Métodos ágeis

Autor:

Fábio Levy Siqueira

Versão:

24/07/2003	Versão original
25/07/2003	Revisão da Introdução e início do XP
27/07/2003	Texto sobre XP
28/07/2003	Texto sobre SCRUM
29/07/2003	Texto sobre Crystal
31/07/2003	Texto sobre FDD
01/08/2003	Texto sobre DSDM
03/08/2003	Texto sobre DSDM e revisão das figuras
04/08/2003	Texto sobre o LD
05/08/2003	Texto sobre o LD e ASD
06/08/2003	Texto sobre o ASD
31/08/2003	Revisão
1/07/2004	Revisão

Índice

Introdução

A complexidade e o tamanho dos sistemas computacionais requisitados pelo crescente e ávido mercado torna praticamente impossível construí-los de forma desorganizada. São necessários métodos de engenharia de software, ou usando a definição feita por Sommervile [22], "uma forma estruturada de se desenvolver software com o objetivo de facilitar a sua produção com alta qualidade e de maneira rentável".

Dentre os diversos métodos disponíveis, um grupo deles está em grande evidência atualmente: os métodos ágeis. O objetivo desses métodos, segundo Highsmith e Cockburn [12], é o de obter um desenvolvimento de software mais adequado ao ambiente turbulento dos negócios, que exige mudanças rápidas e freqüentes. Dessa forma, pregam práticas e princípios bastante diferentes dos outros métodos "tradicionais" (ou segundo Highsmith, rigorosos [13]), algo que pode ser facilmente observado no Extreme Programming, o método mais famoso e discutido desse grupo.

É interessante observar que o conceito de métodos ágeis é bastante novo, moldado em fevereiro de 2001 em uma conferência realizada para discutir as semelhanças entre os então métodos de "peso leve" (*lightweight*). Nesse encontro se reuniram alguns representantes de métodos e outros influentes simpatizantes. Apesar das aparentes diferenças entre os diversos métodos ali representados, os participantes dessa conferência encontraram alguns pontos em comum em suas ideologias, adotando o nome "ágil" e produzindo um manifesto com os princípios e valores [1].

A autodenominação desse grupo de métodos para ágil, ao invés de "peso leve", não é um mero detalhe. O termo anteriormente usado, segundo Cockburn, parece mais uma reação a algo do que uma crença [7]. Além disso, o termo "peso leve" leva ao entendimento que ou os métodos só servem para pequenos projetos – o que não é verdade – ou que utilizam processos pequenos e com poucos artefatos – o que é uma conseqüência dos seus princípios e não o ponto principal dos métodos. Dessa forma, a nova denominação por "ágil" é mais adequada por evidenciar um dos pontos principais desses métodos: a "importância de ser capaz de responder a mudanças de requisitos dentro do período de tempo do projeto" [7].

No entanto, a questão das mudanças, apesar de ser um ponto extremamente importante, não é o único em comum entre esses métodos, podendo ainda citar os seguintes:

- A busca pelo mínimo necessário e suficiente de documentação e processo;
- A importância das pessoas e da comunicação entre elas;
- O foco no cliente e também na sua participação direta;

- A entrega freqüente e com valor ao cliente.

Os demais pontos em comum entre os métodos ágeis são definidos no manifesto (disponível em [1]), considerando assim os 4 valores e os 12 princípios ali destacados.

A seguir serão apresentados os principais métodos ágeis, considerando os métodos representados na conferência original. Pretende-se aqui fornecer apenas a visão geral de cada um dos métodos, focando nas suas características principais que os destacam entre a inúmeras opções disponíveis atualmente.

Extreme Programming (XP)

O Extreme Programming é provavelmente o método ágil mais famoso e discutido atualmente. Ele causou bastante polêmica por ser muito diferente, e algumas vezes conflitante, em relação ao Processo Unificado [14], principalmente ao considerar que "o custo da mudança não deve aumentar dramaticamente com o tempo" [4], algo que contraria um dos princípios básicos da engenharia de software [5]. Por ser a premissa técnica do XP, diversas práticas propostas estão fortemente ligadas a esta, aproveitando as possibilidades criadas por essa afirmação.

Falando mais especificamente sobre o método, o Extreme Programming é direcionado para "times de pequeno a médio tamanho desenvolvendo software em face a requisitos vagos ou em mudança constante" [4]. Para atingir o sucesso nesse tipo de projeto, o XP prega 4 valores fundamentais e 12 práticas que, conforme o próprio nome do método sugere, são levadas ao extremo. Em relação aos valores – as premissas básicas do método que são usadas para direcionar as pessoas nos objetivos do projeto –, o XP prega os seguintes:

- **Comunicação**: promover a comunicação entre as partes envolvidas do projeto.
- **Simplicidade**: fazer algo da forma mais simples possível e funcional.
- **Retro-alimentação (*feedback*)**: permitir a retro-alimentação de informação de forma rápida e freqüente [16].
- **Coragem**: capacidade de assumir riscos e desafios em favor do projeto.

Apesar de ser normalmente colocado em segundo plano, o XP também define um conjunto de princípios (no total de 15 – vide Tabela 1) com o objetivo de concretizar os valores, facilitando o seu uso pelas pessoas envolvidas. O interessante desses princípios é o fato deles terem conceitos fortes e muito importantes para entender o XP e utilizá-lo[1], como, por exemplo, a

[1] Alguns dos princípios do Extreme Programming são bastante difíceis de se aplicar, já que pregam mudanças no comportamento das pessoas e das idéias enraizadas nas empresas, como, por exemplo, a comunicação aberta e honesta, responsabilidade aceitada e medir honestamente.

adaptação local que prega a adequação do XP às condições de trabalho – mostrando que o XP não é tão fechado como as pessoas costumam vê-lo.

Princípios Centrais	Outros Princípios	
▪ Retro-alimentação rápida; ▪ Simplicidade assumida; ▪ Mudança incremental; ▪ Abraçar as mudanças; ▪ Trabalho com qualidade.	▪ Ensinar aprendendo; ▪ Pequeno investimento inicial; ▪ Jogar para ganhar ▪ Experimentos concretos ▪ Comunicação aberta e honesta;	▪ Trabalhar com o instinto das pessoas e não contra eles; ▪ Responsabilidade aceitada; ▪ Adaptação local; ▪ Viagem leve; ▪ Medir honestamente;

Tabela 1: Os princípios do Extreme Programming.

