UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

NÚCLEO DE COMPUTAÇÃO ELETRÔNICA

UM ESTUDO DE CASO DA ADOÇÃO DAS PRÁTICAS E VALORES DO EXTREME PROGRAMMING

VINÍCIUS MANHÃES TELES

IM-NCE/UFRJ
Mestrado em Informática

Orientador:

Carlo Emmmanoel Tolla de Oliveira, Ph.D.

Rio de Janeiro

2005

UM ESTUDO DE CASO DA ADOÇÃO DAS PRÁTICAS E VALORES DO EXTREME PROGRAMMING

VINÍCIUS MANHÃES TELES

Dissertação submetida ao corpo docente da Coordenação do Instituto de Matemática e Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro - UFRJ, como parte dos requisitos necessários à obtenção do grau de Mestre em Informática.

Aprovada por:

Prof._____ (Orientador)
Carlo Emmanoel Tolla de Oliveira, Ph.D., UFRJ

Prof._____
Lígia Alves Barros, D.Sc., UFRJ

Prof._____
Alfredo Goldman vel Lejbman, Ph. D., USP

Rio de Janeiro, 28 de março de 2005

MANHÃES TELES, VINÍCIUS.

UM ESTUDO DE CASO DA ADOÇÃO DAS PRÁTICAS E VALORES DO EXTREME PROGRAMMING / Vinícius Manhães Teles. Rio de Janeiro: UFRJ / IM / DCC, 2005.

180 pp.

Dissertação (Mestrado em Informática) – Universidade Federal do Rio de Janeiro - UFRJ, IM / DCC, 2005.

Orientador: Prof. Carlo Emmanoel Tolla de Oliveira

1. Extreme Programming. 2. Engenharia de Software.
I. IM/NCE/UFRJ. II. Título (série).

AGRADECIMENTOS

À minha esposa, Patrícia Figueira, por sua enorme paciência, por todas as noites em que se privou da minha companhia, por suportar meu mau humor na correria da última hora, por todas as alegrias que preenchem o nosso dia-a-dia e pelo seu apoio incondicional.

Ao meu orientador, Prof. Carlo Emmanoel Tolla de Oliveira por aceitar e apoiar o tema da dissertação, pelas sugestões de leitura, pela disponibilidade e pelas observações.

À Profa. Ligia Alves Barros por seus ensinamentos, paciência, disponibilidade e, sobretudo, pelas inúmeras críticas construtivas que fez em torno deste trabalho.

Ao Prof. Alfredo Goldman vel Lejbman pela disponibilidade de vir ao Rio de Janeiro para a defesa desta dissertação, pelo tempo dedicado e pelo trabalho que vem executando na disseminação do XP no Brasil.

Ao Oswaldo Areal, do Grupo XP Rio, que pacientemente revisou grande parte do texto.

Aos meus alunos do curso de Ciência da Computação da UFRJ que, através de seus questionamentos, me levaram a estudar mais, pesquisar mais, e aprimorar meus conhecimentos sobre Extreme Programming.

Aos meus amigos Rodrigo Fernandes, Gilson Tavares, Renato Fiche Junior, Maurício Emanuel Dourado Cescato e Eduardo Bastos Leite por questionarem o XP inicialmente e forçarem o aprimoramento de minha argumentação. E, mais tarde, por apoiarem a aplicação do XP de forma entusiástica e determinada.

A toda a equipe da Improve It, pela coragem de adotar o XP, pelo talento demonstrado na utilização das práticas, por acreditar, por tentar, por melhorar continuamente e por mostrar

que é possível conduzir projetos de software com XP de maneira primorosa. Sobretudo, obrigado por todas as vezes que vocês conseguiram, usando o XP, facilitar a vida de nossos clientes.

Aos clientes da Improve It, por nos concederem a oportunidade de experimentar o XP.

À comunidade XP do Brasil, que através de encontros, congressos e discussões na Internet me ajudaram a consolidar o aprendizado que deu origem a este trabalho.

Aos companheiros da comunidade internacional Kent Beck, Mary Poppendieck, Ron Jeffries, Ward Cunningham e Laurie Williams, por tudo o que aprendi com vocês em nossos encontros e através de suas publicações.

Ao meu editor, Rubens Prates, da Novatec Editora, pela oportunidade que me concedeu de escrever um livro sobre XP.

Aos funcionários da Secretaria do Departamento de Ciência da Computação do IM/UFRJ, em especial à querida Tia Deise Lobo Cavalcante pelo seu carinho e atenção ao longo de todo o mestrado e da graduação. Aos funcionários da Biblioteca do Núcleo de Computação Eletrônica da UFRJ, em especial a Raquel de Melo Porto, pela presteza, profissionalismo e por me ajudarem a localizar e obter os artigos de que necessitei.

Finalmente, um muitíssimo obrigado a minha mãe, Vera Lúcia Martins Manhães, por ter criado e cultivado todas as condições para que eu chegasse até aqui. E por tantas outras coisas que jamais poderei agradecer suficientemente.

RESUMO

TELES, Vinícius Manhães, **Um Estudo de Caso da Adoção das Práticas e Valores do Extreme Programming**. Orientador: Carlo Emmanoel Tolla de Oliveira. Rio de Janeiro : UFRJ/IM, 2005. Dissertação (Mestrado em Informática).

Estudos demonstram que a maioria dos projetos de software falha, seja porque não cumprem o orçamento, ou não cumprem o cronograma, ou as funcionalidades não atendem às necessidades dos usuários ou porque todos estes fatores estão presentes em conjunto. Este trabalho propõe que a adoção do conjunto formado pelas práticas e valores do Extreme Programming possa fornecer um mecanismo eficaz para elevar as chances de sucesso em diversos projetos de software. Como forma de validação, se conduziu uma detalhada revisão da literatura e um estudo de caso é apresentado, no qual o Extreme Programming foi utilizado durante o período de um ano em um projeto comercial com resultados positivos.

ABSTRACT

TELES, Vinícius Manhães, **Um Estudo de Caso da Adoção das Práticas e Valores do Extreme Programming**. Orientador: Carlo Emmanoel Tolla de Oliveira. Rio de Janeiro : UFRJ/IM, 2005. Dissertação (Mestrado em Informática).

Previous works show that the majority of software projects fail due to over budget, delays, features that don't address stakeholders' needs or all of these issues together. This work proposes the adoption of Extreme Programming's set of practices and values as a way to improve the chances of success in several software projects. In order to validate this proposal, a detailed research has been conducted and a study case is presented in which Extreme Programming has been used for one year in a commercial project with positive results.

SUMÁRIO

p.

1 INTRODUÇÃO

Desenvolver software é uma atividade arriscada. Segundo as estatísticas, os maiores riscos são:

- Gastos que superam o orçamento;
- Consumo de tempo que supera o cronograma;
- Funcionalidades que não resolvem os problemas dos usuários e
- Baixa qualidade dos sistemas desenvolvidos.

