UNIVERSIDADE FEDERAL DE MINAS GERAIS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
ESPECIALIZAÇÃO EM INFORMÁTICA:
ÊNFASE: ENGENHARIA DE SOFTWARE

Uso de Metodologias Ágeis no Desenvolvimento de Software

por

Ebenezer Silva de Oliveira

Monografia de Final de Curso
CEI-ES-031-T2-2002-1

Prof. Renato Cardoso Mesquita
Orientador

Belo Horizonte, fevereiro de 2003.

UNIVERSIDADE FEDERAL DE MINAS GERAIS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
ESPECIALIZAÇÃO EM INFORMÁTICA:
ÊNFASE: ENGENHARIA DE SOFTWARE

Uso de Metodologias Ágeis no Desenvolvimento de Software

por

Ebenezer Silva de Oliveira

Monografia apresentada aos Senhores:

_____

Prof. Dr. Renato Cardoso Mesquita
(Orientador)

_____

Profa. Dra. Mariza A. S. Bigonha - Coordenadora do CEI-ES

Vista e permitida a impressão.
Belo Horizonte, ___/___/___.

Este trabalho é dedicado às pessoas que tem caminhado junto comigo, com as quais tenho aprendido tanto. Entre elas tenho alegria de destacar o companheirismo de Yolanda.
Dedico também aos meus pais, que investiram em mim, Ozias e Aparecida.

Agradeço à ATAN e à CST por disponibilizarem as informações citadas neste trabalho.
Agradeço também à ATAN pelo investimento nesta tecnologia que, entre outros aspectos, valoriza as pessoas.

Resumo

Palavras chave: Metodologias ágeis, *Extreme Programming*, *Scrum*

O objetivo deste trabalho é apresentar as metodologias ágeis como alternativa aos processos tradicionais usados na engenharia de software. Com este objetivo ele apresenta um estudo de caso – um sistema desenvolvido utilizando um processo baseado nestas metodologias.

O sistema apresentado é o Sistema de Embarque da Companhia Siderúrgica de Tubarão, desenvolvido por uma equipe da ATAN – Sistemas de Automação. As metodologias usadas no desenvolvimento deste sistema foram *Extreme Programming* e *Scrum*.

Abstract

Keywords: *Agile Methodologies, Extreme Programming, Scrum*

This work presents the agile methodologies as an alternative to the traditional (or heavy weight) process of software engineering. It presents a case study – the development of a system, which uses these agile methodologies.

The system presented is the Maritime Shipping System of CST (Companhia Siderúrgica de Tubarão – Brazil), being developed by a team of ATAN – Systems Automation. The methodologies used during the development of this system were Extreme Programming and Scrum.

Lista de Ilustrações

Lista de Abreviaturas ou Siglas

ATAN – Atan Sistemas de Automação – Empresa de automação industrial e comercial que oferece sistemas de informação, equipamentos e serviços a clientes dos mais diversos segmentos industriais, localizada em Belo Horizonte, MG.

C++ – Linguagem de programação de alto nível largamente utilizada para o desenvolvimento de software comercial.

CMM – *Capability Maturity Model*: Modelo de capacitação em desenvolvimento de software, elaborado pelo SEI (*Software Engineering Institute*) da *Carnegie Mellon University*.

COM – *Microsoft Componente Object Model*: Modelo para o desenvolvimento de componentes independentemente da linguagem de desenvolvimento utilizada.

CST – Companhia Siderúrgica de Tubarão – Siderúrgica líder em exportação de Aço no Brasil, localizada em Serra, ES.

DER – Diagrama de Entidades e Relacionamentos. O diagrama contém o modelamento lógico das entidades existentes no sistema e seus relacionamentos.

Excel – Aplicativo de escritório desenvolvido pela Microsoft para a manipulação de planilhas virtuais.

Microsoft – Empresa líder na área de desenvolvimento de sistemas de software. Atua em diversas áreas, desde sistemas operacionais até o desenvolvimento de plataformas para desenvolvimento de software.

SEI – *Software Engineering Institute*, da *Carnegie-Mellon University*.

Scrum – Metodologia ágil apresentada neste trabalho, cujo foco recai sobre as atividades de gerenciamento de projetos.

TI – Sigla para a expressão “Tecnologia da Informação”.

TPS – Sigla usada para referenciar o Terminal de Produtos Siderúrgicos da CST. O TPS é usado pela CST para despachar produtos via transporte marítimo e se localiza no porto de Praia Mole, no município de Serra, ES.

Windows – Família de sistemas operacionais desenvolvidos pela Microsoft.

XML – *Extensible Markup Language*: linguagem desenvolvida para a descrição de hierarquias de dados.

XP – Extreme Programming – Metodologia ágil apresentada neste trabalho, cujo foco está sobre as atividades de construção dos sistemas.

Sumário

1. INTRODUÇÃO

Desde a década de 1960, a Engenharia de Software tem procurado encontrar técnicas para garantir o sucesso dos projetos de software. Ao longo dos últimos 30 anos, temos convivido com as metodologias tradicionais de desenvolvimento de software. Estas metodologias são baseadas nas disciplinas tradicionais da engenharia como a Engenharia Civil ou Elétrica, onde o desenvolvimento do sistema é dividido em duas fases distintas: o sistema é primeiramente planejado e, depois do planejamento pronto, o sistema é construído [MCC03], [MIL02].

Infelizmente, a análise dos resultados apresentados pelo uso destas metodologias não é muito animadora. Os sistemas construídos freqüentemente extrapolam seus cronogramas e orçamentos. As equipes trabalham desmotivadas e acabam não sendo produtivas como deveriam ser. Os sistemas entregues são muitas vezes cheios de defeitos e nem sempre contêm o conjunto de funcionalidades que melhor atendem a seus usuários [FOW01].

A utilização das metodologias tradicionais nem sempre é viável. São necessárias equipes grandes que trabalham para gerar muitos documentos. As metodologias definem muitos papéis o que conseqüentemente requer muito gerenciamento.

O processo é basicamente preditivo: o sistema idealizado é construído e as mudanças realizadas no conjunto inicial de requisitos são punidas com faturamentos de extra-escopo [MIL02].

Este quadro desanimador tem trazido nos últimos anos aos engenheiros da área uma reflexão quanto a esta ser realmente a melhor forma de se fazer software. Do resultado desta reflexão surgem as metodologias ágeis para o desenvolvimento de software.

Estas novas metodologias se baseiam principalmente no fato de que a natureza dos projetos de software – o objeto de estudo da Engenharia de Software – é diferente das demais áreas da engenharia. O software possui características dinâmicas demais para ser construído de forma preditiva [FOW01].

Neste trabalho apresentam-se os principais conceitos envolvendo as metodologias ágeis. Apresentam-se com mais detalhes as duas metodologias – *Extreme Programming* (XP) e *Scrum* – que foram utilizadas no desenvolvimento do Sistema de Embarque da CST. Este estudo de caso mostra como as metodologias ágeis podem ser aplicadas no desenvolvimento de projetos de software.

2. REVISÃO BIBLIOGRÁFICA

Muito se tem falado sobre as metodologias ágeis para o desenvolvimento de software nos últimos anos. Muitos vêem estas metodologias como um antídoto à burocracia que as metodologias tradicionais representam. Outros as vêem como um modismo da década como muitos outros da área de TI; e ainda outros as vêem como uma oportunidade para produzir sistemas com baixa qualidade [FOW01].

O objetivo desta revisão é apresentar os principais princípios trazidos por estas metodologias. Como elas surgiram? Como elas pretendem resolver os problemas que elas apresentam? E como devem ser usadas? Estas são algumas perguntas que serão respondidas ao longo deste capítulo.

Em primeiro lugar é necessário compreender o problema que este novo modelo pretende resolver.

2.1. A Engenharia de Software

O desenvolvimento de software pode ser feito sem nenhuma metodologia, de forma caótica [PAU01], [FOW01]. Alguns autores chamam esta forma de desenvolvimento de "codifica-remenda" (do inglês – *"code and fix"*). A figura 1 abaixo (retirada de [PAU01]) representa esta situação.

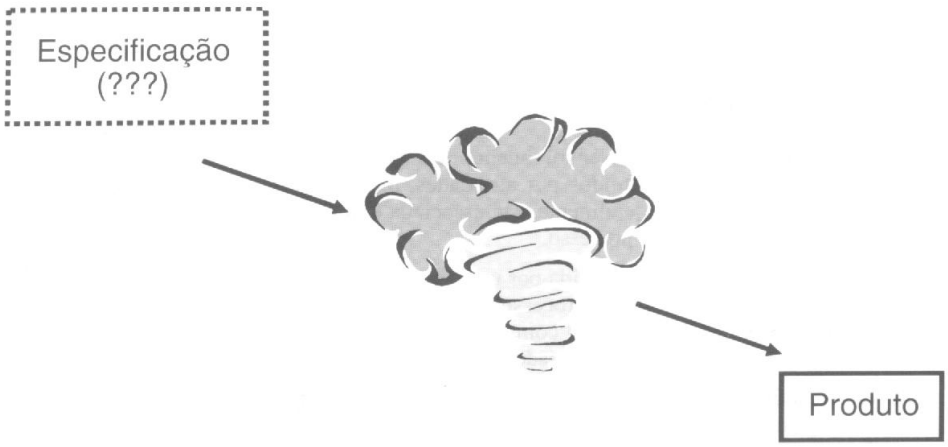

figura 1 – o desenvolvimento de um produto pode ser feito sem o uso de nenhuma metodologia [PAU01].

Este modelo funciona bem para um sistema pequeno. Porém, à medida que seu tamanho aumenta, se torna cada vez mais complexo adicionar novas características. Além disso, defeitos começam a aparecer com freqüência e complexidade cada vez maiores [PAU01].

Esta análise identifica claramente dois problemas distintos. O primeiro é que não é possível desenvolver sistemas grandes ou complexos desta forma. O segundo é que não basta construir o sistema, ele precisa estar preparado para crescer e para ser corrigido.

Um outro problema que não fica tão claro, mas que também deve ser destacado nesta análise, é que o desenvolvimento conduzido desta forma é imprevisível, em conseqüência direta de sua natureza caótica.

Como estes três problemas têm influência direta no negócio, concluímos que é impossível produzir software profissionalmente usando este tipo de desenvolvimento.

Estes problemas não são novos e têm sido estudados há aproximadamente 30 anos pela disciplina de Engenharia de Software [SAN02], [MCC03]. Neste estudo, o desenvolvimento de software é tratado como uma disciplina da engenharia. Um sistema deve ser primeiramente planejado, e somente quando o planejamento estiver correto, o projeto deve ser executado para evitar o mau uso de recursos. Pelo mesmo raciocínio, quanto mais planejado for o sistema, melhor ele será executado. Quanto mais detalhado, menores as possibilidades de insucesso [MIL02].