Os valores e princípios são o espírito do XP, mas é necessário mais que isso para criar um processo eficaz. Nesse âmbito foram definidas as práticas que detalham de maneira mais clara como e quando cada desenvolvedor deve atuar e também organizando a equipe de desenvolvimento para tornar o processo do XP sólido. A seguir são apresentadas essas práticas:

- **O jogo do planejamento (*the planning game*)**: define como planejar, "combinando prioridade de negócio e estimativas técnicas" [4].
- **Entregas breves (*small releases*)**: entregar algo de valor ao cliente rapidamente e com grande freqüência.
- **Metáfora**: criar uma metáfora que expresse de maneira simples e clara a idéia do sistema a ser construído.
- **Design simples**: o design deve ser feito da maneira mais simples possível, focando no na tarefa atual e não pensando em possibilidades futuras.
- **Teste**: os testes devem ser feitos de forma automatizada; os de unidade feitos pelos desenvolvedores antes da implementação da tarefa e os clientes escrevendo os funcionais.
- **Refatoramento (*refactoring*)**: o código deve ser continuamente reestruturado pelos desenvolvedores, tirando a duplicação de código e tornando-o mais simples após uma modificação.
- **Programação em duplas (*pair programming*)**: a produção do código deve ser feita em duplas, uma escrevendo o código e a outra auxiliando e pensando de forma mais estratégica.
- **Autoria coletiva (*collective ownership*)**: o código pode ser alterado por qualquer desenvolvedor, tornando todos responsáveis pelo código do sistema.

- **Integração contínua**: as tarefas terminadas devem ser integradas ao já construído, executando os testes automatizados para verificar a correção do sistema.
- **Semana de 40 horas**: os desenvolvedores devem ter o descanso necessário para executar as tarefas com mais atenção e vontade.
- **Cliente no local (*on-site customer*)**: um cliente deve estar disponível no local para sanar as dúvidas que forem aparecendo durante a implementação e com o poder necessário de decisão.
- **Padrão de código**: definir um padrão para codificação para que o código possa ser entendido de forma mais fácil pelos desenvolvedores.

Muitas dessas práticas são alvos freqüentes de críticas, como a programação em duplas – que seria um desperdício de pessoal – e o refatoramento – que seria uma forma de retrabalho devido a um projeto mal feito. No entanto, não é adequado julgar as práticas de forma isolada, já que cada uma delas tem um papel no processo, ou como Highsmith define, "fortalecendo uma a outra de forma direta e sutil" [13]. Por esse mesmo motivo, alterar alguma prática necessita um profundo entendimento do XP e da equipe de trabalho, para não criar um problema sério no processo. Na Figura 1 é apresentada uma representação do processo do XP, incorporando as práticas, princípios e valores.

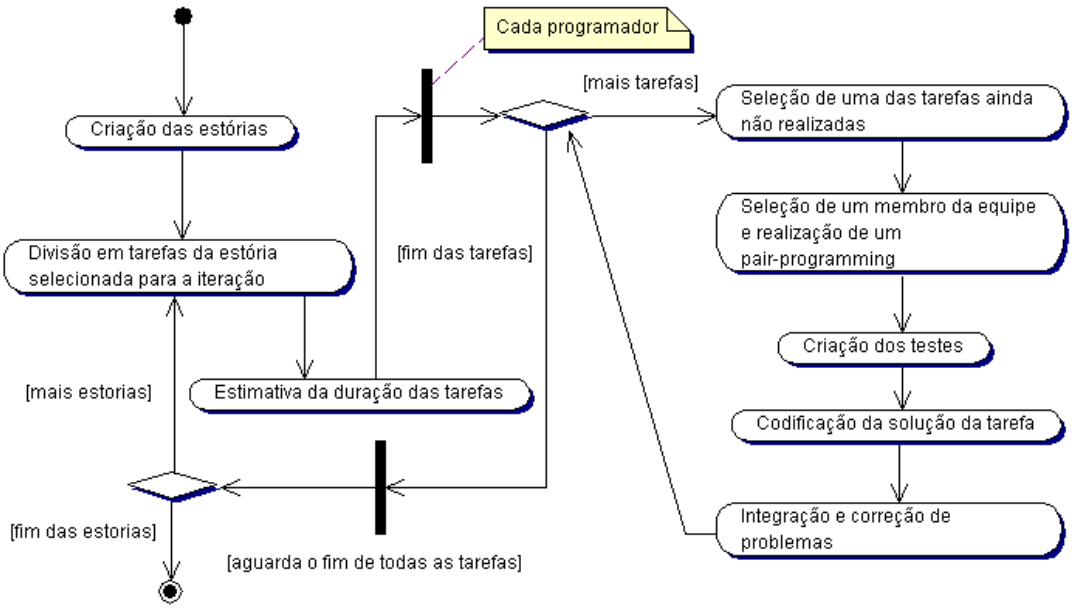

Figura 1: O processo executado em um projeto normal do XP (retirado de [21]).

SCRUM

O SCRUM é um método ágil que utiliza como fundamento principal a classificação retirada da área de processos de controle industrial, que divide os processos em definidos e empíricos. Em

um processo definido, se tem conhecimento suficientemente detalhado de todos os itens que serão utilizados – suas leis de transformação e de mistura. Isso permite criar um processo automatizado, já que é possível prever o resultado de antemão. No caso de um processo empírico, não se sabe exatamente como as partes se relacionam, tratando assim o que acontece internamente como uma caixa preta. Para conseguir obter os resultados desejados em um processo desse tipo o fundamental é controlar adequadamente as partes, atuando rapidamente para manter as partes nos limites conhecidos [3].

Dessa idéia, o SCRUM analisa o processo de desenvolvimento de software e conclui que, com o conhecimento atual, a classificação mais adequada é como processo empírico. Com isso, a utilização de controle adequado e adaptação rápida aumentariam consideravelmente o sucesso dos projetos, principalmente os maiores e a "beira do caos". Essa idéia é representada na Figura 2 que, segundo Highsmith [13], pode ser generalizada para todos os processos ágeis e não a apenas ao SCRUM.