Há algumas décadas a indústria de software vem buscando técnicas de desenvolvimento que possam reduzir os riscos dos projetos de software e tornar essa atividade mais produtiva. Um marco neste sentido é a criação da disciplina de Engenharia de Software em 1968. De lá para cá, inúmeras propostas foram feitas para melhorar o desempenho dos projetos, começando pelo processo de desenvolvimento linear e seqüencial (em cascata) até chegar aos atuais processos ágeis de desenvolvimento.

O Extreme Programming (XP) é um dos representantes destes processos e foi criado por Kent Beck em 1997 em um projeto para a Chrysler (fabricante de veículos norte-americana). O XP é composto por um conjunto reduzido de práticas de desenvolvimento que se organizam em torno de quatro valores básicos. Essas práticas possuem fortes inter-relacionamentos formando um conjunto de elevada sinergia.

Esta dissertação se propõe a mostrar que este conjunto de práticas pode representar uma forma eficaz de melhorar o desempenho de diversos projetos de software. Em particular, visa demonstrar que embora as práticas sejam úteis isoladamente, a característica mais importante do XP é o conjunto. Assim, se optou por

fazer um trabalho mais amplo, envolvendo todas as práticas e não apenas uma análise isolada de algumas delas.

Para isso, fez-se uma detalhada revisão da literatura, buscando justificar cada uma das práticas sugeridas pelo XP e enfatizando o forte relacionamento de interdependência entre elas. Além disso, é apresentado um estudo de caso de um projeto que utilizou todas estas práticas por mais de um ano.

O Extreme Programming não nasceu no meio acadêmico e, embora existam conferências dedicadas ao assunto há alguns anos, o número de publicações ainda é reduzido quando comparado ao de outras áreas da Informática. Na indústria, os resultados de sua adoção têm sido animadores, mas a comunidade científica tem demonstrado um posicionamento cético visto que diversas práticas propostas contrariam conceitos amplamente difundidos e utilizados tanto nas universidades, quanto na indústria.

Dada a controvérsia existente em torno do XP, optou-se por realizar uma revisão de literatura criteriosa e tão embasada quanto possível em autores e trabalhos de prestígio. Por esta razão, os capítulos se utilizam intencionalmente de um elevado número de citações com o objetivo de apresentar bases teóricas relevantes para a adoção dos valores e práticas propostos pelo Extreme Programming.

O segundo capítulo apresentará uma análise dos problemas recorrentes que estão presentes na maioria dos projetos de software há algumas décadas. Em seguida, o terceiro capítulo fará uma análise da natureza do desenvolvimento de software com o objetivo de compreender quais são as maiores dificuldades que afetam os desenvolvedores de sistemas. O quarto capítulo analisará a forma como as soluções para os projetos de software vêm sendo organizadas ao longo do tempo. Elas se baseiam em

metáforas que precisam ser compreendidas e analisadas para estabelecer a validade de utilizá-las. O quinto capítulo apresentará os valores e práticas do Extreme Programming. O sexto capítulo apresentará o estudo de caso e finalmente o sétimo capítulo fará a conclusão da dissertação.

2 A CRISE DO SOFTWARE

O termo “crise do software” vem sendo usado na indústria de software desde 1968, quando houve a primeira admissão aberta da existência de uma crise latente na área (DIJKSTRA, 1972, tradução nossa). Naquele ano, ocorreu a Conferência da OTAN sobre Engenharia de Software (*NATO Software Engineering Conference*) em Garmisch, Alemanha, que é considerado atualmente o momento histórico do nascimento da disciplina de Engenharia de Software (BRYANT, 2000; EISCHEN, 2002).

Diversos autores utilizam o termo “crise do software”, embora alguns o façam com alguma ressalva, como é o caso de Pressman (1997, p.16, tradução nossa) que considera que “temos uma ‘crise’ que vem nos acompanhando há 30 anos e essa é uma contradição de termos. (...) O que nós temos de fato é uma calamidade crônica.”

O Standish Group, uma empresa localizada em Massachusetts, EUA, publica desde 1994 um estudo chamado de *CHAOS Report*. Trata-se de um amplo levantamento envolvendo milhares de projetos na área de tecnologia da informação. Atualmente, seus dados estão entre os mais utilizados para quantificar a “crise do software”.

O *CHAOS Report* do ano 2000 (THE STANDISH GROUP INTERNATIONAL, 2001) apresenta uma coletânea de dados envolvendo 280 mil projetos de software nos Estados Unidos, os quais revelaram que:

- Em média, os atrasos representaram 63% mais tempo do que o estimado;
- Os projetos que não cumpriram o orçamento custaram em média 45% mais e
- No geral, apenas 67% das funcionalidades prometidas foram efetivamente entregues.

O Standish Group classifica o resultado final de um projeto nas seguintes categorias:

- Mal sucedido;
- Comprometido e
- Bem sucedido.

Em 2000, o resultado final da pesquisa mostrou a seguinte distribuição entre os projetos:

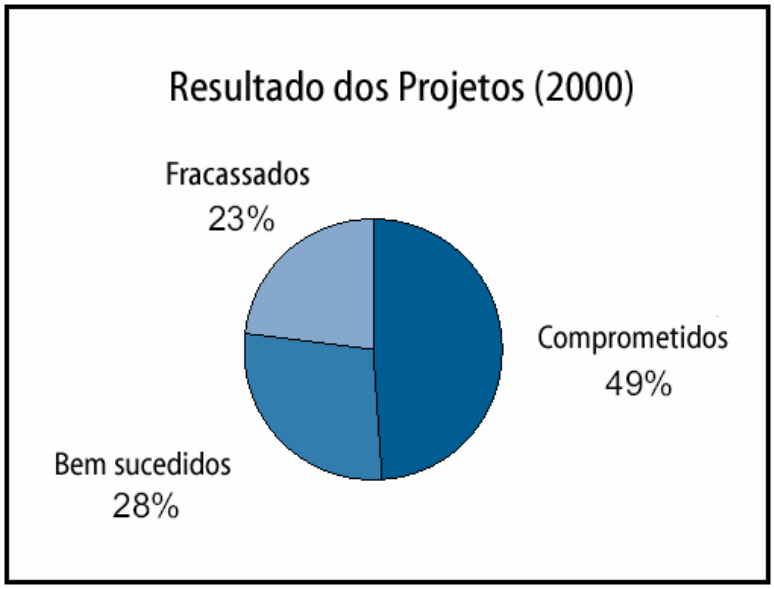

Figura 2.1: estatística sobre o resultado final de projetos de software.