Como resultado destes princípios temos as metodologias tradicionais da engenharia de software. Estas metodologias também são chamadas na literatura de metodologias pesadas (*heavy weight methodologies*) ou metodologias monumentais – como uma forma de crítica ao seu tamanho e à dificuldade de serem implementadas na íntegra [FOW01].

2.2. As Metodologias Pesadas

A engenharia de software encara o software desenvolvido como um produto comercial, e como tal, o software tem um ciclo de vida. O ciclo de vida de um produto de software pode ser representado pela tabela 1 apresentada a seguir [PAU01]:

<table>
<tr><td rowspan="10">Ciclo de vida</td><td colspan="4">Percepção da necessidade</td></tr>
<tr><td rowspan="8">Desenvolvimento</td><td colspan="3">Concepção</td></tr>
<tr><td colspan="3">Elaboração</td></tr>
<tr><td rowspan="5">Construção</td><td colspan="2">Desenho inicial</td></tr>
<tr><td rowspan="3">Liberação</td><td>Desenho detalhado</td></tr>
<tr><td>Codificação</td></tr>
<tr><td>Testes de unidade</td></tr>
<tr><td colspan="2">Testes alfa</td></tr>
<tr><td colspan="3">Transição</td></tr>
<tr><td colspan="4">Operação</td></tr>
<tr><td colspan="4">Retirada</td></tr>
</table>

tabela 1 – ciclo de vida de um produto de produto de software [PAU01].

Geralmente o desenvolvimento de produtos de software é feito dentro de um projeto. O processo de desenvolvimento é dividido em fases caracterizadas pelo objetivo a ser alcançado ao final de sua execução. Tradicionalmente estas fases são divididas da seguinte forma: Levantamento de Requisitos, Análise, Desenho, Implementação, Testes e Implantação.

O desafio de qualquer metodologia de desenvolvimento de software é definir quando cada uma destas fases deve ocorrer dentro do projeto, quando ela deve iniciar e quando deve ser encerrada. A princípio esta é uma tarefa simples, pois existe uma ordem em que estas tarefas devem ocorrer. Se o desenvolvimento pudesse ocorrer totalmente sem erros e de uma maneira isolada (sem a influência de fatores externos) bastaria ordenar uma fase após a outra e nosso objetivo seria alcançado.

Uma das propostas mais simples para a condução de um projeto é desprezar a existência de tais fatores externos, e seqüenciar estas atividades conforme apresentado na figura 2 abaixo [PAU01]. Este modelo de ciclo de vida é chamado de **Cascata**.

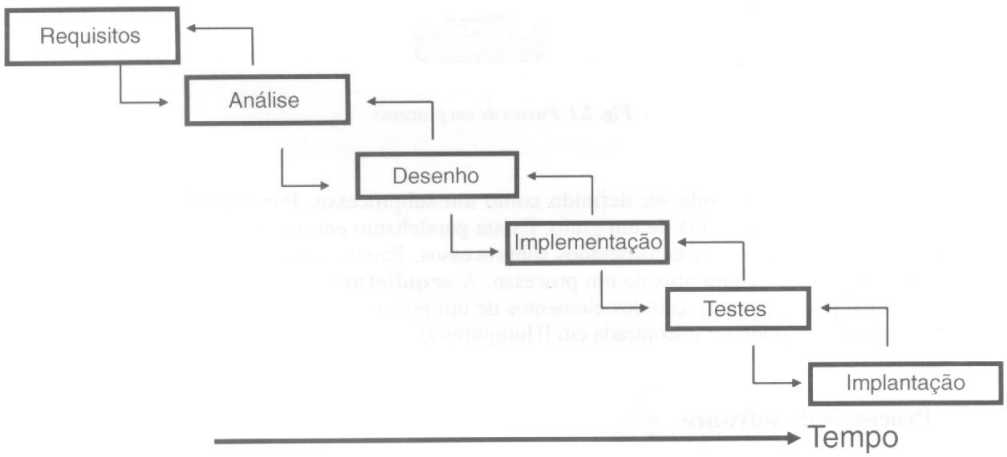

figura 2 – ciclo de vida em Cascata [PAU01].

Cada uma das etapas deve produzir um artefato que servirá de base para a fase seguinte. A vantagem deste modelo é a possibilidade de ordenar o caos descrito na figura 1. Mas o processo não prevê a possibilidade de ocorrência de erros durante a execução da fase anterior, ou de fatores externos causarem alterações no projeto durante a sua execução. Isto soa como um sistema hipotético sem a influência de fatores externos, ao qual qualificaríamos como sistema ideal. Como tal sistema não existe, percebemos que na prática é impossível desenvolver um projeto seguindo este modelo assim como ele é descrito.

Da evolução do modelo em cascata, surgiu o modelo **Sashimi**, apresentado na figura 3 abaixo [PAU01].

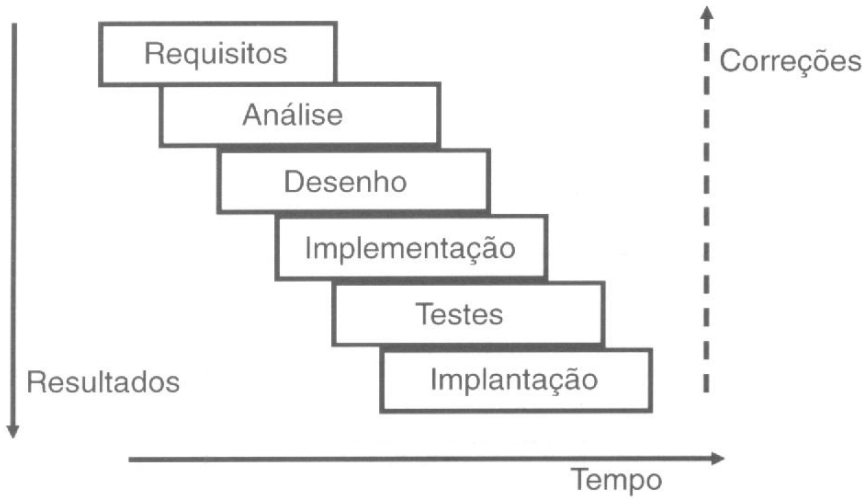

figura 3 – ciclo de vida em Sashimi [PAU01].

Neste modelo as atividades de uma fase podem influenciar nas atividades de uma fase anterior. Isto permite corrigir erros ocorridos em fases anteriores no momento em que eles são detectados.

A desvantagem deste modelo é que o gerenciamento do projeto se torna muito complexo. É difícil de saber quando uma fase está realmente encerrada, porque esta definição acaba dependendo da conclusão de outras fases [PAU01].

Uma outra desvantagem destes dois modelos apresentados é que o cliente e o usuário só virão a receber algum resultado ao final do projeto. Do ponto de utilização por parte do usuário isto significa que a validação dos requisitos do sistema somente pode ser efetivamente realizada após o término do projeto. Do ponto de vista de retorno do investimento, isto significa que os recursos investidos no sistema somente apresentam retorno após o final do projeto.

Não é isto que acontece em outras áreas da engenharia. Em muitos projetos (usinas hidrelétricas, sistemas de distribuição, shopping centers, etc...) após uma etapa inicial é possível iniciar o uso do sistema. A evolução é feita gradativamente, garantindo um retorno mais rápido do investimento e aumentando o conhecimento sobre o quê o usuário realmente espera do sistema.

Uma alternativa mais moderna apresentada pela engenharia de software é o modelo em **Espiral** apresentado na figura 4 a seguir [PAU01].

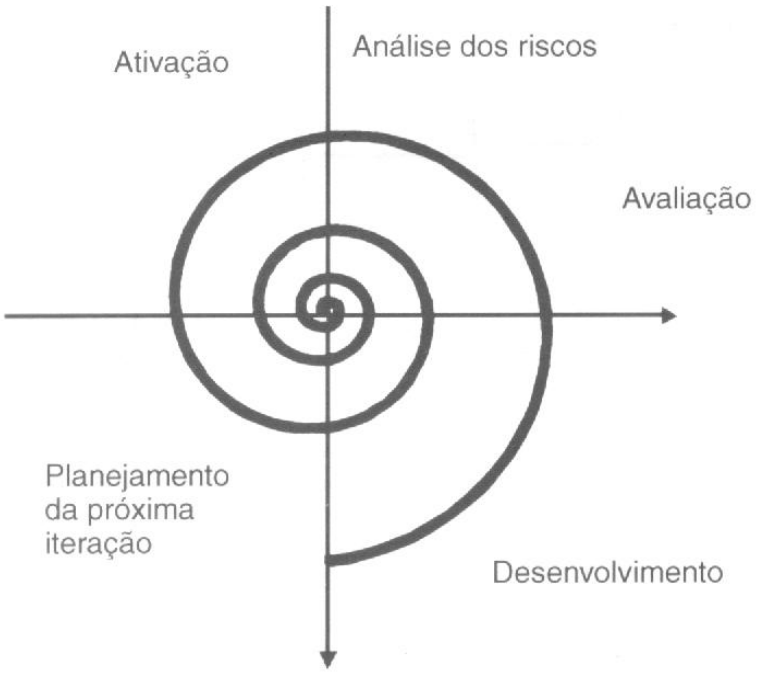

figura 4 – ciclo de vida em Espiral [PAU01].

Neste modelo, o sistema é desenvolvido em uma série de iterações. Cada iteração é representada por uma volta na espiral. Este modelo tem a vantagem de responder facilmente a erros em iterações anteriores, ou a modificações no projeto durante o processo de desenvolvimento. O principal problema apontado na literatura é que ele requer uma gestão muito sofisticada para ser previsível e confiável [PAU01]. Para sistemas muito dinâmicos, também existe o risco de que a inércia do modelo (se o número de documentos a serem atualizados for muito grande) imponha iterações muito grandes, ou não permita a evolução rápida do sistema.

Os modelos de ciclo de vida para projetos apresentados neste capítulo formam a base das metodologias tradicionais de desenvolvimento de software. Existem algumas

variações que são interessantes, mas que não serão citadas aqui porque se desviam completamente do escopo deste trabalho.

2.3. As Metodologias Ágeis

No final da década de 90, as bases das metodologias tradicionais começaram a ser questionadas [MIL02]. Dois principais motivos contribuíram para isto:

- a alta freqüência com que os projetos de software deixavam de cumprir seus cronogramas e extrapolavam seus orçamentos;
- a dificuldade de uso das metodologias pesadas.