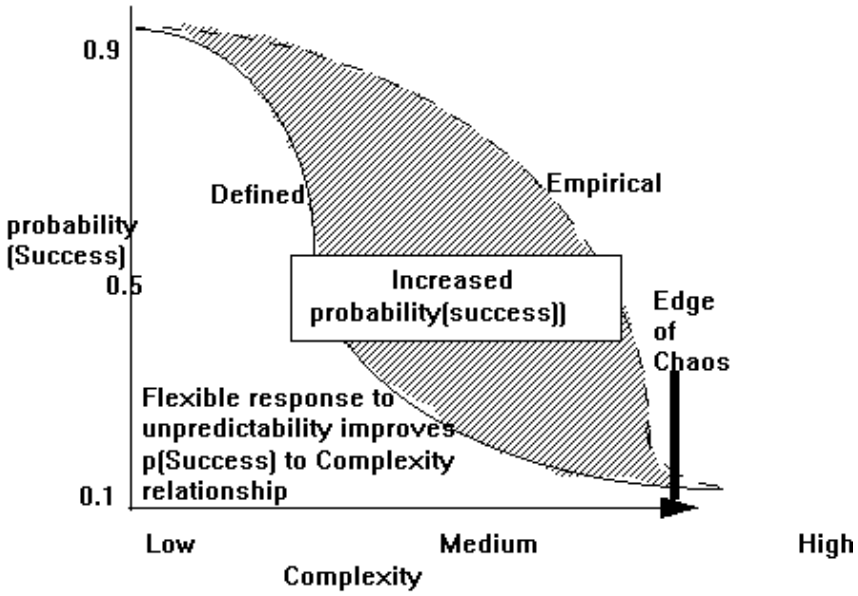

Figura 2: Sucesso do projeto e o aumentado com um método ágil frente à sua complexidade (retirado de [2]).

No entanto, essa avaliação em processo empírico não é completamente aceita, sendo conflitante com o CMM do Software Engineering Institute (SEI), que prega a definição de um processo de desenvolvimento de software que é continuamente melhorado para uma organização de software.

Esquecendo as controvérsias e falando mais especificamente sobre o método SCRUM, ele é dividido nas fases pré-jogo, jogo e pós-jogo. Na fase de pré-jogo é feito o planejamento que

define principalmente as funcionalidades e suas estimativas, criando assim os *backlogs* ("todo trabalho que será realizado em um futuro previsível, bem definido e também precisando de futura definição" [3]). Nela também é criada a arquitetura básica, sendo definida a visão de desenvolvimento do projeto a partir dela [20].

Depois do planejamento é feito o jogo, que é uma série de iterações rápidas de desenvolvimento (de 1 a 4 semanas), chamadas de *Sprint* (corridas) – vide Figura 3. Nelas o primeiro passo é realizar uma reunião em que os clientes, os gerentes e os desenvolvedores criam, decidem e priorizam as tarefas do *backlog* a serem executadas durante nessa iteração, com um determinado objetivo definido. Posteriormente os grupos de desenvolvedores, que devem ser completamente funcionais e de até 10 pessoas, dividem entre eles as tarefas determinadas.

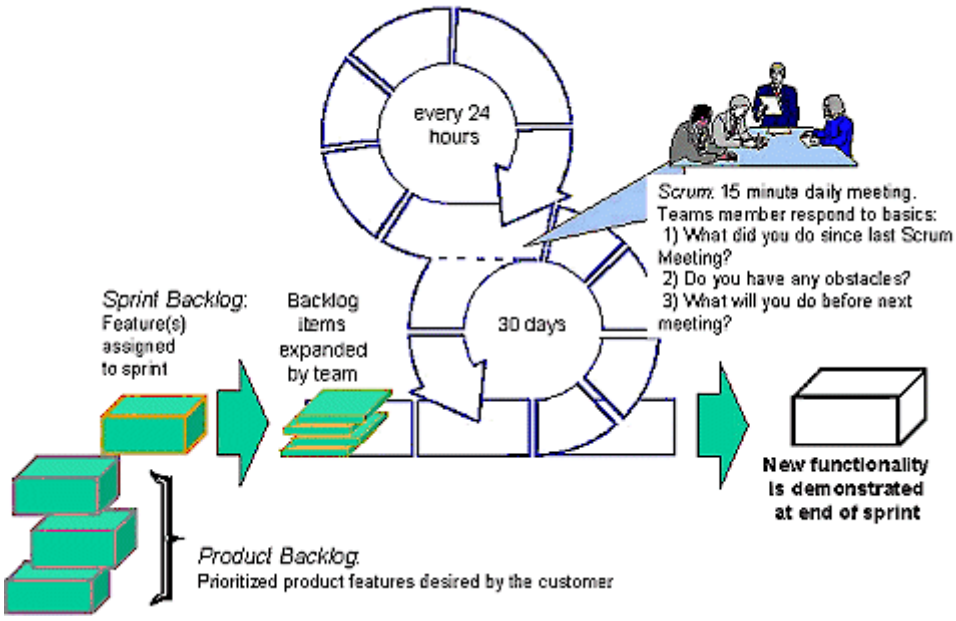

Figura 3: A execução de um *Sprint* (retirado de [3]).

Pensando na teoria de processos industriais, essa é a fase empírica do processo, o que obriga a existência de formas de controle. Com isso, diariamente os grupos de desenvolvedores se encontram, discutindo rapidamente (30 minutos) o andamento do projeto. Isso permite ao gerente e aos desenvolvedores terem a visão do processo por inteiro, e também a possibilidade do gerente corrigir os problemas que estão atrapalhando o desenvolvimento. Também diariamente o software é integrado, o que aumenta ainda mais a visibilidade do andamento do desenvolvimento.

Uma outra forma de diminuir a complexidade do processo é a o fato das tarefas e suas prioridades ficarem estáveis durante cada uma das iterações. No entanto não é uma estabilidade completa, já que os desenvolvedores podem colocar mais trabalho no *backlog* que seja considerado por eles necessário.

Com o término do desenvolvimento desse *Sprint* é feita uma reunião com o cliente para verificar o andamento do projeto, demonstrar as funcionalidades prontas e revisar o projeto sob uma perspectiva técnica [13].

Ao fim de uma iteração, pode ser decidido pela gerência que o projeto já atingiu seu objetivo, criando uma entrega do produto (fase de pós-jogo). Nesse momento é feito todo o treinamento necessário, gerada toda a documentação, executados os testes de sistema, etc.