Tais números, embora desastrosos, mostram um avanço em relação aos resultados do primeiro levantamento realizado em 1994:

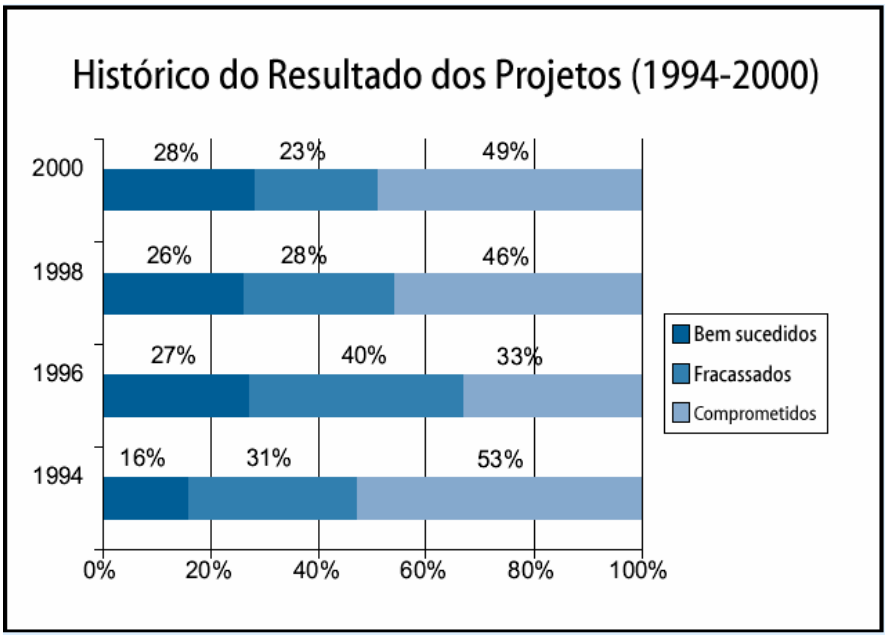

Figura 2.2: evolução do resultado final de projetos de software ao longo dos anos.

Na terceira Conferência Internacional sobre Extreme Programming, que ocorreu na Itália em 2002, Jim Johnson, presidente do Standish Group, apresentou um estudo revelador sobre a utilização das funcionalidades nos projetos que foram pesquisados pelo Standish Group (JOHNSON, 2002). Os resultados demonstram que 45 por cento das funcionalidades encontradas em um sistema típico jamais são usadas e 19 por cento raramente são usadas:

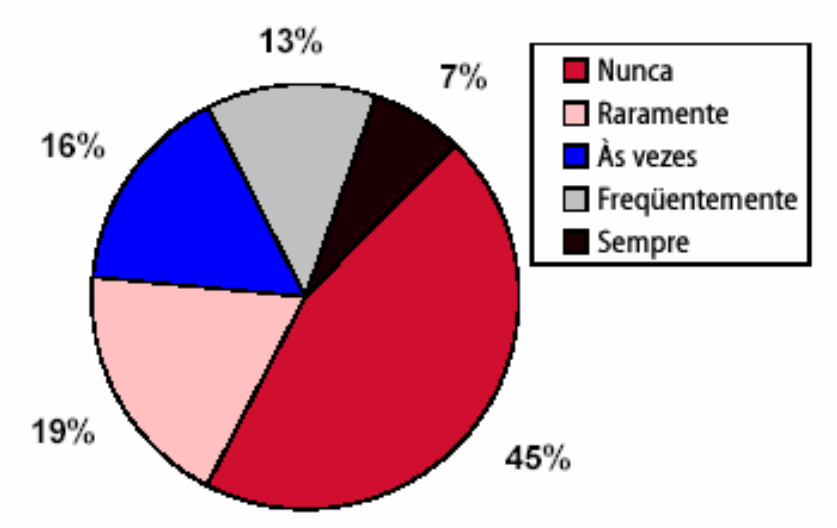

Figura 2.3: estatística sobre a utilização de funcionalidades.

Além de as estatísticas mostrarem, em números, o significado do que vem sendo chamado há quase quarenta anos de "crise do software", a análise de casos isolados demonstra a existência de grandes variações nos resultados dos projetos. É o caso, por exemplo, de dois sistemas desenvolvidos na área governamental dos EUA.

Há alguns anos, os estados americanos da Flórida e Minnesota se lançaram no desafio de criar um Sistema de Informações para o Bem Estar das Crianças (*Statewide Automated Child Welfare Information System* – SACWIS). Cada estado adotou uma abordagem distinta e a diferença de resultados é significativa. Na Florida, o desenvolvimento do sistema teve início em 1990 e foi estimado em oito anos a um custo de US$ 32 milhões. Em 2002, a Florida já havia gasto US$ 170 milhões e o sistema foi re-estimado para ser finalizado em 2005 a um custo de US$ 230 milhões. Por sua vez, Minnesota começou a desenvolver essencialmente o mesmo sistema em 1999 e o finalizou no início de 2000 ao custo de US$ 1,1 milhão. A diferença de produtividade é de 200:1(POPPENDIECK & POPPENDIECK, 2003).

Ao analisar os fatores que levam tantos projetos de software a fracassarem e outros (poucos) a serem bem sucedidos, é relevante avaliar o que significa desenvolver um software. Quais são os fatores que caracterizam o desenvolvimento de software e diferenciam os projetos desta área?

3 A NATUREZA DO SOFTWARE

A compreensão dos fatores que levam à crise do software passa primeiramente pela compreensão da natureza do software e como ele vem sendo tratado ao longo dos anos. Essa análise envolve duas partes: os aspectos básicos que caracterizam o software e as metáforas que são utilizadas no desenvolvimento do mesmo.

O estudo detalhado do software aponta para as seguintes características (BROOKS, 1995; BRYANT, 2000; BUHRER, 2000; KRUTCHTEN, 2001):

- Complexidade;
- Conformidade;
- Maleabilidade;
- Invisibilidade;
- Ausência de leis básicas;
- Imaturidade e
- Baixo custo de manufatura.

3.1 Complexidade

Frederick Brooks, apresenta no artigo "*No silver bullet: essences and accidents of Software Engineering*" (1987) o que considera como sendo as propriedades essenciais do software e começa tratando do problema da complexidade. Ele considera que sistemas de software normalmente possuem uma quantidade elevada de elementos **distintos**, o que os torna mais complexos que qualquer outro tipo de construção humana.

Quando as partes de um software são semelhantes, as mesmas costumam ser agrupadas em métodos, classes e outros elementos. Assim, à medida que um sistema

cresce em tamanho, cresce também a quantidade de partes distintas. Esse comportamento difere profundamente de outras construções, tais como computadores, prédios ou automóveis, nos quais se encontram elementos repetidos em abundância.

Isso leva os programas a terem um número extremamente elevado de estados. Computadores, por exemplo, são produtos bastante complexos que contém uma elevada quantidade de estados. Softwares, por sua vez, costumam ter uma quantidade bem maior de estados.

Outro problema está ligado à necessidade de escalar. Fazer um software escalar não significa apenas repetir os mesmos elementos em tamanho maior. Normalmente é necessário o aumento no número de elementos **distintos**, elevando ainda mais a quantidade de estados de um sistema, e o que é pior, de forma não linear.