Como fruto dos questionamentos levantados em torno destes problemas ao longo dos últimos anos, surge um novo paradigma para o desenvolvimento de software – as metodologias leves (*lightweight metodologies*) também chamadas de metodologias ágeis.

As metodologias ágeis apresentam diferenças marcantes em relação às metodologias tradicionais, entre as quais iremos destacar as seguintes:

- as metodologias ágeis são adaptativas ao invés de preditivas [FOW01];
- as metodologias ágeis são orientadas às pessoas e não a processos [FOW01];
- as metodologias ágeis são mais voltadas para o bem do negócio.

Estes aspectos serão discutidos com mais detalhes nos tópicos a seguir.

2.3.1. Predição ou adaptação?

As metodologias tradicionais de desenvolvimento de software são completamente baseadas na predição. Cada etapa de desenvolvimento do projeto é baseada na etapa anterior.

Isto funciona bem se a concepção do sistema não sofre nenhuma alteração. Quando se percebe que uma definição não foi a mais acertada, a tendência natural deste tipo de metodologia é resistir às mudanças. Isto prejudica a evolução do sistema.

As metodologias ágeis se baseiam no fato de que os sistemas mudam durante o desenvolvimento, pois à medida que o desenvolvimento ocorre, mais se conhece sobre o problema que o sistema deseja resolver. À medida que se conhece mais sobre o problema, melhores são as soluções encontradas.

Nas metodologias ágeis, os processos são orientados para adaptar o sistema a mudanças durante todo o tempo.

2.3.2. A questão dos requisitos

Um dos principais problemas (se não o principal) relacionados à predição em sistemas de software é a instabilidade dos requisitos. A natureza dos requisitos é dinâmica. Os requisitos de um sistema mudam constantemente.

Um dos motivos pelos quais os requisitos mudam é o limite da capacidade humana de prever o sistema como um todo. Isto acontece em projetos de qualquer porte,

mas quanto maior o sistema, maior a dificuldade de se realizar uma especificação sem nenhum erro.

A percepção que os usuários têm de suas necessidades também evolui à medida que eles conhecem o sistema. É difícil compreender o valor de uma determinada funcionalidade até que ela seja efetivamente usada, principalmente porque, não se pode requerer de um usuário comum a mesma capacidade de abstração que um desenvolvedor possui ao olhar um conjunto de requisitos. E mesmo que o sistema seja desenvolvido por desenvolvedores e para desenvolvedores, a percepção que o usuário tem de seu problema evolui à medida que ele conquista melhores mecanismos para atuar.

Os requisitos do sistema também mudam por fatores externos ao sistema. Leis, condições de mercado, ou mesmo fatores políticos dentro das organizações fazem com que os requisitos se tornem obsoletos e se tenha que alterar o que era considerado um bom conjunto de requisitos há alguns meses atrás.

Os processos tradicionais se baseiam na estabilidade dos requisitos. Como os requisitos têm uma natureza dinâmica, e os processos das metodologias pesadas são resistentes às mudanças, estes processos falham. As metodologias ágeis se adaptam constantemente ao conjunto de requisitos mais atual, e por isto têm chances de oferecer um melhor resultado aos projetos.

2.3.3. Desenho e construção

Nas metodologias tradicionais percebe-se uma distinção clara entre a fase de desenho e a de construção. Isto vem da herança de outras disciplinas de engenharia, como a civil, mecânica ou elétrica, onde estas fases têm um perfil completamente diferente.

Nestas engenharias, a fase de desenho geralmente é feita por uma mão de obra mais qualificada (e mais cara), que realiza toda a parte criativa do trabalho. Esta fase não é muito previsível. A fase de construção é feita por uma mão de obra menos qualificada (e mais barata), que executa o planejamento como ele foi feito. Esta fase é extremamente previsível.

Durante o desenvolvimento de um projeto de software, dificilmente obtém-se um desenho com detalhamento suficiente para que os seus artefatos sejam diretamente codificados, sem que seja necessário tomar novas decisões [MIL02]. As decisões são atividades de desenho, e requerem pessoas com habilidades para atividades criativas. Em conseqüência disto, ou os projetos voltam para a área de desenho constantemente até serem refinados o suficiente (o que torna o processo lento, e acumula atrasos no projeto como um todo), ou os programadores são obrigados a tomar decisões durante a fase de construção, o que a descaracteriza como tal.

Uma outra questão a ser analisada é a diferença de custos. Segundo Martin Fowler [FOW01], na construção de uma ponte por exemplo, os custos da etapa de desenho representam aproximadamente 10% do custo total de construção da ponte. Os outros 90% são consumidos durante a construção efetiva. Em software, esta relação praticamente se inverte: o tempo dedicado à codificação é aproximadamente 15% do tempo total de desenvolvimento do sistema.

Isto nos leva a compreender que devemos tomar cuidado ao comparar sistemas de software com sistemas estudados em outras engenharias. A natureza do software é diferente.

Uma outra idéia interessante, apresentada por Jeck Reeves e citada por [FOW01], sugere uma nova forma de analisar o desenvolvimento de software. O código escrito durante o desenvolvimento de um sistema faz parte de sua documentação de desenho. O serviço de construção de fato é realizado pelos compiladores ao gerar o código de máquina de um software, linkar as bibliotecas e gerar arquivos executáveis. Analisando desta forma, a etapa de construção é instantânea e praticamente gratuita.

Conclusões importantes: a atividade de codificação faz parte das atividades de desenho, e portanto, requer mão de obra qualificada para fazer o trabalho criativo e tomar decisões; atividades criativas são mais difíceis de se estimar, e por isto, trabalhar baseado em predição não é uma boa alternativa.

2.3.4. Uma alternativa à predição

As metodologias ágeis propõem que os projetos devam ser conduzidos de forma adaptativa. Isto é feito através de desenvolvimento iterativo. A idéia central é trabalhar com iterações curtas. Cada iteração entrega ao seu final um produto completo e pronto para ser usado, que contém a implementação de um novo subconjunto de características.

Esta idéia não é efetivamente nova. Formas de desenvolvimento iterativo já têm sido apresentadas pelas metodologias tradicionais em diferentes propostas (como o ciclo de vida em espiral, apresentado na seção 2.2 desta monografia).

A novidade apresentada pelas metodologias ágeis é que o sistema entregue em cada iteração é completo e pode ser colocado diretamente em produção. Na verdade, cada iteração produz uma versão do sistema final. O desenvolvimento é feito de forma evolutiva.

Ao final de cada iteração, usuários, clientes e desenvolvedores decidem sobre quais características devem ser adicionadas, quais devem ser modificadas, e até, quais devem ser retiradas do sistema. O sistema é desenvolvido da forma mais iterativa possível.

A documentação e *design* do sistema geralmente são feitos de maneira informal durante o desenvolvimento. Ao final da iteração, a documentação considerada essencial ao entendimento do sistema é gerada ou atualizada de forma objetiva, visando facilitar a manutenção. Documentos que não apresentam utilidade real no processo não são gerados. Muitas vezes, parte da documentação é gerada por ferramentas que automatizam o processo de documentação.

O tamanho das iterações é uma das questões chave sobre o desenvolvimento iterativo. Elas devem ser tão curtas quanto possível, desde que sejam suficientes para construir e testar um novo conjunto de funcionalidades. A ordem de grandeza sugerida na bibliografia da área é de três a quatro semanas.

2.3.5. Focalizando pessoas e não processos

As metodologias ágeis procuram usar a natureza humana a seu favor. Desenvolver software, como qualquer outra atividade profissional, dever ser uma experiência interessante para quem a pratica.

Por um outro lado, executar um processo adaptativo não é uma tarefa fácil. É preciso reunir pessoas talentosas. E somente reunir pessoas talentosas não é o suficiente:

elas precisam trabalhar bem como equipe. O resultado produzido deve ser maior que a soma das partes.

Uma das principais características das metodologias tradicionais em se tratando de relacionamento com seus colaboradores é que as pessoas são tratadas como recursos, classificados pelo tipo de atividade que desempenham. A equipe de um projeto geralmente é indicada por números, como por exemplo, 1 analista, 3 programadores e 2 testadores, 1 gerente, etc. As pessoas são tratadas como componentes, como partes removíveis e substituíveis.

Embora este tratamento facilite a predição, levanta-se a seguinte questão: as pessoas que trabalham em um projeto podem ser tratadas como componentes substituíveis? Uma das tendências das metodologias ágeis é rejeitar este tipo de pensamento.

De acordo com a citação de [FOW01], Alistair Cockburn apresenta uma idéia bem interessante sobre este assunto no artigo entitulado "Characterizing People as Non-Linear, First-Order Components in Software Development" ("Caracterizando as Pessoas como Componentes Não Lineares de Primeira Ordem no Desenvolvimento de Software"). A idéia central é que uma pessoa apresenta um ritmo de trabalho único e completamente diferente, que é fruto da sua caminhada pessoal (sua formação e suas atividades profissionais e não profissionais praticadas até o momento). A reação de uma pessoa a determinados desafios pode ser completamente diferente da reação de outra pessoa. Os fatores que determinam seu sucesso e fracasso podem ser únicos.

O resultado desta análise é que, tratar pessoas como componentes é simples demais e pouco eficiente. Para melhor gerenciar projetos de software, pessoas devem ser tratadas como indivíduos e não como recursos.

Outro resultado interessante é que a abordagem tradicional também tem outros efeitos colaterais danosos. Tratar pessoas como recursos, afeta o moral e em conseqüência, a produtividade da equipe. Além disso, a tendência é que os indivíduos que se destacam por seu talento acabem indo procurar lugares em que são reconhecidos [FOW01].

2.3.6. Gerenciamento de processos orientados a pessoas

Conduzir processos orientados a pessoas é muito diferente de conduzir processos orientados a processos. Isto se manifesta em vários aspectos das metodologias ágeis. Um dos pontos mais importantes é a questão da motivação dos colaboradores para adotar o processo. Quando o processo é aceito ao invés de ser imposto, os resultados são equipes mais motivadas e mais produtivas [FOW01].

Uma situação completamente adversa é quando a gerência informa à equipe a adoção de um processo. A situação fica ainda mais delicada se a figura da gerência não gera simpatia para com a equipe, ou se a equipe não confia na gerência em assuntos técnicos. Nestas condições, é natural que as pessoas resistam ao processo imposto [FOW01].