Crystal

Crystal é o nome de uma família de métodos que devem ser ajustados para melhor se adaptarem a uma determinada equipe e projeto. Cada método é moldado para ter a quantidade exatamente suficiente de processo, capaz de atender os projetos a partir da análise de três fatores: a carga de comunicação (representada pelo número de pessoas), a criticidade do sistema e a prioridade do projeto. Na Figura 4 são apresentadas apenas duas dimensões, demonstrando qual método da família Crystal deve ser utilizado (Clear, Yellow, Orange, Red). Segundo Cockburn [8], existem diversos outros fatores para a escolha de um método, no entanto ao utilizar apenas esses três já se obtêm um resultado bastante adequado.

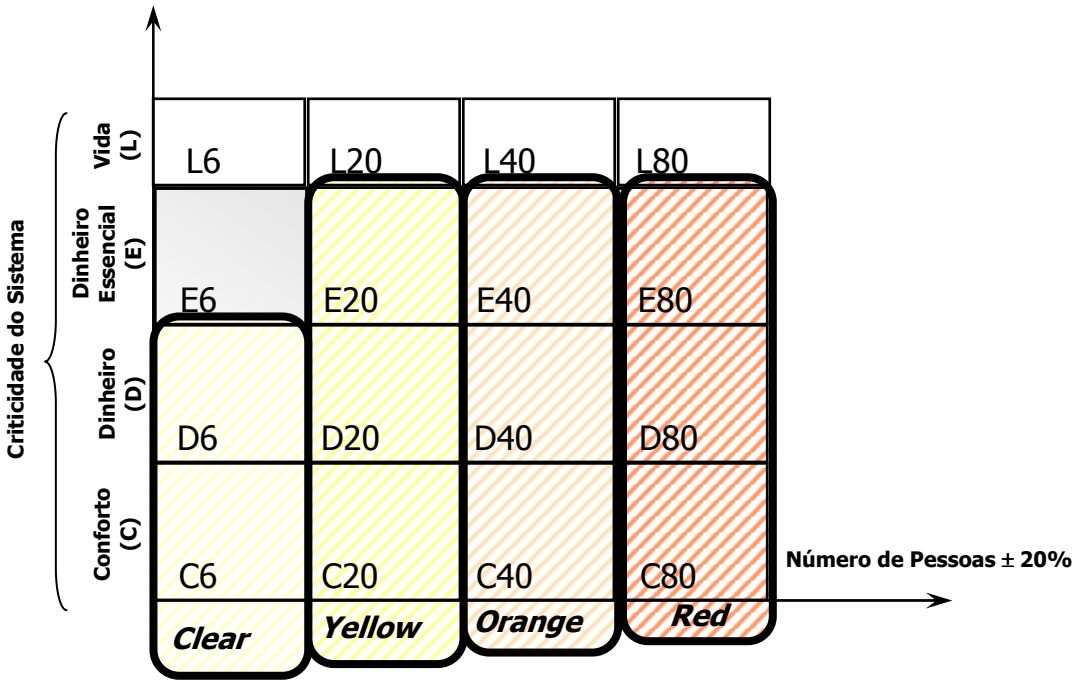

Figura 4: A distribuição dos métodos da família Crystal a partir de duas dimensões (retirado e adaptado de [9]).

Conforme as cores dos membros da família Crystal se tornam mais escuras, tem-se um maior peso dos métodos, o que é necessário devido à complexidade dos projetos. Esse peso é representado pela quantidade de artefatos e a rigidez da gerência, itens que são absorvidos entre os 13 elementos definidos para cada método: papéis, habilidades, times, técnicas, atividades, processos, artefatos, produtos de trabalho, padrões, ferramentas, personalidades, qualidade e valores da equipe [13]. Por exemplo, ao observar os papéis no Crystal Clear, nota-se que existem 6 tipos [9], enquanto que no Orange existem mais de 14 [7].

Apesar das diferenças entre as complexidades dos projetos abordados pelos métodos da família Crystal, eles apresentam em comum valores, princípios e a capacidade de serem ajustados durante o uso (*on-the-fly*). Os valores denotam a visão do desenvolvimento de software como um "jogo cooperativo de invenção e comunicação, com o objetivo primário de entregar algo útil e, em segundo, preparar para o próximo jogo" [7]. Dessa forma, os valores pregam que os métodos são centrados nas pessoas e na comunicação, além de serem altamente tolerantes a modificações – considerando as diferenças entre as pessoas. Isso permite que sejam utilizadas partes de outros métodos, como práticas e princípios do XP e o arcabouço do SCRUM. No entanto, existem duas regras que devem ser seguidas, limitando essas modificações: o desenvolvimento deve ser incremental, com incrementos de até 4 meses, e devem ser realizados workshops de reflexão antes e depois de cada incremento.

Em relação aos princípios, eles são os seguintes (retirado de [7]):

- Comunicação iterativa, face-a-face é o canal mais barato e rápido para o intercâmbio de informações.
- O excesso de peso do método é custoso.
- Times maiores precisam de métodos mais pesados.
- A maior cerimônia é apropriada para projetos com maior criticidade.
- Aumentando a retro-alimentação (*feedback*) e comunicação é reduzida a necessidade de itens intermediários entregues.
- Disciplina, habilidade e entendimento contra processo, formalidade e documentação.
- Pode-se perder eficiência em atividades que não são o gargalo do processo.

O interessante dos métodos Crystal é que eles têm uma aparência de métodos não-ágeis. Apesar de terem princípios e valores claramente alinhados com o manifesto de desenvolvimento ágil, eles apresentam elementos (papéis, documentação, artefatos, etc) que são normalmente vistos em métodos como o Processo Unificado [14]. Essa característica permite que os métodos Crystal possam ser mais facilmente utilizados por uma equipe que o Extreme Programming, mas segundo o próprio autor [8], o XP apresenta melhores resultados que o Crystal Clear. Dessa forma, uma equipe pode começar com o Crystal Clear e posteriormente passar para o XP, ou uma equipe que tenha sérios problemas de adaptação com o XP pode passar para o Crystal Clear [8].

Feature-Driven Development

O Feature-Driven Development (FDD) é um método que tem um foco grande na modelagem (diagrama de classes e seqüência) e utiliza como forma de planejamento e medição as características, ou melhor, "funções com valor ao cliente" [17]. A utilização de características no processo FDD tem diversas vantagens, o que, segundo Coad et al [6], permite agradar à todas as partes envolvidas. Os programadores ficam mais motivados e satisfeitos, uma vez que mudam o foco de trabalho em poucas semanas. Os gerentes reduzem riscos ao entregar "resultados freqüentes, tangíveis e funcionando", além de poder usar as características como forma de medição, gerando relatórios com porcentagens. Os clientes recebem freqüentemente resultados que eles entendem e também podem saber a qualquer momento o andamento do projeto com precisão.

