature analysis. Amsterdam, Elsevier: 1–8, 1983.

SOMMERVILLE, I. Software engineering. New York, Addison-Wesley. Song, X. and Osterweil, L. J. (1991). **Comparing planejamento methodologies through process modeling**. 1st International Conference on Software Process, Los Alamitos, Calif., IEEE CS Press, 1996.

SONG, X. E OSTERWEIL, L. J. **Toward objective, systematic planejamentomethod comparisons**. IEEE Software 9(3): 43–53, 1992.

STAPLETON, J. **Dynamic systems development method – The method in practice**. Addison Wesley, 1997.

SUCCI, G. E MARCHESI, M. **Extreme Programming Examined: Selected Papers from the XP 2000 Conference**. XP 2000 Conference, Cagliari, Italy, Addison-Wesley, 2000.

SULTAN, F. E CHAN, L. **The adoption of new technology: The case of object-oriented computing in software companies**. IEEE Transactions on Engineering Management 47(1): 106–126, 2000.

SUN MICROSYSTEMS INC. **“Code Conventions for the Java Programming Language”:** http://www.java.sun.com/docs/codeconv, 1999.

TAKEUCHI, H. E NONAKA, I. **The New Product Development Game**. Harvard Business Review Jan./Feb.: 137–146, 1986.

TRUEX, D. P., BASKERVILLE, R. E TRAVIS, J. **Amethodical systems**

development: The deferred meaning of systems development methods, 2000.

WAKE, WILLIAN C. **Extreme Programming Explored** - Addison-Wesley, 2000.

WARSTA, J. **Contracting in Software Business: Analysis of evolving contract processes and relationships**. Department of Information Processing Science. Oulu, University of Oulu, Finland: 262, 2001.

WELLS, DONAVAN**. Projeto de Programação Extrema:** http://www.extremeprogramming.org/map/project.html, 2003.

WIEGERS, K. E. **Read my lips: No new models**. IEEE Software 15(5): 10– 13, 1998.

WILLIAMS, LAURIE A., **The Collaborative Software Process**. Dissertation submitted to the faculty of The University of Utah in partial fulfillment of the requirements for the degree of Doctor of Philosophy, 2000.

VALIM, CARLOS E. **A Tecnologia da Informação na Era Lula**. Revista Information Week, São Paulo, v. 85, ano 5, p. 18-21, Jan, 2003.