A situação é completamente simétrica nas metodologias ágeis. Em algumas metodologias que requerem alto grau de disciplina por parte dos integrantes das equipes, como XP, o uso da metodologia precisa passar por uma decisão pessoal de cada indivíduo da equipe [BEC00].

Nas metodologias ágeis esta filosofia é levada às últimas conseqüências. Em alguns casos, somente a equipe técnica está autorizada a estimar horas para as tarefas

pendentes. Um outro exemplo é a prática de os próprios indivíduos escolherem as tarefas que desenvolverão durante a próxima iteração [BEC00].

2.3.7. Processos versus Resultados

O objetivo final de um projeto de software é que o sistema produzido cumpra seu papel. Na maioria dos projetos, todas as pessoas envolvidas no processo (gerentes, desenvolvedores, clientes e usuários) desejam realmente o sucesso do projeto. Vamos refletir um pouco mais fundo sobre o que realmente significa a palavra **sucesso** nesta situação.

O objetivo final de um sistema de software é que os usuários tenham melhores condições de trabalho. O software deve permitir que o seu usuário produza mais e/ou melhor para que haja a compensação do custo do sistema. Isto significa não somente que o software deve atender às suas especificações, mas também que ele deve ser devidamente estudado para que os requisitos atendidos realmente reflitam as necessidades dos usuários. O software também deve funcionar bem dentro das condições para as quais foi desenvolvido.

Por um outro lado, quem investe no desenvolvimento do sistema também tem suas aspirações para o sistema. É preciso haver retorno do investimento feito, para que possam ser feitos novos investimentos. A melhoria das condições de trabalho para os usuários deve refletir em benefício para o cliente.

A empresa que fornece o software deseja fornecer um bom sistema, que atenda aos usuários e que agrade ao cliente. Isto gera retorno para a empresa através de novos contratos, melhores condições de venda e conseqüentemente, crescimento econômico.

A equipe que desenvolve o sistema deseja fazer um bom trabalho. É um sentimento comum nas áreas da engenharia, e na área de software não é diferente. Fazer parte de uma equipe de sucesso traz satisfação pessoal. Além disto, os profissionais desejam ser reconhecidos pelo seu trabalho e a fórmula mais simples para alcançar este objetivo é apresentar um bom trabalho.

Mas se todos os envolvidos efetivamente buscam o sucesso do projeto, porque tantos projetos de software nunca alcançam o sucesso? Porque é comum o fato de os projetos de software não cumprirem seus cronogramas, gastarem mais que o orçamento, apresentarem muitos *bugs* e, além de tudo, não atenderem completamente ao cliente? A única resposta plausível para esta questão é que existem erros no modelo tradicional de desenvolvimento [MIL02].

As metodologias ágeis, por sua natureza adaptativa e pelo seu foco em resultados, atende melhor às necessidades de cada uma das partes envolvidas no processo.

Os usuários têm a oportunidade de obter um sistema mais próximo de suas necessidades, pois o modelo permite que, durante o seu desenvolvimento, o valor de cada característica seja reavaliado. As funcionalidades dispensáveis podem ser abandonadas e as funcionalidades essenciais podem ser melhoradas.

O cliente tem um retorno mais rápido do seu investimento. Com o desenvolvimento através de iterações, as características que trazem mais resultados podem ser primeiramente implementadas. O retorno do investimento acontece antes do que aconteceria em outros casos, e pode influenciar de forma positiva o desenvolvimento do restante do sistema. Por outro lado, características que não agregam muito valor e que seriam incluídas no escopo de um sistema desenvolvido no modelo de

escopo fechado, acabam não sendo produzidas devido ao modelo iterativo do desenvolvimento.

Os desenvolvedores têm a oportunidade de trabalhar em um ambiente de melhor produtividade. Os resultados positivos aumentam o moral da equipe, e isto influencia naturalmente na sua produtividade, formando assim um ciclo que torna a atividade mais interessante. Não existe necessidade de trabalhar contra a natureza humana.

O fornecedor acaba sendo beneficiado trabalhando com equipes mais eficientes e produtivas. O sucesso dos projetos reflete no marketing da empresa, atraindo novos contratos e aumentando o seu valor de mercado.

O modelo de produção de software trazido pelas metodologias pesadas não busca uma análise crítica de seus processos. O fato de os erros no produto final serem encarados como deficiências na execução do processo é extremamente simplista.

Para concluir este tópico, é necessário lembrar que os processos tradicionais foram criados para o sucesso dos projetos de software. Se estes processos não conseguem garantir o sucesso dos projetos, eles merecem ser repensados. Se em algum momento se valorizou mais aos processos do que o software gerado através dele, é porque o foco do negócio foi completamente perdido.

2.4. Exemplos de Metodologias Ágeis

Neste tópico são apresentadas com mais detalhes as metodologias ágeis em uso no desenvolvimento do Sistema de Embarque da CST. As fontes de pesquisa para estes tópicos são [BEC00] e [SCH01].

2.4.1. Extreme Programming

Segundo [BEC00], *Extreme Programming* (ou simplesmente XP) *"é uma metodologia ágil para equipes pequenas e médias desenvolvendo software com requisitos vagos e em constante mudança"*.

XP nasceu no final dos anos 90, e tem como principal mentor Kent Beck – apontado como o autor da metodologia. Ele é autor de vários livros sobre o tema, entre eles "*Extreme Programming Explained*" [BEC00], uma das mais importantes publicações sobre o tema das metodologias ágeis.

Segundo os autores da metodologia, XP é código-cêntrica, ou seja, todo o desenvolvimento é voltado ao código. Em XP, a comunicação entre a equipe, ou entre equipes separadas geograficamente é feita através do código. Para se entender novas funcionalidades nos sistemas, os desenvolvedores olham o seu código. O ciclo de vida e o ambiente que envolve objetos complexos são definidos através de casos de teste, novamente no código. Problemas no software são demonstrados para a equipe com casos de testes que provam a sua existência, no código. O código é continuamente melhorado através do *refactoring*. O foco do desenvolvimento em XP é o código escrito para o sistema.

A expressão *Extreme Programming* surgiu da seguinte idéia: se fosse possível controlar o desenvolvimento de um sistema através de um painel, onde houvesse botões giratórios, e que cada botão representasse a importância de uma atividade no desenvolvimento do projeto, qual seria a configuração que permitiria os melhores

resultados? Segundo [BEC00], a principal idéia de XP é girar todos estes os botões para o seu valor máximo. O resultado é um conjunto de práticas que, ao serem usados em conjunto, produzem um resultado estável, previsível e flexível.

Embora estejamos falando de processos durante todo o tempo até agora, XP não se apresenta como um processo, e sim como um conjunto de práticas que, ao ser usado por completo, tem a capacidade de produzir resultados melhores que o que seria proporcionado pela soma dos resultados alcançados por cada prática individualmente.

Ao contrário do que muitos pensam, a adoção de XP requer muita disciplina. Para aproveitar ao máximo todas as suas vantagens, é necessário adotar todas as práticas de forma disciplinada.

XP traz consigo conquistas importantes. Merece destaque o artigo *"Extreme Programming from a CMM Perspective"*, de Mark C. Paulk, do Instituto de Engenharia de Software (SEI) da Carnegie Mellow University [PAU01a]. Segundo o autor, as práticas usadas em XP, se corretamente aplicadas, podem conduzir instituições aos níveis 4 ou 5 do CMM.

2.4.1.1. Os quatro valores de XP

Em muitas situações, os interesses pessoais entram em conflito com os interesses comuns. As sociedades têm aprendido a lidar com este problema desenvolvendo conjuntos de valores, muitas vezes apresentados através de leis, mitos, rituais ou punições. Através destes conjuntos de valores, as principais idéias das sociedades são eficientemente transmitidas.

XP apresenta um conjunto de quatro valores antes de apresentar as suas principais práticas. Estes valores são: comunicação, simplicidade, realimentação e coragem.

Comunicação

O primeiro valor de XP é a comunicação. Os problemas nos projetos podem ser invariavelmente rastreados até um ponto onde alguém não falou a alguém alguma coisa importante.

Existem várias situações que podem conduzir um projeto à falta de comunicação. Como exemplo, podemos citar um programador que é punido por um gerente por trazer más notícias, ou um cliente que é ignorado ao informar algo importante a um programador.

XP procura estimular a comunicação nos projetos através da adoção de práticas que não podem ser realizadas sem comunicação.

Simplicidade

O segundo valor de XP é a simplicidade. Muitos dos problemas do desenvolvimento de software vêm da tentativa dos programadores de tentar prever todas as possibilidades para execução de seu código. XP traz uma proposta de fazer qualquer módulo de sistema da forma mais simples que atenda aos requisitos, sem antecipar necessidades futuras. Se depois de pronto o módulo simples precisar de ajustes para atender a algum requisito novo, ele é alterado com este objetivo. O principal ganho que

vem desta análise é que a maioria dos requisitos que são antecipados pelos programadores no decorrer do projeto, nunca acabam sendo usados.

Comunicação e simplicidade possuem um relacionamento de suporte mútuo. Quanto mais se comunica, mais claramente se apresenta o que precisa ser feito, e conseqüentemente, mais claro fica o que não precisa ser feito. Quanto mais simples o sistema, menos comunicação ele requer. Quanto mais se comunica, mais simples o sistema se apresenta.

Realimentação (*feedback*)

O terceiro valor em XP é a realimentação. Uma realimentação real sobre o status do sistema no instante avaliado é uma vantagem sem preço. Programadores possuem uma tendência natural de serem otimistas em relação ao código. A realimentação fornecida pelos testes do sistema é a garantia de que tudo realmente vai bem. O uso de iterações curtas permite aos usuários e clientes fazerem uma avaliação do sistema assim que uma versão inicial é colocada em produção.

A realimentação no projeto é o que permite a condução do projeto na melhor direção em que ele pode andar durante todo o tempo. Não é possível acompanhar um processo sem ter como avaliá-lo. Por isto em XP, várias práticas proporcionam vários tipos de realimentação diferentes, procurando tornar o desenvolvimento o mais transparente possível a gerentes, desenvolvedores e clientes.

A realimentação trabalha em conjunto com simplicidade e comunicação. É impossível obter realimentação sem comunicação. Por outro lado, a comunicação efetiva faz a realimentação acontecer de forma natural. Sistemas simples são mais simples de testar, e conseqüentemente de avaliar, por isto contribuem para uma realimentação mais efetiva.

Coragem

O quarto valor de XP é coragem. O que fazer quando, depois de muito tempo passado no projeto, descobre-se uma falha na arquitetura do sistema? O que fazer se, depois de uma série de iterações desenvolvendo um módulo, descobre-se que ele não é necessário? A resposta a estas perguntas pode ser fácil, mas para mudar a arquitetura de um sistema, ou para eliminar código desnecessário em sistemas que estejam em funcionamento é preciso coragem. Para corrigir erros em sistemas em produção é preciso coragem.