Para que seja possível gerenciar de forma adequada as características, o FDD define papéis, práticas e um processo de maneira bastante simples. Em relação aos papéis, apesar de serem definidos diversos como, por exemplo, o gerente e o arquiteto chefe, existem dois papéis que são principais: o "dono" da classe e o chefe programador. O chefe programador é um desenvolvedor que cuida da implementação de algumas das características, criando um time

para o desenvolvimento através da verificação de quais são as classes que serão usadas e recrutando seus respectivos "donos". Com isso, podem existir (dependendo do tamanho da equipe) diversos times de desenvolvimento, o que pode fazer com que os desenvolvedores trabalhem em mais de um time (recomenda-se no máximo 3) durante uma iteração. O interessante é que o chefe programador é também "dono" de algumas classes, podendo ser recrutado para a participação de um time liderado por outro chefe programador.

Em relação às práticas definidas no FDD, elas não são extremamente rígidas, pregando a adaptação ao ambiente de desenvolvimento. No entanto, existe um conjunto de práticas que são fundamentais e que definem o FDD [17]:

- **Modelagem em objetos de domínio**: construir um diagrama de classes básico com os objetos de domínio e suas relações, definindo assim uma arquitetura básica para o modelo do sistema.
- **Desenvolvimento por características**: a implementação deve ser orientada pelas características.
- **Autoria individual**: o código é de autoria de um "dono" da classe, o que permite uma maior rapidez na implementação das tarefas associadas.
- **Times da característica**: para a implementação de uma determinada característica, o chefe programador recruta os "donos" das classes que serão usadas. Esse grupo de pessoas é o time da característica.
- **Inspeções**: a forma de verificação de qualidade do código e do projeto.
- **Integração (build) regular**: em um determinado período de tempo fixo devem ser integradas as características já terminadas, permitindo a verificação de erros e também criando uma versão atual que pode ser demonstrada ao cliente.
- **Gerência de configuração**: manter versões de todos os artefatos criados.
- **Reportar/Visibilidade dos resultados**: permitir que se conheça o progresso do projeto.

Essas práticas são incorporadas nos 5 processos que são descritos através de uma forma textual direta e objetiva (disponível em [15]) em que é definido o critério de entrada, as tarefas associadas, formas de verificação e o critério de saída. A visão geral dos processos e seu sequenciamento é apresentado na Figura 5.

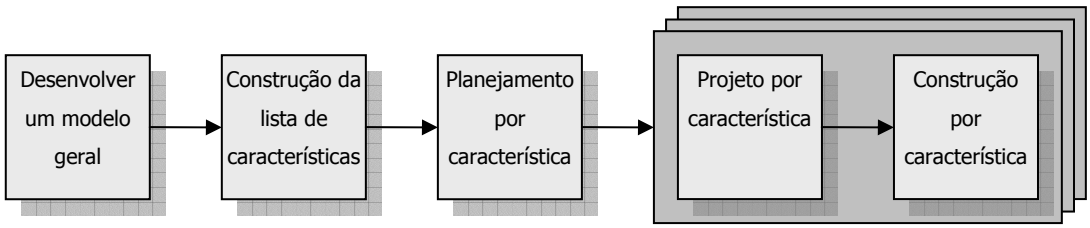

Figura 5: Os processos do FDD (retirado e adaptado de [6]).

O primeiro processo é o desenvolvimento de um modelo geral, incorporado em um modelo de domínio em que são colocadas apenas as classes e suas relações, com o detalhamento suficiente para apenas dar forma ao sistema. Esse processo tem como objetivo diminuir o refatoramento, uma vez que gera um arcabouço para a construção do sistema nos demais processos, mantendo assim uma integridade conceitual [17].

No segundo processo é feita a construção de uma lista de características, agrupando, priorizando e dando pesos a elas. Essa lista é obtida através da transformação de métodos criados anteriormente no modelo para uma característica e alterando, removendo e adicionando para se obter uma lista que represente adequadamente as expectativas do cliente. Um outro ponto importante é que as características devem ser breves, preocupando com o período de desenvolvimento que não deve ultrapassar duas semanas – o que pode obrigar a uma decomposição de uma característica.

O terceiro processo é a criação de um plano feito apenas pelos desenvolvedores – sem o cliente – que foca na implementação. Esse plano deve considerar "as dependências entre as características, a carga da equipe de desenvolvimento e a complexidade das características" [15].

Após esse processo é iniciada a implementação das características de forma iterativa através de dois processos: o projeto por característica e a construção por característica. No processo de projeto são criados os times (da característica) que devem desenvolver um diagrama de seqüência que reflete a característica a ser implementada. Com isso, o chefe programador deve refinar o modelo de objeto através do que foi obtido no diagrama de seqüência e os "donos" das classes devem escrever a documentação dos métodos. No processo seguinte, o de construção por característica, os programadores devem implementar o que foi discutido na etapa anterior (métodos, classes, etc) e criar os testes de unidade.

Lean Development

O Lean Development (LD) é um método baseado na visão da produção Lean, da área de manufatura e processos. Essa forma de produção seria o estágio seguinte à produção em massa, pregando a customização em massa, com a ambição de diminuir consideravelmente os gastos: metade do esforço, do espaço, do investimento em ferramentas, etc [13]. No âmbito do desenvolvimento de software essas propostas seguem a mesma linha, filosofia e ambição. Dessa forma, o LD não se preocupa em pregar práticas de desenvolvimento – ao contrário do XP –, já que seu foco é no ponto de vista estratégico, direcionado primordialmente ao nível gerencial da organização e não ao nível de instrumentação, no caso, os desenvolvedores.

A idéia geral dessa forma da produção Lean é a observação dos problemas da mudança de requisitos sob uma nova perspectiva: as decisões do cliente devem ser adiadas ao máximo, deixando para que elas sejam feitas quando houver um maior conhecimento do assunto [18]. E quando as decisões forem feitas, a implementação deve ser rápida, evitando que as condições externas afetem uma funcionalidade antes dela ser entregue.