Brooks acredita que grande parte dos problemas clássicos relacionados ao desenvolvimento de sistemas deriva diretamente da complexidade que está na essência de qualquer software e sua correspondente elevação não linear com o aumento de tamanho. A complexidade dificulta a comunicação entre os membros da equipe de desenvolvimento e torna difícil enumerar e compreender todos os possíveis estados do programa, resultando em falta de confiabilidade. Por sua vez, a complexidade das funções gera dificuldades para invocá-las, tornando os sistemas difíceis de serem utilizados.

A complexidade estrutural também torna difícil estender os programas para que incorporem novas funcionalidades sem criar efeitos colaterais. Além disso, é difícil ter uma visão geral do software, o que impede que se alcance integridade conceitual no mesmo. Finalmente, o esforço de aprendizado e transmissão de conhecimento se torna

bastante elevado, razão pela qual trocas de pessoal costumam acarretar prejuízos significativos.

A complexidade do software colabora e explica, em parte, os resultados apresentados no capítulo anterior. E é importante compreender que, segundo Brooks, trata-se de um problema que está na essência do software. Isso significa que não existem ferramentas ou técnicas que possam evitá-lo. Elas naturalmente podem ajudar a tratar a complexidade, mas a alta complexidade sempre estará presente nos projetos de software. Segundo Weinberg (1971, p.15, tradução nossa), "(...) programar não é apenas um comportamento humano; é comportamento humano *complexo*."

3.2 Conformidade

A complexidade é um problema que também afeta outras áreas de conhecimento, como por exemplo a física. Esta, além de lidar com objetos extremamente complexos, eventualmente chega ao ponto de lidar com elementos no nível "fundamental" das partículas. Apesar disso, o físico se baseia na convicção de que existam princípios unificadores, os quais, uma vez descobertos, facilitam a compreensão e o tratamento dos problemas. Além disso, princípios físicos tendem a ser estáveis.

Infelizmente, sistemas de software não costumam existir em conformidade com princípios fundamentais e estáveis. Grande parte da complexidade com a qual o desenvolvedor deve lidar é arbitrária. Ela é imposta sobre ele por instituições e sistemas humanos, com os quais o software precisa estar em conformidade. Tal conformidade é dinâmica, visto que os sistemas humanos mudam com o tempo, as pessoas mudam e o software passa a ter que se adequar a novas realidades (BROOKS, 1987).

3.3 Maleabilidade

O software, por ser digital, possui uma maleabilidade extremamente elevada e infinitamente superior àquela encontrada em outros tipos de produtos, como aqueles compostos por elementos físicos. Isso gera pressões permanentes por mudanças nos sistemas.

Fazendo um comparativo, observa-se que construções tais como prédios, carros e computadores também sofrem pressões por mudanças. Entretanto, por serem formadas de elementos físicos, os custos das mudanças são melhores compreendidos e observados, o que reduz os caprichos daqueles que as desejam. "Software, por sua vez, é apenas pensamento, o que o torna infinitamente maleável. Isso é notado pelas pessoas de um modo geral, que naturalmente pressionam por mais mudanças, por considerarem que as mesmas terão um custo reduzido (BROOKS, 1987)."

A maleabilidade do software difere, portanto, daquela encontrada na engenharia civil. "Se você constrói uma ponte, você não tem este tipo de flexibilidade. Você não pode dizer 'Hum, agora que eu já vejo os pilares, eu gostaria que essa ponte fosse colocada duas milhas rio acima' (KRUTCHTEN, 2001, tradução nossa)."

Na situação ilustrada acima, qualquer pessoa seria capaz de avaliar o enorme custo de mover a ponte de uma posição para a outra, o que tenderia a reduzir ou eliminar completamente a mudança. Mas, no caso do software, a sua maleabilidade torna as mudanças muito mais fáceis e menos custosas. Seus usuários têm a exata percepção de que é infinitamente mais fácil alterar um software que a posição de uma ponte, o que os leva a solicitar mudanças com mais freqüência e mais intensidade.

3.4 Invisibilidade

Ao elaborar um projeto, diversos profissionais têm a oportunidade de utilizar uma importante ferramenta: abstrações geométricas. Um arquiteto, por exemplo, tem a possibilidade de utilizar uma planta baixa que o ajuda, bem como ajuda seu cliente a avaliar espaços, fluxos de trânsito, disposição de elementos, entre outros. Com ela, torna-se simples identificar contradições e omissões. Ao capturar a realidade geométrica, em uma abstração geométrica correspondente, o arquiteto tem a sua disposição uma ferramenta que facilita e melhora o seu trabalho.

Já o software, é invisível e impossível de ser visualizado, visto que sua realidade não se encontra inserida de modo intrínseco no espaço. Assim, não possui uma representação geométrica à disposição, da mesma forma que terras possuem mapas, por exemplo. Ao se tentar diagramar a estrutura de um software, notamos que ela "é constituída não por um, mas por vários grafos direcionados de forma genérica, superpostos uns sobre os outros. Os vários grafos podem representar o fluxo de controle, o fluxo de dados, padrões de dependência (...) (BROOKS, 1987, tradução nossa)."

Brooks acredita que, embora tenha havido avanços no sentido de simplificar as estruturas de software, elas continuam sendo praticamente impossíveis de serem visualizadas. Isso gera uma deficiência para o desenvolvedor que acaba não podendo contar com uma importante ferramenta. "Esta falta não apenas retarda o processo de design dentro de uma mente, como também prejudica severamente a comunicação entre mentes diferentes (BROOKS, 1987, tradução nossa)."

3.5 Inexistência de princípios básicos

Como já vimos, o software não possui leis fundamentais como a física, tornando difícil pensar sobre ele sem efetivamente construí-lo. Além disso, "os poucos padrões de engenharia de software que conhecemos se baseiam apenas em boas práticas, enquanto códigos de construção em outras disciplinas se originam em sólidos princípios físicos (KRUTCHTEN, 2001, tradução nossa)."

3.6 Rápida evolução tecnológica

As técnicas de desenvolvimento de software, ferramentas e o próprio ambiente de software mudam em um ritmo profundamente acelerado. Isso torna difícil consolidar uma base de conhecimento e pressiona os profissionais a se treinarem e re-treinarem permanentemente. Portanto, os profissionais parecem conviver com um eterno re-começo, pois aquilo que aprenderam há pouco tempo, perde a utilidade com enorme rapidez. Além disso, "a engenharia de software, ao contrário de outras disciplinas, não se beneficia de centenas ou até mesmo milhares de anos de experiência (KRUTCHTEN, 2001, tradução nossa)."

3.7 Baixo custo de manufatura

Quando os desenvolvedores de software tratam do design de um aplicativo, normalmente estão se referindo a uma descrição de alto nível das estruturas do sistema. Freqüentemente acreditam que elaborar este design é a parte complexa, enquanto a codificação é uma parte mecânica. Comparando com uma fábrica de automóveis, é como se elaborar o design se assemelhasse ao trabalho do engenheiro que projeta um novo modelo, enquanto a codificação é o trabalho mecânico de produzir tal modelo no chão de fábrica.