É preciso ter coragem para modificar partes que funcionam no sistema para torná-las melhores. É necessário coragem para realizar grandes mudanças no decorrer do projeto. É necessário coragem para tentar coisas pouco prováveis, e até mesmo para errar e retornar ao ponto onde o sistema estava anteriormente.

É importante destacar aqui que ter coragem, sem ter comunicação, simplicidade e realimentação, pode ser um tipo de estupidez. Porém, se combinada com os outros valores, ela se torna necessária para que o processo seja executado de forma leve. A comunicação dá suporte à coragem pelo fato de que, se comunicando melhor, se conhece mais sobre os riscos envolvidos. A simplicidade também atua de forma conjunta, pois é mais fácil ser corajoso num sistema simples do que em um sistema complexo. A realimentação por sua vez coopera para uma avaliação rápida das hipóteses, e até mesmo de decisões tomadas.

2.4.1.2. As práticas de XP

Todos estes valores apresentados na seção anterior são importantes. Por esta razão, devem ser desenvolvidas formas de fazer com que se tornem práticas, hábito, para que eles deixem de ser apenas um conjunto de boas intenções. XP define um conjunto de práticas que se completam com o objetivo de colocar estes quatro valores em prática.

Planejamento (*The planning game*)

Em XP, o planejamento é detalhado para releases próximas e imaginado para as *releases* distantes. Para se fazer o planejamento é necessário adequar decisões técnicas e gerenciais, de forma que sejam tomadas as decisões que atendam as necessidades do projeto como um todo.

As responsabilidades no planejamento podem ser divididas entre responsabilidades técnicas e gerenciais. Entre as responsabilidades técnicas, podemos destacar as estimativas de tempos para as solicitações, as conseqüências de algumas decisões gerenciais (como, por exemplo, qual plataforma a ser usada no sistema), organização do time de desenvolvimento e cronograma detalhado (dentro de uma release, em que ordem serão desenvolvidas as atividades previstas).

Entre as responsabilidades gerenciais, podemos destacar: o escopo de uma funcionalidade, a prioridade de uma característica em relação às outras, a composição de uma *release* e as datas importantes do projeto.

Releases pequenas

Todas as *releases* devem ser tão pequenas quanto possível, e devem conter o máximo de valor comercial possível. Isto significa que não justifica dividir funcionalidades ao meio para fazer uma *release*. O planejamento detalhado deve ser feito para uma ou duas *releases*.

Metáfora

Todo projeto de software que usa XP deve ser conduzido por uma metáfora que ilustre o sistema. Uma metáfora é uma forma simples de demonstrar características do sistema, ou o sistema como um todo. Um exemplo de metáfora é ilustrar a tela inicial do computador como uma escrivaninha, ou o editor de textos como uma máquina de escrever.

Uma metáfora é uma idéia que permite às pessoas entenderem os principais objetivos das funcionalidades que compõem o sistema. O uso da metáfora permite uma comunicação fácil entre as pessoas dentro do projeto, inclusive entre a área gerencial e a área técnica.

Desenho simples

O desenho correto para qualquer software, em qualquer momento, deve atender aos seguintes quesitos:

- deve rodar todos os testes;
- não deve possuir lógica duplicada em nenhum ponto;
- deve demonstrar as intenções do programador (o artigo [FOW01a] apresenta idéias muito interessantes sobre o assunto e mecere apreciação);
- ter o mínimo de classes e métodos possíveis.

Cada parte de desenho do sistema deve justificar sua existência nestes termos.

Testes Automatizados

Em XP, qualquer característica desenvolvida que não possua um teste automatizado, simplesmente não existe. Os programadores devem escrever testes para as unidades desenvolvidas por eles. Os testadores (ou o cliente) devem escrever testes funcionais para verificar as funcionalidades do sistema. Como resultado, o sistema vai ficando mais confiável (e não menos) à medida que evolui.

Refactoring

Ao implementar uma característica em um sistema, o programador procura a forma mais simples de fazer as alterações necessárias. Depois de feita a alteração, o programador deve verificar se é possível simplificar o sistema de forma que todos os testes continuem rodando. Isso é *refactoring*.

Se um programador tem a opção de incluir uma característica em um sistema simplesmente acrescentando código em 1 minuto, ou refazer uma parte do sistema em 10 minutos de forma que ele fique mais simples e atenda aos testes, ele deve gastar os 10 minutos. Esta prática contribuirá para que o sistema permaneça simples.

Programação em Pares

Todo código de produção do sistema deve ser escrito com duas pessoas trabalhando em uma máquina, com um teclado e um mouse. Enquanto um programador está no teclado implementando um método, o seu parceiro deve estar olhando o desenho e pensando de forma mais estratégica sobre o código que está sendo gerado.

Os pares devem ser dinâmicos. Os pares devem revezar, entre si, quem fica no teclado e quem observa o desenho. Os pares devem se revezar também entre as pessoas da equipe. Se um programador precisa desenvolver uma característica que envolva uma parte do sistema que ele não conhece profundamente, ele deve procurar alguém que conheça especificamente aquela parte para programar em dupla. No final desta tarefa, os dois conhecerão melhor aquela parte do sistema.

Propriedade Coletiva

Qualquer pessoa que perceba uma boa oportunidade de agregar valor a qualquer parte do código deve fazê-lo, em qualquer hora. As partes do sistema não pertencem a nenhum responsável especificamente, ao mesmo tempo em que todos os envolvidos são responsáveis por todo o sistema.

É verdade que as pessoas da equipe não conhecem igualmente todo o sistema, ao mesmo tempo em que todos sabem pelo menos alguma coisa sobre todas as partes. Se um par está trabalhando em um código e percebe uma oportunidade de melhorar este código, ele deve aproveitar a oportunidade.

Integração Contínua

Novas versões são integradas e testadas com intervalos pequenos (às vezes poucas horas). Uma forma simples de implementar isto é ter uma máquina de integração que não pertença especificamente a nenhum programador.

Quando um par termina de escrever uma funcionalidade, aguarda até que a máquina de integração esteja livre. Assim que a máquina de integração estiver livre, eles carregam a última versão de código na máquina, carregam suas alterações e executam os testes de integração. Se houver erros, eles devem ser corrigidos para que esta se torne a versão corrente. Se não for possível retirar todos os erros, a última versão deve ser recarregada na máquina de integração.

A máquina de integração sempre estará com a versão mais recente, e que contém o maior número de funcionalidades devidamente testadas.

Semana de 40 horas

Horas-extras geralmente indicam a existência de problemas sérios no projeto. A regra de XP é a seguinte: não se pode trabalhar por duas semanas seguidas fazendo hora-extra. Por uma semana, este regime pode ser útil para alcançar objetivos e compensar possíveis erros. Mais do que isto, indica que existe um problema no projeto que não pode ser resolvido simplesmente por trabalhar mais horas.

Cliente *On-site*

Um futuro usuário do sistema, com conhecimentos técnicos e poder para tomar pequenas decisões deve fazer parte do time para retirar dúvidas, resolver disputas e definir prioridades de pequena escala.

É possível que existam objeções a esta prática. O gerente deverá avaliar o que custa mais caro: manter um usuário *on-site* e ter o sistema funcionando melhor e mais cedo, ou não ter este usuário e o sistema demorar mais para ficar pronto. Se o fato de ter o sistema pronto não agrega mais valor que ter um funcionário a mais para a sua construção, talvez o sistema não justifique o seu desenvolvimento.

Padrões de programação

O projeto tem um conjunto de desenvolvedores que está constantemente trabalhando em diferentes partes no sistema, trocando de parceiros de programação, e fazendo *refactoring* no código dos outros constantemente. É impossível que isto funcione sem conjuntos de padrões de programação.

O padrão deve ser tal que conduza ao mínimo de trabalho possível, de acordo com a prática de Desenho Simples. Este padrão deve enfatizar a comunicação e deve ser adotado pelo time inteiro de forma voluntária.

2.4.2. *Scrum*

Segundo os seus criadores, *Scrum* é um processo incremental e interativo para desenvolvimento de produtos ou para gerenciamento. Este trabalho se limita em apresentar a sua aplicação a projetos de software.

Diferentemente da maioria das outras metodologias que tratam o desenvolvimento a partir das tarefas de programação explicitamente, *Scrum* é orientado para o gerenciamento dos projetos de software. A literatura [FOW01] sugere que, utilizar *Scrum* em conjunto com outras metodologias – como XP – pode produzir resultados muito interessantes.

2.4.2.1. A Metodologia

Em *Scrum*, todo o desenvolvimento é feito em iterações. Todo o esforço é orientado de forma que seja apresentado um novo conjunto de funcionalidades ao final de cada iteração. Cada iteração tem um período de tempo definido (a duração sugerida é de 30 dias por iteração).

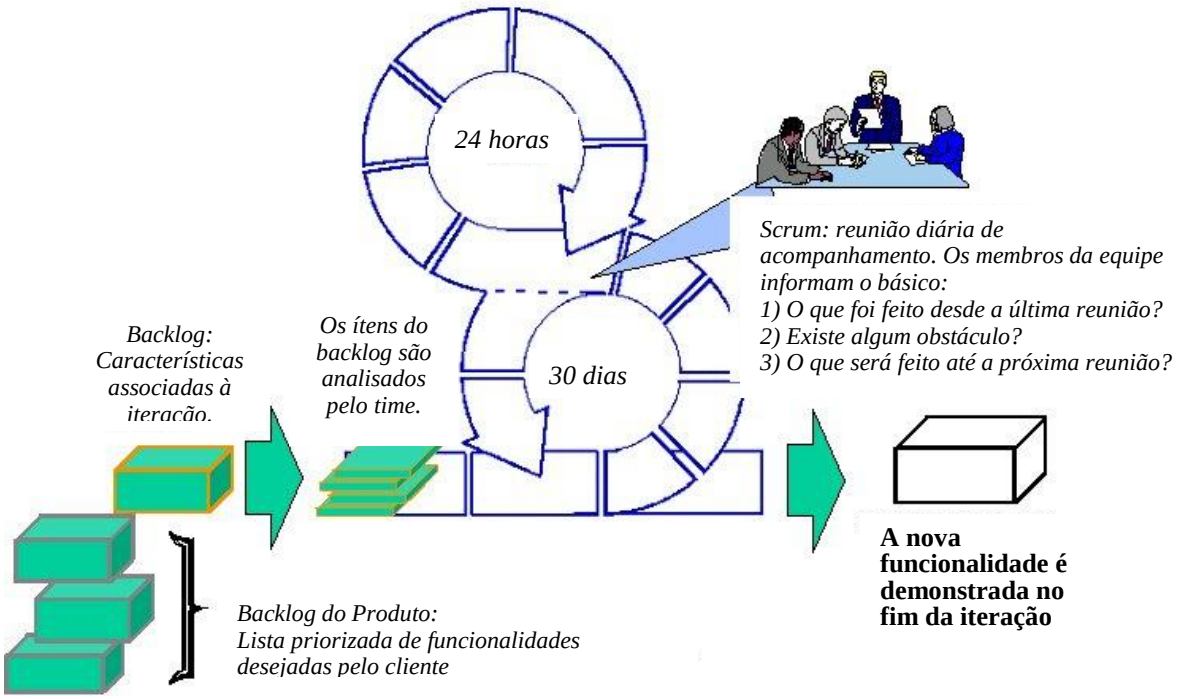

figura 5 – ciclo de vida de um projeto em Scrum [AGI03].