Por ser uma forma de produção passada para a área de Engenharia de Software, é necessário que ocorra uma adaptação adequada de seus princípios com a preocupação de manter a filosofia para esse outro ambiente de produção. Dessa forma, a seguir são apresentados esses princípios, derivados para a visão do software segundo Poppendieck [19]:

- **Eliminar o desperdício**: ao observar o processo produtivo por inteiro.
- **Amplificar o aprendizado**: aumentando a retro-alimentação, através de ciclos rápidos e iterativos.
- **Atrasar o compromisso**: deixar suas opções disponíveis pelo maior tempo possível, para que as decisões sejam feitas com o máximo de conhecimento sobre o assunto [18].
- **Entregar rapidamente**: as entregas devem estar alinhadas com os anseios do cliente em um determinado momento de tempo. A demora para entregar pode fazer com que o resultado não seja mais o que o cliente precise ou deseje.
- **Dê poder ao time**: a equipe de desenvolvimento deve ter o poder de decidir, algo necessário devido à rapidez da implementação.
- **Construa com integridade**: o software deve satisfazer ao cliente – realizando o que ele deseja – (integridade percebida), ter um núcleo coeso (integridade conceitual) e manter-se útil com o tempo.
- **Veja o todo**: a visão dos implementadores deve ser do produto como um todo e não apenas de uma parte ou subsistema.

Uma adaptação mais prática e voltada para implementação desses princípios para a Engenharia de software é feita por Charette apud Highsmith [13]. Dessa forma, foi interpretada a produção Lean para criar o método de desenvolvimento de software conhecido por LD. A seguir, é colocado o conjunto de princípios que regem esse método [13]:

1. A satisfação do cliente é a maior prioridade.
2. Sempre provenha o melhor valor para o dinheiro.
3. Sucesso depende na participação ativa do cliente.
4. Todo projeto LD é um esforço do time.
5. Tudo pode ser mudado.
6. As soluções devem ser de domínio e não pontuais.
7. Complete, não construa.
8. 80% da solução hoje é melhor do que 100% da solução amanhã.
9. Minimalismo é essencial.
10. As necessidades determinam a tecnologia.
11. O crescimento do produto é o crescimento de características, e não de tamanho.
12. Nunca force o LD para fora de seus limites.

Em relação ao processo de desenvolvimento, são definidas três etapas principais: início, o estado de crescimento regular (*steady-state*), e transição e renovação. No início o foco é "análise de valor ao cliente, o estudo do negócio e projeto de viabilidade" [13], com isso é feita uma análise de riscos e outras etapas de escopo gerencial.

Na etapa de estado de crescimento regular o objetivo é realizar o desenvolvimento de forma iterativa e incremental em períodos fechados (*time-boxes*) de 90 dias no máximo. Cada iteração realiza um pouco das fases tradicionais de desenvolvimento (análise, projeto, implementação e testes), integrando continuamente o software. A grande diferença em relação aos outros métodos é a utilização em larga escala de *templates* (soluções de domínio) para aumentar a velocidade de produção e sua reutilização (condizente com o princípio 7).

Na etapa de transição é feita a entrega, focando na documentação e treinamento, pensando ainda na evolução do sistema.

Dynamic Systems Development Method

O Dynamic Systems Development Method (DSDM) é uma formulação dos métodos RAD (*Rapid Application Development*) organizada por um consórcio de companhias membros que, além de fornecer serviços e treinamentos, também cuida do licenciamento de uso do método. A ênfase do DSDM, que se conceitua mais como um arcabouço do que um método, é na criação de

protótipos que evoluem para o sistema, utilizando para isso a colaboração muito próxima do cliente.

As idéias principais do DSDM podem ser observadas no conjunto de princípios que foram definidos para nortear o método [10]:

- O envolvimento ativo do usuário é imperativo.
- O time deve ter o poder para tomar decisões.
- O foco é na entrega freqüente de produtos.
- O encaixe ao propósito do negócio é o critério essencial para a aceitação das entregas.
- O desenvolvimento iterativo e incremental é necessário para convergir com precisão às soluções do negócio.
- Todas as mudanças durante o desenvolvimento são reversíveis.
- Requisitos são alinhados em um alto nível.
- O teste é integrado por todo o ciclo de vida.
- Uma abordagem colaborativa e cooperativa entre as partes envolvidas é essencial.

É possível notar com esses princípios que o cliente e o negócio são os pontos essenciais do método, enquanto que os aspectos técnicos, como a programação, são pouco abordados, o que fica ainda mais evidente na descrição do processo. Esse enfoque e o fato do DSDM ser visto como um arcabouço faz com que existam soluções para mesclá-lo a outros métodos, sendo possível a utilização em conjunto, por exemplo, com o XP ou o RUP, mas ainda mantendo os princípios definidos no DSDM.

Além desses princípios, existem algumas técnicas principais que são usadas durante a execução de um projeto usando DSDM [10]:

- **Time-boxes**: definição de um período fixo para a execução do projeto, colocando até datas de entrega. Com isso, caso haja alguma funcionalidade que não possa ser implementada durante o período estipulado, ela deve ser feita após o desenvolvimento em si (antes da fase de pós-projeto).
- **MoSCoW**: regra básica para a priorização de requisitos durante o período de desenvolvimento. A idéia fundamental é priorizar e implementar os requisitos que sejam considerados principais, deixando os menos importantes para depois.
- **Modelagem**: não deve ser uma atividade burocrática, sendo usada para prover um melhor entendimento do problema e da solução.
- **Prototipação**: forma de verificar a adequação dos requisitos e facilitar as discussões com o cliente. O protótipo criado deve evoluir juntamente com o projeto.
- **Teste**: essa atividade deve ser executada sistematicamente e de forma contínua durante o projeto.
- **Gerência de configuração**: essencial, visto que os produtos são entregues com uma grande freqüência.

Em relação ao processo do DSDM, existem 5 fases básicas (que podem ser vistas na Figura 6), antecedidas por uma fase de pré-projeto e precedidas pelo pós-projeto. No pré-projeto, tem-se como objetivo definir se o projeto deve ou não ser implementado, observando aspectos gerenciais básicos, como questões monetárias e um plano para o estudo de viabilidade. O estudo de viabilidade em si é feito na etapa seguinte, em que se verifica se o DSDM é a solução mais adequada, além das atividades tradicionais em um estudo desse tipo. Na etapa seguinte, de estudo do negócio, são observados "os processos que serão afetados e as suas necessidades de informação" [10], definindo o escopo do projeto.