Segundo Krutchten (2001), isso não corresponde à realidade. Praticamente todo o trabalho do desenvolvedor, incluindo a codificação, representa um esforço de design. Usando a comparação anterior, praticamente todo o trabalho de desenvolvimento de software pode ser comparado ao trabalho do engenheiro que projeta um novo automóvel.

A manufatura de um automóvel, ou seja, o trabalho realizado no chão de fábrica para reproduzir o mesmo modelo inúmeras vezes, buscando eliminar variações, corresponde em software ao trabalho de compilar, empacotar e gerar um CD, por exemplo. Este é o trabalho que pode, é automatizado, e é essencialmente o trabalho que se assemelha ao que é feito na linha de produção de uma indústria, com uma importante diferença: o custo.

Manufaturar um software tem um custo extremamente baixo. Comparando-se com uma indústria automobilística, por exemplo, é quase como se não existisse custo de manufatura. Quase todo o investimento está associado ao design. Uma vez projetado, o software pode ser replicado infinitas vezes, fazendo-se uma cópia de arquivos, gerando-se um CD ou transferindo-se o aplicativo pela Internet (KRUTCHTEN, 2001).

Projetar um novo modelo de automóvel é um trabalho essencialmente difícil, pouco previsível, demorado e com potencial limitado de ser acelerado. São necessárias várias idas e vindas durante o projeto (POPPENDIECK & POPPENDIECK, 2003). A automação não exerce um efeito tão drástico na velocidade de elaboração do projeto, quanto exerce na manufatura do automóvel.

A partir das características que analisamos, “podemos compreender que sempre haverá uma ‘crise do software’, pois a causa de muitos dos problemas está na própria natureza do software (BRYANT, 2000, tradução nossa).” O que certamente podemos

fazer é reconhecer estas características e buscar soluções que as levem em consideração, de modo a atenuar os problemas que as equipes de desenvolvimento costumam vivenciar.

4 METÁFORAS E O DESENVOLVIMENTO DE SOFTWARE

Há décadas, a "crise do software" vem inspirando estudiosos e profissionais a criarem propostas para solucioná-la. De um modo geral, tais soluções carregam metáforas que procuram lançar luz sobre a natureza do software e possíveis abordagens para tratá-la.

Como mostra Bryant (2000), a prática de desenvolvimento de software é "inevitavelmente fundada sobre metáforas. A própria palavra 'software' é uma metáfora." A compreensão das metáforas, que vêm sendo usadas historicamente, pode ser útil para o entendimento de diversos fatores que colaboram para a "crise do software", ao mesmo tempo em que pode nos ajudar a visualizar novas soluções, através do uso de novas metáforas. "A tensão entre sua invisibilidade e intangibilidade por um lado e sua complexidade por outro, praticamente exige mecanismos e alusões metafóricas (BRYANT, 2000, tradução nossa)."

Ao longo dos anos, a metáfora que mais vem influenciando o processo de desenvolvimento de software é a da engenharia. Software vem sendo visto como um processo de construção e de fabricação. Em tal metáfora, utilizam-se termos tais como construção, desenvolvimento, manutenção, prototipagem, entre outros (BRYANT, 2000).

Um dos efeitos mais marcantes da "crise do software", é a busca permanente por soluções que possam tornar os projetos:

- Mais produtivos;
- Mais previsíveis e
- Com resultados de melhor qualidade.

Felizmente, outras disciplinas se depararam com esta mesma busca no passado. Algumas foram extremamente bem sucedidas, tendo alcançado os três objetivos descritos acima. Em particular, vale a pena analisar a indústria de produção em massa, que vem melhorando continuamente nestes quesitos há alguns séculos (DRUCKER, 1999; TOFFLER, 2001).

4.1 A indústria de produção em massa

Alvin Toffler (2001) avalia a história da humanidade como uma sucessão de ondas de mudança em marcha, as quais focalizam a nossa atenção não apenas para as continuidades históricas (por mais importantes que sejam), mas também as descontinuidades, ou seja, as inovações e interrupções.

> Começando com a simplíssima idéia de que o aparecimento da agricultura foi o primeiro ponto decisivo do desenvolvimento social humano, e de que a revolução industrial foi a segunda grande ruptura, olha cada um destes acontecimentos não como um discreto evento no tempo, mas como uma onda de mudança avançando a uma certa velocidade (TOFFLER, 2001, p.27).

A Revolução Industrial, que nos lançou na Segunda Onda de mudanças, trouxe consigo uma série de regras, consistindo em "seis princípios inter-relacionados (...). Nascendo naturalmente da desunião da produção e do consumo, estes princípios afetavam todos os aspectos da vida (...) (TOFFLER, 2001, p.59)."

Os seis princípios básicos apontados por Toffler (2001) são:

- Padronização;
- Especialização;
- Sincronização;
- Concentração;
- Maximização e

- Centralização.

4.1.1 Padronização

A padronização foi um princípio inventado por um construtor de móveis chamado Thonet que "(...) descobriu que, em vez de fabricar cem cadeiras, cada uma diferente da outra, é muito mais lucrativo fazê-las todas iguais: o desperdício é menor, a produção é mais rápida e a menor custo (MASI, 2000, p.59)."

Pelos seus méritos, tal princípio foi levado às últimas conseqüências por Frederick Winslow Taylor no início do século XX. Ele propôs que o trabalho podia ser científico, caso fosse possível padronizar os passos executados pelos trabalhadores para executarem suas atividades. Sobretudo, Taylor acreditava que "havia uma maneira melhor (padrão) de realizar cada tarefa, uma ferramenta melhor (padrão) para executá-la com ela e um tempo estipulado (padrão) no qual podia ser completada (TOFFLER, 2001, p.61)."

4.1.2 Especialização

Em 1776, no auge da Revolução Industrial, Adam Smith publicou um dos mais famosos e importantes livros de economia: A riqueza das nações. No início da obra, ele aborda o princípio da especialização, declarando que "o maior aprimoramento das forças produtivas do trabalho, e a maior parte da habilidade, destreza e bom senso com os quais o trabalho é em toda parte dirigido ou executado, parecem ter sido resultados da divisão do trabalho (SMITH, 1996, p.65)."

A **divisão do trabalho** é uma das características mais importantes do processo de industrialização devido ao enorme aumento de produtividade que acabou

proporcionando. Como exemplo, Toffler faz referência à Smith, que ao visitar uma fábrica de alfinetes ofereceu o seguinte relato:

> Um trabalhador único de velho estilo, efetuando todas as operações necessárias sozinho, escreveu, podia produzir apenas um punhado de alfinetes por dia – não mais de 20 e talvez nem um. Em contraste, Smith descrevia uma "manufatura" que ele tinha visitado, na qual se exigiam 18 operações diferentes efetuadas por dez trabalhadores especializados, cada um efetuando apenas uma ou algumas fases. Juntos, conseguiam produzir mais de 48 mil alfinetes por dia – mais de quatro mil e oitocentos por trabalhador (TOFFLER, 2001, p.62).

4.1.3 Sincronização

O princípio da sincronização está associado à necessidade de reunir os trabalhadores em um local no qual se encontram os meios de produção, ou seja, na fábrica. Isso causa a necessidade de que as pessoas estejam juntas ao mesmo tempo e, portanto, tenham os mesmos horários.