A figura 5 (retirada de [AGI03]) apresenta de forma simplificada o desenvolvimento de um projeto através de *Scrum*.

Os representantes de clientes e de usuários definem suas necessidades através de uma lista, chamada de *Backlog* pela metodologia. Os itens desta lista são ordenados pela sua prioridade, e apresentados à equipe de desenvolvimento que avalia o tempo necessário para atender a cada item da lista.

Com este resultado em mãos é definido o conteúdo da iteração, chamada de *Sprint*. Durante a iteração a equipe desenvolve os itens do *backlog* pertencentes à iteração. O status do projeto é informado diariamente através de reuniões rápidas.

Ao final da iteração a equipe entrega uma nova versão do sistema com o novo conjunto de funcionalidades implementado. O resultado da iteração é avaliado com o cliente, e o conteúdo da iteração seguinte é definido.

2.4.2.2. Principais práticas de *Scrum*

Da mesma forma que em XP, em *Scrum* o foco está em definir práticas e posturas a serem adotadas durante o desenvolvimento, e não em definir processos especificamente. Apresentam-se, a seguir, alguns detalhes sobre as principais práticas de *Scrum*.

Backlog do produto

O *backlog* é uma lista que contém todas as tarefas conhecidas, a serem desenvolvidas no sistema. Esta lista geralmente contém requisitos de todas as espécies, além de tecnologias, melhorias, e até correções de bugs do sistema. Em resumo, o *backlog* representa todo o trabalho a ser feito no sistema.

A lista começa vazia no início do projeto. Muitas vezes ela contém apenas idéias sobre o que o sistema deve vir a ser. Ela é dinâmica, e sempre ordenada por prioridades. À medida que novas necessidades são identificadas no sistema, elas são incluídas na lista, de acordo com a sua prioridade.

As tarefas de maior prioridade ficam no início da fila. Como são de maior prioridade, estas atividades ganham maior atenção, e por isto, geralmente são melhor conhecidas, já foram definidas com um maior detalhamento e podem ser implementadas com segurança e previsibilidade imediatamente.

A manutenção do *backlog* é feita exclusivamente por uma pessoa, chamada na metodologia de *Product Owner*. Este papel geralmente é ocupado pelo gerente do produto em desenvolvimento, ou por alguém que responda pelo produto. É importante destacar que o *Product Owner* é uma pessoa, e não um time ou um comitê.

Equipes

As equipes (*Scrum Team*) são as responsáveis por produzir o trabalho técnico durante a iteração. A equipe deve concordar em executar o objetivo da iteração. A equipe tem completa autoridade para decidir o que é necessário para desenvolver o seu trabalho durante a iteração.

As equipes em *Scrum* devem ser formadas por um conjunto de cinco a nove pessoas. Equipes muito pequenas não produzem tanto quanto poderiam se as pessoas

estivessem em equipes maiores, porque minimizam os ganhos obtidos pela sua sinergia. Equipes maiores que nove pessoas acabam gerando muita complexidade para um processo completamente empírico. Neste caso, a melhor opção é dividir uma equipe grande em duas equipes menores, que podem trabalhar no mesmo produto, em itens diferentes do *backlog*.

Outra figura importante definida em *Scrum* é o *Scrum Master*. Embora não seja considerado parte da equipe, ele é o responsável por garantir o uso correto da metodologia durante o andamento do projeto. Ele organiza a reunião diária e cria condições para que ela aconteça. Ele retira os obstáculos que estejam impedindo as equipes de prosseguir em suas atividades. Ele também é o responsável por informar a outras equipes sobre o andamento do projeto, no caso de várias partes de um sistema estarem sendo desenvolvidas em paralelo.

As equipes devem possuir pessoas com diferentes perfis, de forma que todas as habilidades necessárias para atingir o objetivo da iteração (o *Sprint Goal*) estejam presentes no time. A metodologia também sugere que se tenha pelo menos um engenheiro experiente para orientar os engenheiros juniores.

A equipe é responsável por desenvolver, testar e documentar todas os itens desenvolvidos. Por este motivo, em muitas vezes é interessante que haja pelo menos uma pessoa com experiência em testes e em documentação técnica de projetos de software, de acordo com os padrões da empresa.

Com o passar do tempo, a equipe se desenvolve em consequência da auto-organização. As pessoas descobrem como melhor podem contribuir com o desenvolvimento da equipe, ao mesmo tempo em que a equipe descobre os melhores indivíduos para desenvolverem as tarefas.

Durante a iteração, a equipe não deve ser alterada. Somente ao final da iteração é permitido trazer novos talentos ou substituir pessoas que não apresentam o rendimento esperado. É importante lembrar que, sempre que a equipe é alterada, a sua produtividade cai, até que alcance o estado de auto-organização novamente.

As equipes têm o direito e o dever de desenvolver somente os ítens no topo da lista de *backlog*. Ninguém pode interferir no trabalho da equipe, indicando novas tarefas aos desenvolvedores. A única forma de atribuir atividades aos desenvolvedores é através do *backlog*.

Reunião Diária de Acompanhamento (Daily Scrum Meeting)

O desenvolvimento de software é um processo complexo que requer constantemente um alto nível de comunicação. A reunião de planejameto é a ferramenta usada em *Scrum* para garantir que um mínimo de comunicação ocorra dentro do time.

A reunião de acompanhamento é uma reunião diária cuja duração não deve ultrapassar quinze minutos e que todos os integrantes do time precisam participar. O objetivo desta reunião é informar a todos os integrantes do time o que cada um está fazendo, e como o time está caminhando para alcançar o objetivo da iteração.

Durante a reunião de acompanhamento, cada integrante responde a 3 perguntas:

- o que foi feito desde a última reunião de acompanhamento;
- o que será feito até a próxima reunião;
- quais obstáculos estão atrapalhando ou impedindo o trabalho.

Qualquer outro assunto que precise ser resolvido pelo time, ou por uma parte do time deve ser resolvido em outras reuniões, que podem até ser agendadas durante a reunião

de acompanhamento. O objetivo é que a reunião diária não represente um alto custo de tempo aos seus participantes.

O *Scrum Master* é o responsável por conduzir a reunião de acompanhamento. Ele deve garantir que a reunião não ultrapasse o tempo previsto para a sua execução, lembrando sempre aos participantes sobre as regras e garantindo que todos os integrantes sejam breves em suas respostas.

Através do acompanhamento diário da reunião de acompanhamento, o time e o *Scrum Master* podem compreender o *status* atual do projeto e quais são as perspectivas de andamento. Os obstáculos informados pela equipe devem ser atacados imediatamente pelo *Scrum Master* para minimizar as situações em que o time fica impedido de produzir.

Iterações (Sprints)

Todo projeto de software é amarrado por quatro variáveis distintas: tempo disponível, custo (em pessoas e recursos), qualidade do software gerado e requisitos atendidos (funcionalidades entregues). Durante uma iteração (*Sprint*) as três primeiras variáveis são fixadas. O tempo é fixado no período de 30 dias. O custo de desenvolvimento de software também é fixo pois envolve somente salários de empregados e o custo das instalações. A qualidade do software é um parâmetro definido pela instituição.

Antes do início da Sprint, acontece um encontro para planejamento da iteração, chamado pela metodologia de *Scrum Planning Meeting*. Desta reunião participam representantes dos clientes e usuários, o gerente e os integrantes do time de desenvolvimento. O objetivo da reunião é determinar o objetivo da iteração e as funcionalidades a serem construídas para alcançar este objetivo.

O time trabalha isoladamente por um período fixo de tempo chamado de *Sprint*. Ao final do período, o time se compromete a entregar um incremento no produto atual, a partir de uma lista de itens que contém os seus requisitos, e possivelmente algo mais. O grande desafio da iteração para a equipe é o de construir um produto previsível a partir de requisitos de complexidade imprevisível.

Ao final da iteração é realizado um outro encontro, chamado pela metodologia de *Sprint Review*. Este encontro tem uma característica mais informal, onde o time apresenta aos clientes, usuários e à gerência o incremento feito ao produto durante a iteração.

É um fato conhecido que o processo produz uma quantidade significativa de erros e correções. Isto pode ser encarado de uma forma positiva, entendendo que a medida que a equipe produz uma série de funcionalidades para o produto, ela também produz uma série de experiências de aprendizado cujo valor muitas vezes não pode ser mensurado.

3. ESTUDO DE CASO

Neste capítulo apresentam-se os principais pontos destacados da experiência de utilização de metodologias ágeis no Sistema de Embarque da CST.

3.1. O sistema de embarque da CST

Em meados de setembro de 2002, a CST procurou a ATAN para solicitar o desenvolvimento de uma interface em coletores de dados portáteis para o embarque de bobinas em seu Terminal de Produtos Siderúrgicos (TPS).

Em uma reunião realizada no dia 16/09/2002, a CST apresentou à ATAN as necessidades básicas do sistema. Ele deveria permitir ao usuário registrar o recebimento de bobinas no TPS, a movimentação das bobinas no pátio do TPS e o embarque de bobinas durante as operações de embarque.

Um outro requisito apresentado como imprescindível para o sistema era o seguinte: o primeiro embarque de bobina estava agendado para o dia 30/11/2002. Este embarque deveria ser feito com o sistema.

A conclusão tirada do primeiro encontro foi que o maior desafio para o sistema era o cumprimento do prazo. As funcionalidades poderiam ser negociadas, desde que fosse possível realizar o embarque do dia 30/11/2002.