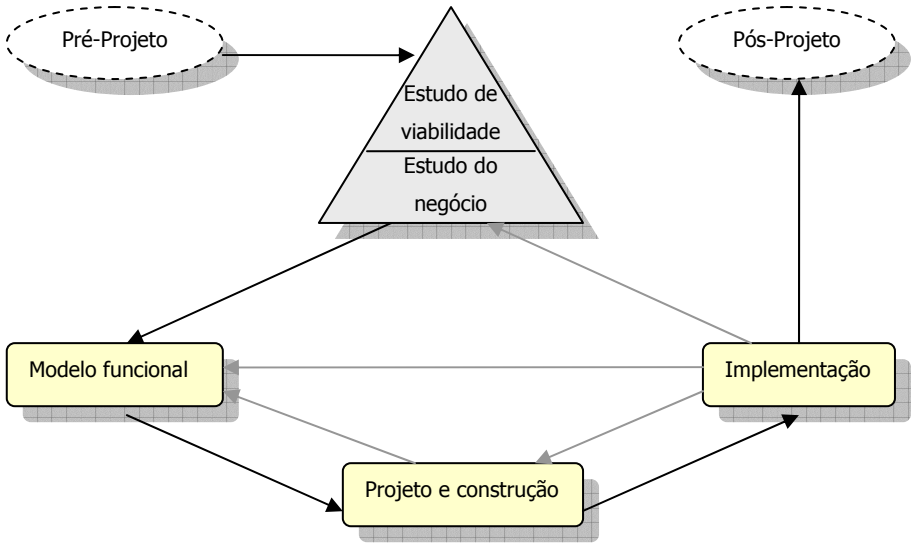

Figura 6: O processo do DSDM (adaptado de [10]).

Posteriormente é iniciado o desenvolvimento em si, que é executado de forma interativa em cada uma das três fases seguintes: modelagem funcional, projeto e construção e implementação. Como a transição entre essas fases é algo bastante complicado, a decisão de quando e como isso deve acontecer acaba sendo feita de projeto a projeto, podendo haver sobreposição e mescla entre elas. Além disso, a qualquer momento pode haver um refinamento do projeto, fazendo com que se volte a fases anteriores para corrigir problemas, solucionar dúvidas, etc.

Na primeira fase de desenvolvimento, que cuida do modelo funcional, os requisitos (funcionais e não funcionais) são obtidos, montando uma lista de prioridades e colocando-os no protótipo, documentando a maioria dessa forma ao invés da textual [13]. No entanto, o foco é apenas a visão básica dos requisitos, uma vez que os detalhes deles serão desenvolvidos na fase de projeto e construção, em que o objetivo é obter o sistema testado. Na fase de implementação é

feita a transição do sistema do ambiente de desenvolvimento para o operacional, cuidando do treinamento e outras tarefas que sejam necessárias.

Ao finalizar as etapas de desenvolvimento com um resultado satisfatório na realização dos requisitos, chega-se a fase de pós-projeto. Nela é feita a manutenção do sistema, realizando as tarefas de alteração praticamente da mesma forma que foi feito o desenvolvimento.

Adaptive Software Development

O Adaptive Software Development (ASD) tem como base principal um método RAD (*Rapid Application Development*), o RADical Software Development, evoluindo-o ao incorporar conceitos da teoria de sistemas adaptativos complexos. Sob esse panorama, o ASD propõe atualizar o ciclo de desenvolvimento baseado em planejar, projetar e construir, trocando-o por um com as fases de especular, colaborar e aprender. Essa mudança seria necessária devido ao enfoque diferente dos dois ciclos: o primeiro considera a estabilidade no ambiente de negócios, enquanto o segundo foca em ambientes de incerteza e de grande mudança [11] – visão comum a todos os métodos ágeis.

Com esse novo ciclo de desenvolvimento, seria mais fácil se adequar a esse ambiente turbulento, trocando uma fase de planejamento, algo baseado na tentativa de predição, por algo mais factível ao ambiente de incerteza: o "especular". O foco nesse caso seria explorar e experimentar, sem abandonar o planejamento por completo, mas assumindo sua falta de precisão. A próxima fase do ciclo propõe o trabalho colaborativo entre as partes envolvidas, criando um fluxo de informação suficiente para que possam ser resolvidos rapidamente os problemas técnicos e de requisitos de negócios. Essa forma de trabalho é necessária devido à grande quantidade de informação que deve ser coletada e trabalhada para um sistema complexo, além da turbulência do ambiente. Por fim, é necessário que haja retro-alimentação para que seja possível aprender com os erros anteriores, corrigindo-os para as próximas iterações. Isso pode ser realizado através de grupos de foco do cliente e retrospectivas do projeto.

Para que o desenvolvimento seja realmente adaptativo é necessário que esse novo ciclo tenha as seguintes características:

- **Enfoque na missão**: fazer com que a equipe tenha um objetivo definido e permitindo que sejam feitas decisões com maior embasamento.
- **Baseado em características**: o objetivo é entregar resultados mais palpáveis ao cliente, através de características implementadas no sistema – ao invés de outras formas de produtos (como, por exemplo, documentação).

- **Iterativo**: a construção deve focar na evolução do produto.
- **Períodos fechados (*time-boxes*)**: a equipe deve ter um objetivo definido em um determinado período, priorizando e decidindo para que seja entregue o combinado no prazo adequado.
- **Dirigido a riscos**: é necessário analisar e avaliar os riscos do projeto continuamente, assim como em um desenvolvimento em espiral.
- **Tolerante a mudanças**: incorporar as mudanças que forem aparecendo durante o projeto, para que o sistema tenha maior valor ao cliente.

Sob essa perspectiva de um novo ciclo e suas características necessárias, o ASD define o seu ciclo de vida para projetos. Com isso, as fases do ciclo especular-colaborar-aprender são preenchidas com algumas práticas, conforme colocado na Figura 7.

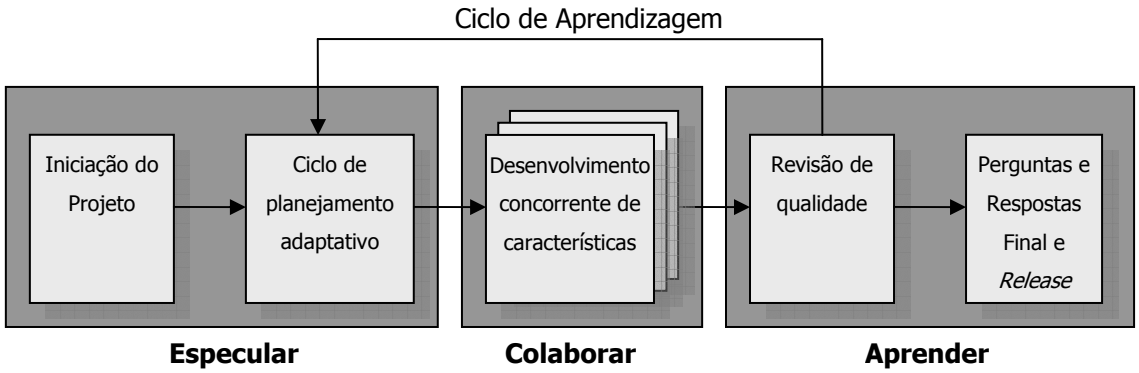

Figura 7: Ciclo de vida do ASD (adaptado de [13]).