> Se fôssemos artesãos numa oficina de vasos, cada um fabricaria um vaso inteiro. Se, ao contrário, trabalhássemos numa linha de montagem, você enroscaria um parafuso e, cinco segundos depois, eu deveria apertar outro: logo, deveríamos ambos estar presentes no instante em que a cadeia se inicia (MASI, 2000, p.61).

Além da interdependência intensificada, "máquinas caras não podem ficar ociosas e operam ao seu ritmo próprio (TOFFLER, 2001, p.64)." Por essa razão, a pontualidade se torna essencial e toda a fábrica precisa operar em sincronia, de acordo com o ritmo estabelecido pelas máquinas.

4.1.4 Concentração

A concentração (ou economia de escala) se baseia na idéia de que "(...) se eu concentro dez empresas de mil pessoas numa única megaempresa [sic] de dez mil pessoas, será necessário um número menor de dirigentes, de empregados, de fiscais e o lucro será maior (MASI, 2000, p.66)." Naturalmente, as fontes de economia podem

englobar também outros fatores, tais como a reutilização de equipamentos, o maior poder de barganha com fornecedores e maior capacidade de impor preços vantajosos no mercado.

4.1.5 Maximização

A maximização está presente na tentativa de maximizar o lucro das empresas, bem como o tamanho e a taxa de crescimento das mesmas. Neste sentido, "Taylor concebe a fórmula E = P/H, que quer dizer que a eficiência (E) é igual a P, de produção, dividido por H, horas de trabalho (MASI, 2000, p.65)."

A busca por maior produtividade gerou efeitos importantes no mundo inteiro, a um tal ponto que autores como Peter Drucker consideram que a acentuada elevação da produtividade ao longo do século XX foi a responsável pela atual existência de países desenvolvidos.

> Menos de uma década depois que Taylor examinou o trabalho e analisou-o, a produtividade do trabalhador manual iniciou sua ascensão sem precedentes. Desde então, ela tem subido regularmente à taxa de 3,5% ao ano, o que significa que aumentou 50 vezes desde Taylor. Nesta realização baseiam-se todos os ganhos econômicos e sociais do século XX. A produtividade do trabalhador manual criou aquelas que hoje chamamos de economias "desenvolvidas". Antes de Taylor, isso não havia – todas as economias eram igualmente "subdesenvolvidas". Hoje, uma economia subdesenvolvida, ou mesmo "emergente", é aquela que ainda não tornou produtivo o trabalhador manual (DRUCKER, 1999, p.112).

Analisando a citação de Drucker, é fundamental notar o uso da expressão "trabalhador manual" e, sobretudo o fato de que o aumento de produtividade ao qual ele se refere diz respeito apenas ao "trabalhador manual". Voltaremos a este assunto nas próximas seções, quando começaremos a analisar a produtividade do trabalho de desenvolvimento de software.

4.1.6 Centralização

Da mesma forma que a Segunda Onda trouxe consigo a forte divisão entre produção e consumo (TOFFLER, 2001), criou também a divisão entre planejamento e execução, isto é, deixou claro que existem aqueles que pensam (e conseqüentemente mandam) e aqueles que fazem (e, portanto, obedecem). Utiliza-se a premissa de que "(...) a organização deve ter a forma de uma pirâmide: o vértice sabe tudo e pode tudo. Entre quem pensa e quem executa, a divisão é cristalina (MASI, 2000, p.66)."

4.2 O desenvolvimento de software como trabalho do conhecimento

No início deste capítulo, apontamos a busca por soluções que possam tornar os projetos de desenvolvimento de software mais produtivos, mais previsíveis e com resultados de melhor qualidade. Esses objetivos, entre outros, foram alcançados com sucesso pela indústria de produção em massa utilizando os princípios explicados nas seções anteriores, que estão presentes no cotidiano. "Desde 1776, quando Adam Smith defendeu a divisão do trabalho em 'A riqueza das nações', racionalizar a produção vem servindo como um método provado para elevar a qualidade, reduzir custos e aumentar a eficiência (EISCHEN, 2002).

Uma questão relevante que se poderia levantar é se não seria possível utilizar estes mesmos princípios no desenvolvimento de software e obter os mesmos resultados positivos. Esse questionamento não é novo. Ele acompanha a indústria de software há bastante tempo. Muitos acreditam que é possível mapear estes princípios no desenvolvimento de software e cada vez mais encontramos metáforas que procuram aproximá-lo do processo de produção de uma fábrica, culminado inclusive na atual idéia de "fábrica de software" tão extensamente divulgada no mercado.

Em janeiro de 2005, uma busca pela expressão "fábrica de software" no Google[1] gerou como resultado 333 mil referências, dentre as quais era possível identificar uma grande quantidade de empresas brasileiras que atuam no desenvolvimento de software, desde pequenas a grandes. Utilizando-se o equivalente em inglês, ou seja, a expressão *software factory*, o Google retornou 10.2 milhões de referências. Tais números dão uma idéia da extensão desta metáfora e da importância que lhe é dedicada atualmente.

Seu uso é facilmente compreensível quando observamos os resultados da produção industrial e o tremendo impacto gerado pelos princípios descritos anteriormente sobre o "trabalho manual" (DRUCKER, 1999). A expressão "trabalho manual" nos chama a atenção para a possibilidade de existência de outros tipos de trabalho, os quais podem ou não ser beneficiados pelos princípios da industrialização em massa, ou Segunda Onda, para usar as palavras de Toffler (2001).

Outro tipo de trabalho que está cada vez mais presente na sociedade contemporânea é o **trabalho do conhecimento**. Compreender a diferença entre trabalho manual e trabalho do conhecimento é relevante tendo em vista o fato de que "(...) grande parte da discussão atual (...) parece assumir que programar é similar à produção industrial, o programador sendo visto como (...) um componente que precisa ser controlado (...) e que pode ser substituído facilmente (COCKBURN, 2002, p.237, tradução nossa)."

Diversos autores defendem a teoria de que existem pelo menos dois grupos de trabalhadores bastante distintos: os "trabalhadores manuais" e os "trabalhadores do conhecimento", utilizando-se a nomenclatura adotada por Drucker (1999). Entre estes autores destacam-se Cockburn (2002), DeMarco (2001), DeMarco e Lister (1999;

---

[1] Ferramenta de busca na Internet.

1987), De Masi (2000), Drucker (1999), Poppendieck e Poppendieck (2003) e Toffler (2001).

Os trabalhadores do conhecimento existem há bastante tempo, mas ao longo da Segunda Onda, isto é, da Era Industrial, o número de trabalhadores manuais era maior e mais significativo do ponto de vista de resultados para a sociedade. Entretanto, à medida em que avançamos pela Terceira Onda, conforme a nomenclatura de Toffler (2001), ou a Sociedade Pós-industrial, segundo De Masi (2000) ou a Revolução da Informação, segundo Drucker (1999), "os trabalhadores do conhecimento estão se tornando rapidamente o maior grupo isolado da força de trabalho de todos os países desenvolvidos (DRUCKER, 1999, p.116)."