3.2. Porque foram escolhidos XP e Scrum

No final do ano de 2001 e início do ano de 2002, a gerência da área de Tecnologia da Informação da ATAN estava preocupada com a observação dos pontos destacados no item 2.3 dessa monografia. Os projetos de software freqüentemente tinham um comportamento completamente diferente do previsto. A exceção era quando o processo preditivo tinha um índice de acerto que não comprometia orçamentos e cronogramas. A percepção obtida pela observação, pelo estudo de casos citados na literatura e pela própria experiência é que estes problemas ocorriam de forma generalizada nos projetos de software.

Este período coincidiu com o primeiro contato com o material de metodologias ágeis na organização. O primeiro material que trouxe um grande impacto foi o manifesto de Kent Beck [BEC00]. Como a recepção ao material foi positiva, aos poucos a metodologia foi se tornando conhecida e aceita na organização.

Durante o primeiro semestre de 2002 a gerência da área de tecnologia de informação incentivou a pesquisa sobre as tecnologias ágeis na organização. Os novos projetos da área começaram a utilizar algumas das práticas de XP.

Alguns autores indicavam o uso de XP em parceria com o *Scrum* [FOW01], [AGI03]. No início do segundo semestre, já havia alguma experiência em algumas práticas de XP e algumas idéias para implementar práticas de Scrum.

A proposta de desenvolvimento do Sistema de Embarque parecia uma ótima oportunidade para colocar em prática as idéias do *Scrum*. O tempo que poderia ser dedicado para a construção era de no máximo dois meses, o tempo exato de duas iterações.

Não havia nenhum tipo de detalhamento dos requisitos funcionais do sistema. Havia algumas idéias, alguns levantamentos preliminares realizados pela equipe de informática da própria CST, a experiência e o conhecimento que a equipe da CST tinha

de seus sistemas e usuários. Mas, mesmo assim, seria necessário fazer um levantamento inicial dos requisitos do sistema.

Quando a CST convidou a ATAN para propor o projeto do Sistema de Embarque, houve um consenso na ATAN que esta seria uma excelente oportunidade para reforçar as práticas já conhecidas e para implementar as novas idéias. A ATAN propôs à CST que fosse construída a primeira *release* do sistema com data de implantação no dia 30/11/2002, ainda que algumas funcionalidades estivessem simplificadas. A CST concordou com a proposta da ATAN para a condução do projeto.

3.3. O desenvolvimento do Sistema

Para o levantamento de requisitos do sistema, foi realizada uma primeira iteração de 3 semanas a partir do dia 23/09/2002. Os objetivos desta iteração eram: 1) Definir uma proposta de arquitetura do sistema de embarque; 2) Detalhar, na medida do possível, os requisitos para a primeira release do sistema que deveria entrar em operação na data estipulada; 3) Sugerir um plano de desenvolvimento para o software.

No planejamento inicial constavam as seguintes datas: a construção do sistema deveria ocorrer no período de 14/10/2002 a 14/11/2002 – pouco mais que 4 semanas de desenvolvimento. A implantação do sistema seria feita no período de 18 a 29/11/2002.

O mês de dezembro seria utilizado para estabilização dos requisitos e do software. O mês de janeiro estava reservado para as férias dos envolvidos na construção do sistema. No mês de fevereiro já haveria conhecimento suficiente sobre o sistema para a elaboração de uma nova *release*.

3.3.1. O levantamento de requisitos

Realizar uma primeira iteração somente para levantamento de requisitos e planejamento não é uma prática de XP. A idéia em XP é a de ir gerando versões do sistema a partir das novas solicitações trazidas à equipe. Através da dinâmica dos requisitos, o sistema tende a convergir para o sistema ideal.

O nosso objetivo ao redigir esta documentação não era de levantar um documento que conteria tudo o que seria feito e que amarraria o sistema. O objetivo era o de nivelar cliente e fornecedor, tanto no conhecimento sobre as necessidades do sistema quanto nas prováveis soluções. O trabalho deste período produziu três documentos: uma Especificação de Requisitos para o sistema, um Plano de Desenvolvimento de Software e uma Descrição da Arquitetura.

Não havia nenhum conhecimento prévio por parte da ATAN sobre o processo da CST para embarque de bobinas. Sendo assim, demoraria algum tempo até que a ATAN pudesse agregar algum valor ao sistema, em termos de análise. Realizar uma iteração de levantamento de requisitos resolveu este problema, além de propiciar ao cliente um período onde ele poderia ter uma visão prévia da arquitetura.

A Especificação de Requisitos enumerava todas as funcionalidades desejadas para o sistema, incluindo as que seriam desenvolvidas imediatamente e as que seriam desenvolvidas futuramente. As funcionalidades que seriam construídas na primeira *release* estavam detalhadas o suficiente para iniciar a construção do sistema. As outras foram descritas de acordo com o conhecimento existente.

O documento de Descrição da Arquitetura trazia uma sugestão para o DER (diagrama de entidades e relacionamentos) do sistema, diagramas explicando a arquitetura de hardware e de software sugeridas para o sistema e a citação das bibliotecas de base da ATAN voltadas para o desenvolvimento adaptativo.

O Plano de Desenvolvimento de Software apresentava a proposta da ATAN para o desenvolvimento do sistema de forma iterativa (através de *releases*) e descrevendo um resumo de como se pretendia implementar as práticas de metodologias ágeis no desenvolvimento do sistema.

3.3.2. O desenvolvimento do sistema

A seguir, apresenta-se um resumo da arquitetura do Sistema de Embarque da CST. Os coletores acessam remotamente o servidor de aplicações através de um serviço de Telnet. Um sistema da CST faz a validação do usuário e carrega uma lista de módulos disponíveis para o usuário.

O software do sistema é executado em uma janela de comando padrão console dos sistemas operacionais da família Windows. O software é escrito em C++, encapsulado dentro de um componente COM.

O sistema utiliza duas bibliotecas de base de propriedade da ATAN para aumentar a velocidade de produção e para possibilitar o desenvolvimento iterativo do sistema: uma biblioteca é usada para implementar o acesso ao banco de dados e a outra é usada para implementar a camada de apresentação. A interface do sistema é apresentada em uma janela de console. As informações sobre a formatação de cada tela do sistema são armazenadas em arquivos XML, que são carregados dinamicamente durante sua execução. A primeira release do sistema possuía um total de onze interfaces de usuário, onze classes persistentes e cinco interfaces de software com o sistema legado da CST.

O sistema foi desenvolvido no período de 14/10/2002 a 22/11/2002, por uma equipe de 3 pessoas na sede da ATAN. A implantação do sistema iniciou-se no dia 19/11/2002.

Durante o desenvolvimento do sistema, foram realizadas algumas mudanças profundas na sua concepção. O maior impacto veio com a descoberta de que a melhor abordagem para o problema seria expandir a idéia original de construir um sistema para embarque de bobinas e fazer do sistema em construção o sistema unificado para embarque de produtos. Na prática, isto implicou em preparar o sistema para trabalhar com o conceito de produtos no porto, e não de bobinas. Posteriormente as funcionalidades para o tratamento de placas seriam desenvolvidas.

No decorrer do sistema, outra alteração importante foi na data de entrada do sistema em operação. Houve problemas com a importação e instalação de equipamentos que inviabilizaram a entrada do sistema na data prevista. Esta instalação só foi concluída no final de dezembro.

Também houve um atraso durante a implantação do sistema. Os testes de aceitação só foram 100% aprovados no ambiente de produção no dia 06/12/2002, com uma semana de atraso. Não houve desgaste no relacionamento com o cliente neste sistema porque, durante a implantação, os atrasos decorrentes da instalação já eram conhecidos.

Ao final da implantação do sistema, a CST solicitou a adição das funcionalidades para o tratamento de placas no TPS. Neste aditivo foram incluídas uma nova interface de usuário, uma nova interface de software e uma nova classe persistente.

No decorrer do mês de janeiro de 2002 o sistema foi testado pela CST, e foram encontrados alguns erros de requisitos e de programação que não foram descobertos durante a implantação. Os erros foram corrigidos pela ATAN tão logo foram descobertos.

O sistema entrou em operação no início de fevereiro de 2003. A realimentação dado pelos usuários do sistema é de que as operações realizadas no sistema são confortáveis e que o sistema atende a todas as funcionalidades que estavam previstas para a primeira *release*. Já existem solicitações de outras funcionalidades para a próxima *release*.

3.4. Práticas implementadas

Destacamos a seguir as práticas XP e de Scrum que foram utilizadas no desenvolvimento do sistema.

Pela forma que o desenvolvimento da primeira *release* foi conduzido e pela forma que o sistema foi desenhado para as próximas *releases*, podemos considerar que o planejamento e a divisão em *releases* foram feitos segundo os princípios apresentados em XP e em *Scrum*.

Este trabalho não apresenta os detalhes do desenho do sistema, mas é válido citar que trabalhar com o desenho simples sempre aperfeiçoado através de *refactoring* foi uma das primeiras práticas a serem adotadas na ATAN. O material de [FOW01a] teve um papel importante na formação de opinião neste sentido.

A propriedade coletiva e o ritmo de integração contínua foram implementados através da ferramenta de controle de versões *Microsfot Source Safe*. Como não foram implementados os testes automatizados, não foi possível ter uma realimentação rápida do resultado da implantação de algumas versões.

A semana de 40 horas foi uma prática adotada. Um bom exemplo disto é que, quando a implantação do sistema estava sendo executada e descobrimos que precisaríamos de uma semana a mais, poderíamos ter trabalhado longas horas extras nas duas semanas de implantação e garantido o cumprimento das datas. Isto não teria trazido benefício algum ao sistema – muito pelo contrário. Em alguns momentos foi necessário que alguns membros da equipe trabalhassem algum período extra, mas isto não foi comum no desenvolvimento do sistema.

O controle de tarefas no sistema foi implementado através de uma planilha EXCEL baseada no *backlog* apresentado em *Scrum*. A planilha era atualizada diariamente com o tempo restante para o cumprimento das tarefas.

A equipe, embora não tivesse o tamanho recomendado por *Scrum*, trabalhou de uma forma bem semelhante à proposta nesta metodologia. A equipe, formada por um engenheiro sênior e dois *trainees*, apresentou ganhos por trabalhar em conjunto e teve independência suficiente para tomar todas as decisões a ela pertinentes.

Uma ferramenta que contribuiu em muito para o acompanhamento do projeto foi a reunião diária de acompanhamento. Através dela, a planilha de acompanhamento era atualizada diariamente e era possível avaliar o andamento do projeto em qualquer dia durante a etapa de construção.