Na fase de especular são feitas algumas atividades gerenciais, com a iniciação do projeto e o planejamento iterativo de seu desenvolvimento. Na iniciação é recomendada a utilização de sessões JAD (*joint application development*), para que sejam definidos os objetivos do projeto, requisitos, problemas, riscos e também estimando o tamanho e o escopo. A seguir deve-se definir os períodos de implementação para todo o projeto, o número de ciclos necessários, um objetivo ou tema para cada um deles, as características que serão implementadas, as tecnologias utilizadas e, se desejado, uma lista de tarefas. A maioria dessas definições deve ser feita em conjunto com o cliente, obtendo seu consentimento.

Na fase seguinte do ciclo, é feita a implementação do sistema em paralelo. Dessa forma, os desenvolvedores devem tentar promover ao máximo a comunicação e colaboração entre as pessoas e entre times, como, por exemplo, ao realizar discussões em lousas ou em conversas pessoais, ou até aplicar práticas como a programação em pares e autoria coletiva do XP. Os gerentes devem facilitar essas práticas, se preocupando com a concorrência na implementação.

Por fim, devem ser feitas revisões da qualidade pela gerência, avaliando o que foi entregue pelos desenvolvedores. Essas informações são retro-alimentadas ao ciclo, permitindo que sejam feitos planejamentos mais adequados posteriormente. Além disso, deve-se avaliar o desempenho das iterações, observando os seguintes aspectos [13]:

- Qualidade resultante sob a perspectiva do cliente.
- Qualidade resultante sob a perspectiva técnica.
- O funcionamento do time de desenvolvimento e as práticas que os seus membros empregam.
- O status do projeto.

Essas informações permitem o maior entendimento do andamento do projeto e suas perspectivas, além de serem úteis para as conclusões que devem ser tiradas ao final do projeto, em seu *post-morten*.

Referências

[1] AGILE ALLIANCE WEB SITE. **Manifesto for Agile Software Development**. 2001. Manifesto com os princípios e valores dos métodos ágeis de desenvolvimento. Disponível em: <http://agilemanifesto.org/>. Acesso em: 24 de mar. 2003.

[2] ADVANCED DEVELOPMENT METHODS, INC. **SCRUM Software Development Process: Building The Best Possible Software**. ADM, 1995.

[3] ADVANCED DEVELOPMENT METHODS, INC. **ControlChaos.com**. 2003. Página do método ágil SCRUM. Disponível em: <http://www.controlchaos.com>. Acesso em: 28 de jul. 2003.

[4] BECK, K. **Extreme Programming Explained: Embrace Change**. Addison Wesley, 1999.

[5] BOEHM, B. **Software Engineering**. IEEE Transactions on Computers 25. Dez. 1976, n. 12, p.1226-41.

[6] COAD, P.; LEFEBVRE, E.; DE LUCA, J. **Java Modeling In Color With UML: Enterprise Components and Process**. Prentice Hall, 1999.

[7] COCKBURN, A. **Agile Software Development**. Addison Wesley, 2002.

[8] COCKBURN, A. **Crystal Light Methods Comments By Alistar Cockburn**. Cutter Consortium Whitepaper, 2002.

[9] COCKBURN, A. **Crystal Clear: A Human-Powered Methodology for Small Teams**. Addison Wesley. Rascunho. Disponível em: <http://members.aol.com/humansandt/crystal/clear/>.

[10] DYNAMIC SYSTEMS DEVELOPMENT METHOD LTD. **DSDM Consortium**, 2003. Site do consórcio que cuida do DSDM e onde estão disponíveis diversas informações sobre o método. Disponível em: <http://www.dsdm.org/>. Acesso em: 1 de ago. 2003.

[11] HIGHSMITH, J. **Retiring Lifecycle Dinosaurs**. Software Testing & Quality Engineering 2, n.4, July/August 2000.

[12] HIGHSMITH, J.; COCKBURN, A. **Agile Software Development: The Business of Innovation**. IEEE Computer, November 2001, p.120-122.

[13] HIGHSMITH, J. **Agile Software Development Ecosystems**. Addison Wesley, 2002.

[14] JACOBSON, I.; BOOCH, G.; RUMBAUGH, J. **The Unified Software Development Process**. Addison-Wesley, 1999.

[15] NEBULON PTY. LTD. **Nebulon Pty. Ltd**. Página do Feature-Driven Development, com o processo e outras informações. Disponível em: <http://www.nebulon.com>. Acesso em: 31 de jul. de 2003.

[16] NEWKIRK, J. **Introduction to Agile Processes and Extreme Programming**. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 2002. Anais Eletrônicos. Disponível em: <http://www.acm.org>. Acesso em: 24 de mar. 2003.

[17] PALMER, S. R.; FELSING, J. M. **A Practical Guide to Feature-Driven Development**. Prentice Hall, 2002.

[18] POPPENDIECK, M. **Lean Software Development**. C++ Magazine, 2003. Disponível em: <http://www.poppendieck.com/>. Acesso em: 4 de ago. 2003.

[19] POPPENDIECK, M.; POPPENDIECK, T. **Lean Software Development**. Addison-Wesley, 2003. Introdução disponível em: <http://www.poppendieck.com/>. Acesso em: 4 de ago. 2003.

[20] RISING, L.; JANOFF, N. S. **The Scrum Software Development Process for small Teams**. IEEE Software, v.17, n.4, July/August 2000, p.26-32.

[21] SIQUEIRA, F. L.; GIORGI, R. P.; USHISIMA, R. T. **Método de Comparação e análise de metodologias para o desenvolvimento de um sistema de discussão e colaboração**. Escola Politécnica, 2002.

[22] SOMMERVILLE, I. **Software Engineering**. 6 ed. Addison Wesley, 2001.