Os desenvolvedores de software encontram-se entre os "trabalhadores do conhecimento" (COCKBURN, 2002; DEMARCO & LISTER, 1987; POPPENDIECK & POPPENDIECK, 2003). Portanto, é fundamental avaliar os fatores que podem ser utilizados para elevar a produtividade, a previsibilidade e a qualidade do "trabalhador do conhecimento", os quais, não são, necessariamente, os mesmos que regem o "trabalho manual". De fato, como veremos adiante, tais fatores, embora ainda não sejam tão bem conhecidos, parecem se distanciar profundamente daqueles tradicionalmente usados para os "trabalhadores manuais".

Segundo Drucker (1999), existem seis fatores essenciais que determinam a produtividade do trabalhador do conhecimento. São eles:

1. Definir qual é a tarefa a ser feita;
2. Permitir que os próprios trabalhadores se auto-gerenciem. Ou seja, assegurar que eles tenham autonomia e responsabilidade sobre o que produzem;

3. Assegurar que os trabalhadores tenham a oportunidade de inovar;
4. Aprendizado e ensino contínuo;
5. Qualidade é um fator tão o mais importante que a quantidade produzida e
6. Os trabalhadores do conhecimento precisam ser tratados como "ativos" e não como "custo". Além disso, precisam querer trabalhar para a organização.

Portanto os princípios apresentados anteriormente, sobre a produtividade do "trabalhador manual", não se aplicam ao "trabalho do conhecimento", podendo inclusive prejudicá-lo.

> O que os empregadores da Terceira Onda precisam cada vez mais, por conseguinte, são homens e mulheres que aceitem responsabilidade, que compreendam como o seu trabalho combina com o dos outros, que possam manejar tarefas cada vez maiores, que se adaptem rapidamente a circunstâncias modificadas e que estejam sensivelmente afinados com as pessoas em volta deles. (...)
> A firma da Terceira Onda exige pessoas que sejam menos pré-programadas e mais criativas. (...)
> Tais pessoas são complexas, individualistas, orgulhosas das maneiras como diferem umas das outras. (...)
> Elas procuram significado juntamente com recompensa financeira (TOFFLER, 2001, p.378).

De Masi (2000) cita a Wiener Werkstätte como exemplo de uma organização que sabia alcançar elevada produtividade para trabalhadores do conhecimento utilizando premissas praticamente opostas àquelas propostas por Taylor. Tratava-se de uma cooperativa em Viena onde se produzia, por exemplo, cartões postais, papel de parede, talheres, móveis e até mesmo bairros inteiros. Nela, o processo de criação e produção era completamente diferente daqueles de Taylor: "escassa divisão do trabalho, pouca padronização, pouca especialização, pouca sincronização, pouca centralização, pouca maximização. Com resultados (...) extraordinários (MASI, 2000, p.69). "

4.2.1 Definir a tarefa

Os trabalhadores manuais são programados pela tarefa que executam e normalmente executam um conjunto reduzido de tarefas repetidas vezes. As tarefas realizadas por um trabalhador do conhecimento, entretanto, costumam ser maiores, mais complexas e pouco estruturadas. Além disso, no dia-a-dia, um trabalhador do conhecimento é solicitado a fazer inúmeras tarefas distintas. "Os engenheiros estão constantemente sendo afastados de sua tarefa por terem de redigir um relatório ou reescrevê-lo, serem solicitados para uma reunião etc (DRUCKER, 1999, p.118)."

Drucker (1999) considera que, diante desta realidade, o aspecto mais importante da produtividade do trabalhador do conhecimento é a capacidade de **priorizar**. Trabalhadores do conhecimento, incluindo os desenvolvedores de software, se envolvem em uma infinidade de atividades diferentes, as quais evoluem e mudam dinamicamente ao longo do tempo. Para obter a máxima produtividade, é necessário que eles saibam priorizar e concentrar o máximo de esforços naquilo que mais pode fazer diferença para a organização ou seu projeto a cada dia de trabalho.

4.2.2 Qualidade

A produtividade do trabalho manual está fortemente atrelada ao volume produzido, desde que se respeitem os padrões mínimos de qualidade. O mesmo não acontece com o trabalho do conhecimento, pois neste caso, a qualidade é a essência da produção. Por exemplo, "ao julgar o desempenho de um professor, não questionamos quantos alunos pode haver em uma classe, mas quantos deles aprendem algo – e esta é uma pergunta de qualidade (...) (DRUCKER, 1999, p.117)."

O trabalhador do conhecimento deve executar suas atividades de modo a atingir não apenas a qualidade mínima, mas sim a máxima qualidade possível. A questão da quantidade só começa a fazer sentido depois de se alcançar o maior nível de qualidade.

No caso do desenvolvimento de software, por exemplo, onde técnicas, ferramentas e ambientes de software evoluem em ritmo extremamente acelerado, atingir alta qualidade está diretamente associado a um processo de aprimoramento contínuo. Portanto, a produtividade do desenvolvedor de software está profundamente associada a sua capacidade de executar suas atividades com qualidade elevada e continuar aprendendo tanto quanto possível à medida que as executa.

4.2.3 O trabalhador do conhecimento como ativo

Na lógica do trabalho manual, acredita-se comumente que o trabalhador é um custo e, ao mesmo tempo uma peça da engrenagem que constitui os sistemas de negócio de uma organização. "Com exceção do custo de *turnover*[2], o gerenciamento de trabalhadores, baseado em milênios de trabalho quase totalmente manual, ainda assume que (...) um trabalhador manual é igual a outro (DRUCKER, 1999, p.121)."

A perda de um trabalhador manual gera custos de recontratação, re-treinamento etc, bem como a possibilidade de se perder a experiência dessa pessoa. Apesar disso, o trabalhador manual ainda é visto como um custo e basicamente como uma peça intercambiável.

No caso do trabalhador do conhecimento a situação é bem mais grave. O trabalhador manual não possui os meios de produção. Portanto, sua experiência só é valiosa no local em que trabalha, ela não é portável. Por sua vez, os trabalhadores do

[2] Custo de substituição de um trabalhador.

conhecimento **possuem** os meios de produção. “O conhecimento que está entre suas orelhas é um ativo enorme e totalmente portátil. Pelo fato de possuírem seus meios de produção, eles são móveis (DRUCKER, 1999, p.121).”

> Hoje, se sou um publicitário e estou tentando criar um slogan, quando saio do escritório e volto para casa, levo o trabalho comigo: na minha cabeça. A minha cabeça não pára de pensar e às vezes acontece que posso achar a solução para o slogan em plena noite, ou debaixo do chuveiro, ou ainda naquele estado intermediário entre o sono e o despertar (MASI, 2000, p.205).