3.5. Práticas não implementadas

Embora tenhamos adquirido muitas experiências interessantes em implementar uma série de práticas sugeridas na bibliografia, algumas práticas ainda não puderam ser implementadas.

Durante o desenvolvimento do sistema não conseguimos implantar um esquema automatizado para executar os testes do sistema. Sempre que era necessário executar o roteiro de testes, isto tinha que ser feito manualmente.

Uma outra prática que ainda é um desafio no desenvolvimento deste sistema é a programação em pares. O principal impedimento para a realização desta prática ainda é o custo. Um outro complicador é o seguinte: para que duas pessoas programem em pares da forma descrita na literatura, as duas pessoas precisam ter conhecimentos das ferramentas e da linguagem a um nível de profundidade semelhante, ou a atividade acaba se tornando improdutiva.

A prática XP de possuir um cliente como parte da equipe não foi implementada conforme descrito na documentação. Isto seria praticamente impossível pois as duas organizações envolvidas estão separadas por mais de 500 Km de distância. A lacuna deixada pelo descumprimento desta prática foi compensada pela intensa comunicação entre cliente e fornecedor. Um alto nível de comunicação foi mantido através de contatos telefônicos, e-mail e viagens do fornecedor ao cliente.

Quanto ao uso de padrões de programação, considera-se que esta prática não foi utilizada como poderia porque, mesmo que tenham sido utilizados muitos padrões para a realização de diversas rotinas utilizando as bibliotecas de base, ainda não existe uma política de utilização destes padrões na organização. Isto significa que algumas vezes os padrões são utilizados por iniciativas individuais, mas isto não caracteriza o uso de padrões no sistema.

3.6. Resultados obtidos

Para analisar o sucesso do desenvolvimento desta primeira release do sistema, usaremos o conceito de sucesso apresentado no item 2.3.7 dessa monografia. Segundo a apresentação feita, o software deve permitir que o usuário produza mais ou em melhores condições, e isto deve implicar em aumento de produtividade, para haver compensação do custo do sistema.

O sistema atende a este requisito segundo as análises feitas até o momento em que este trabalho foi escrito. Em primeiro lugar, não era possível aos trabalhadores do TPS efetuar as operações descritas com bobinas no sistema antigo. No sistema atual, todas as operações podem ser feitas para bobinas. As operações feitas com placas anteriormente eram anotadas em planilhas de papel e depois transferidas para o sistema através de um terminal que fica no escritório. Hoje, todas as operações podem ser feitas *on-line*. Embora ainda não exista nenhuma informação quantitativa sobre os resultados desta implantação, a simples mudança no processo é suficiente para demonstrar que haverá melhoras nas condições de trabalho e na produtividade.

O retorno do capital investido pode ser observado a partir do momento em que o sistema se tornou disponível, em fevereiro de 2003. Comparando com o tamanho do sistema a ser construído (deverá envolver uma média de 12 funcionalidades ao todo, contra 3 que foram implementadas na primeira *release*) é possível perceber que o

retorno acontece bem antes do que seria possível imaginar com os métodos tradicionais, baseados em predição.

Para o fornecedor, o sistema representa um bom negócio. O cliente satisfeito é uma garantia de novos contratos no futuro. E como a extensão da *release* é pequena, a estimativa de custos pode ser feita com mais conhecimento, o que diminui o risco de erro nesta estimativa. Um outro fator interessante a se destacar é que ainda que ocorra um erro na estimativa de uma *release,* devido ao seu tamanho reduzido o prejuízo em uma release não significa uma grande perda de capital. Se os erros não se sucederem, é possível que haja reversão do prejuízo ao longo do projeto.

A equipe trabalhou motivada por ver que o seu trabalho foi valorizado. Não houve grandes quantidades de horas-extras, não houve desgaste com os clientes nem com a gerência. As sugestões da equipe foram ouvidas, as decisões técnicas foram tomadas por quem de direito.

Outros resultados interessantes merecem ser destacados. O requisito mais importante no início do desenvolvimento do sistema foi alterado porque existiam outras variáveis envolvidas que influenciaram no processo. É impossível equacionar todas as variáveis que podem interferir nos requisitos do sistema. Isto nos leva a uma profunda reflexão sobre como são arriscadas as técnicas baseadas em predição.

O desenvolvimento do sistema ocorreu em um clima de total cooperação. Mais do que o cumprimento de um contrato de fornecimento, o desenvolvimento do sistema foi uma excelente demonstração do sucesso da parceria entre cliente e fornecedor, onde a transparência na tomada de decisões foi uma característica marcante. O bom relacionamento entre cliente e fornecedor favoreceu de maneira decisiva o bom andamento do projeto.

O resultado que podemos avaliar ao final deste trabalho é que, através de todas as perspectivas, pelo que se conhece até este momento, o projeto que desenvolve o sistema de embarque da CST é um sucesso.

É verdade que o desenvolvimento do sistema prossegue. Apenas um pequeno número de funcionalidades foi desenvolvido. Serão necessárias ainda algumas *releases* para que o sucesso do sistema possa ser analisado mais profundamente.

4. CONCLUSÃO

Este trabalho tem uma extensão curta, apresentando somente o estudo de caso do Sistema de Embarque da CST, que utilizou duas metodologias ágeis de forma conjunta. O uso destas duas metodologias desta forma é sugerido na bibliografia [FOW01] pois suas práticas são complementares.

Através dos resultados apresentados neste trabalho, e dentro das suas limitações, podemos concluir que as práticas apontadas pelas metodologias ágeis apresentadas são eficientes para desenvolver sistemas, desde que usadas corretamente.

As metodologias ágeis surgem de uma mesma motivação e apresentam as principais idéias em comum. O que identifica cada uma de forma unívoca é a forma de implementar as idéias através de práticas para o dia a dia. Cada conjunto de práticas deve ser examinado de maneira isolada, conforme feito neste trabalho, para verificar sua eficiência.

De uma maneira geral, pelo material apresentado neste trabalho, entendemos que para o desenvolvimento de software, as metodologias baseadas em processos adaptativos têm potencial de apresentar resultados melhores que os resultados apresentados pelas metodologias baseadas em processos preditivos. Para as pessoas que desenvolvem o sistema, as metodologias orientadas a pessoas apresentam melhores condições de trabalho e melhores resultados que as metodologias orientadas a processos.

Conforme foi apresentado no desenvolvimento deste trabalho, a ATAN é uma organização em transição e por isto nem todas as práticas foram implementadas ainda. As principais práticas a serem implementadas hoje são as de Testes Automatizados e Padrões de Desenvolvimento. A implantação destas duas práticas é hoje objeto de estudos dentro da organização. Quando implantadas, deverão aumentar a qualidade do software gerado e a realimentação durante o desenvolvimento do projeto. Também deverão favorecer diretamente as práticas de Propriedade Coletiva do código, *Refactoring* e Integração Contínua, que por sua vez favorecem o desenvolvimento adaptativo.

Para a implantação das práticas que já estão sendo utilizadas na ATAN, alguns fatores contribuíram de forma decisiva. Entre eles, podemos destacar três fatores de grande importância.

Em primeiro lugar, para desenvolver sistemas de forma adaptativa é necessário ter uma equipe capacitada. As pessoas que compõem o time precisam ter conhecimentos profundos sobre as melhores práticas de desenvolvimento no ambiente a ser utilizado e estar constantemente atualizadas sobre as tecnologias mais recentes. Também devem ter um perfil maleável e uma base sólida para aprender a lidar com novas tecnologias ao longo dos projetos.

Em segundo lugar, trabalhar com tecnologias orientadas a objetos também favorece muito o caráter adaptativo do desenvolvimento. O software orientado a objetos, se corretamente modelado, proporciona caminhos muito mais simples para realizar alterações que sistemas desenhados na arquitetura *top down*.

Em terceiro lugar, as ferramentas utilizadas ao longo do desenvolvimento, como por exemplo os geradores automáticos de código, agilizam o processo de atualização do sistema após mudanças nos requisitos. O desenvolvimento interno destas ferramentas, e o uso de ferramentas comercias têm proporcionado à ATAN um importante diferencial competitivo para implementar as práticas das metodologias ágeis.

As metodologias ágeis são uma nova escola para o desenvolvimento de software, mais moderna e mais preparada para lidar com os problemas do novo século. Os

desafios para a Engenharia de Software hoje são um pouco diferentes dos desafios de 30 anos atrás. Os sistemas são mais complexos e os requisitos mais dinâmicos. Isto nos leva a concluir que as metodologias ágeis deverão tomar cada dia mais espaço no mercado de desenvolvimento de software.

5. REFERÊNCIAS BIBLIOGRÁFICAS

[AGI03] AGILLE ALIANCE. “What is Scrum?”. Advanced Development Methods, 2003. Disponível por WWW em
http://www.controlchaos.com/.
Acesso em: BH, 15/02/2003, 1p.

[BEC00] BECK, Kent. Extreme Programming Explained: Embrace Change. Addison-Wesley, 2000, 190p.

[FOW01] FOWLER, Martin. The New Methodology. Disponível por WWW em
http://www.martinfowler.com/articles/newMethodology.html
Acesso em: BH, 28/02/2003.

[FOW01a] FOWLER, Martin. To Be Explicit. Disponível por WWW em
http://www.martinfowler.com/articles/explicit.pdf
Acesso em: BH, 28/02/2002

[MCC03] MCCLURE, Robert M. The NATO Software Engineering Conferences. Disponível por WWW em
http://www.cs.ncl.ac.uk/old/people/brian.randell/home.formal/NATO/index.html.
Acesso em: BH, 09/02/2003.

[MIL02] MILLER, David. SLAYING THE DRAGONS: AN AGILE APROACH TO SOFTWARE DEVELOPMENT – A Management Overview. Disponível por WWW em
http://www.cirrustech.com.au/downloads/slayingdragons.pdf
Acesso em: BH, 28/02/2002

[PAU01] PAULA FILHO, Wilson de Pádua. Engenharia de Software Fundamentos, Métodos e Padrões. LTC Livros Técnicos e Científicos, 2001, 584p.

[PAU01a] PAULK, Mark C. Extreme Programming from a CMM Perspective. IEEE Software, November/December 2001, p:1-8.

[SAN02] SANTOS, Jefferson de Barros. Extraindo o Melhor de XP, Agile Modeling e RUP para Melhor Produzir Software. Disponível por WWW em
http://www.xispe.com.br/evento2002/download.html
Acesso em: BH, 28/02/2002.

[SCH01] SCHWABER, Ken, BEEDLE, Mike. Agile Software Development with Scrum. Prentice Hall, 2001, 158p.