A perda de um trabalhador do conhecimento tem conseqüências mais sérias, porque significa também a perda do meio de produção e de todo o aprendizado obtido ao longo do tempo. Portanto, o trabalhador do conhecimento precisa ser encarado como um ativo que deve ser mantido na organização. Se os custos de substituição são elevados para o trabalhador manual, eles são significativamente maiores para os trabalhadores do conhecimento.

4.2.4 Motivação

No trabalho do conhecimento (também chamado de trabalho intelectual por De Masi) o trabalho é pouco estruturado e não existe uma linha de produção que imponha o ritmo de trabalho. Portanto, é preciso que cada trabalhador decida produzir com a máxima qualidade no ritmo mais ágil possível. Por esta razão, “no trabalho intelectual a motivação é tudo (MASI, 2000, p.223).”

Segundo Cockburn (2002, p.63, tradução nossa), “existem três tipos de recompensa que podem manter a motivação intrínseca de uma pessoa: orgulho no trabalho, orgulho de realizar e orgulho da contribuição.” Brooks (1995), por sua vez, acredita que o trabalho de desenvolvimento de software proporciona 5 tipos de satisfações:

> - A satisfação de montar coisas;
> - A satisfação de montar coisas que são úteis para outras pessoas;
> - O fascínio de montar objetos que se assemelham a quebra-cabeças;
> - A satisfação de estar sempre aprendendo coisas não repetitivas e
> - O prazer de trabalhar em um meio tão maleável – pensamento puro – que, apesar de maleável, existe, se move e trabalha de uma forma diferente dos objetos do mundo físico (BROOKS, 1995, p.230, tradução nossa).

Se motivação é essencial para elevar a produtividade do trabalhador do conhecimento, precisamos compreender que fatores levam uma pessoa a ficar motivada ou desmotivada. De um modo geral, não é fácil motivar alguém, pois é necessário que a própria pessoa seja capaz de se motivar. Entretanto, é relativamente fácil reduzir ou eliminar a motivação de um trabalhador do conhecimento.

O trabalhador do conhecimento precisa compreender o propósito daquilo que faz. "Você não pode lhe dizer para fazer alguma coisa porque você é o chefe e você diz que precisa ser feito. (...) Você não pode lhe impor objetivos que não lhe façam sentido (DEMARCO, 2001, p.28, tradução nossa)."

Além disso, não se pode estruturar as atividades de um trabalhador do conhecimento. Ele tem que ser o próprio responsável pela forma como o trabalho é conduzido. Alem disso, tem que saber o que deve ser feito e porque.

A essência da Administração Científica de Taylor é ensinar ao trabalhador manual a melhor forma de executar uma tarefa. Ou seja, estruturar a atividade é fundamental. No trabalho do conhecimento ocorre o inverso. Não se pode estruturar a atividade e, sobretudo, não se pode estruturá-la de uma forma que não dê ao trabalhador a chance de crescer. "Crescimento é essencial para ele, tão essencial quanto o contra-

cheque. Você não pode mais esperar que ele trabalhe sem desafios significativos (...) (DEMARCO, 2001, p.28, tradução nossa)."

A preocupação em padronizar a forma de execução das atividades de um trabalhador do conhecimento também se mostra ineficaz porque acabamos nos concentrando na mecânica da atividade que é uma parte pouco significativa em relação ao trabalho como um todo. "A forma como o trabalho é executado dentro dos nós do organograma não é nem de perto tão importante quanto estabelecer quão ampla e rica são as conexões [entre os trabalhadores] (DEMARCO, 2001, p.108, tradução nossa).

Ao lidar com atividades mais complexas, os trabalhadores do conhecimento normalmente necessitam da ajuda de diversos colegas para atingir seus objetivos. Por essa razão, a riqueza da comunicação, do relacionamento e da colaboração entre os trabalhadores do conhecimento é mais relevante que a forma como as atividades são estruturadas. Por isso a preocupação em padronizar o modo de trabalho é pouco eficaz e, eventualmente, prejudicial. Especialmente quando os padrões adotados prejudicam o fluxo de comunicação ou reduzem as chances de se executar um trabalho de alta qualidade do qual se possa ter orgulho.

4.3 A produção enxuta

A mesma indústria automobilística que levou a industrialização em massa às últimas conseqüências através do Taylorismo criou, algumas décadas depois, um processo de produção diferente, que aborda de maneira diferente o trabalho do conhecimento. Desta vez, ao invés da Ford, os conceitos vieram da Toyota, no Japão.

Na década de 1940, a Toyota buscou formas de produzir automóveis com maior agilidade e qualidade, mas com custos mais reduzidos. Além disso, precisava viabilizar um modelo de produção que não fosse em massa, visto que naquele momento não havia

demanda no Japão para absorver uma oferta baseada na produção em massa. Assim a Toyota criou a *produção enxuta* (*lean production*, em inglês) que foi se aperfeiçoando ao longo de décadas e atualmente é mais conhecida pelo termo *just-in-time* (POPPENDIECK & POPPENDIECK, 2003).

Atualmente, "a Toyota é uma das montadoras de automóveis mais lucrativas do mundo. Ela fabrica produtos excelentes, cresce rapidamente, tem altas margens de lucratividade e fatura muito dinheiro (BECK & ANDRES, 2005, p.135, tradução nossa)." Entretanto, utiliza um conjunto de práticas completamente diferente daquelas sugeridas por Taylor. Ela procura eliminar esforços desnecessários em cada passo de fabricação de seus automóveis, na crença de que eliminando desperdícios se consegue avançar mais rapidamente.

A produção enxuta é mencionada nesta obra por conter princípios que estão na base dos processos ágeis de desenvolvimento de software como o Extreme Programming. Ela é caracterizada por um conjunto de sete princípios básicos (POPPENDIECK & POPPENDIECK, 2003):

1. Eliminar desperdícios;
2. Amplificar o aprendizado;
3. Adiar decisões ao máximo;
4. Entregar o mais rapidamente possível;
5. Delegar poder à equipe;
6. Incorporar integridade e
7. Ver o todo.

Estes princípios incorporam os fatores citados anteriormente sobre a produtividade do trabalhador do conhecimento. Por esta razão, existe uma chance de

que processos de desenvolvimento de software baseado nos mesmos possam efetivamente elevar a produtividade e a qualidade dos projetos de desenvolvimento.

4.3.1 Eliminar desperdícios

Ao analisar a produtividade do trabalhador manual, Taylor buscou formas de eliminar desperdícios e, portanto, obter maior produtividade. A forma utilizada por ele foi avaliar o modo de trabalho dos operários e lhes ensinar "a melhor forma" de executar as tarefas. Esse modelo funcionou e ao adotá-lo a Ford elevou tanto a sua produtividade que praticamente levou à falência todas as fábricas artesanais de automóveis que existiam na época (em torno de quinhentas) (POPPENDIECK & POPPENDIECK, 2003).

O Sistema de Produção da Toyota, por sua vez, também priorizou a redução de desperdícios, porém adotou estratégias diferentes. Ela procurou colocar o cliente final no centro do problema e analisar toda a cadeia produtiva desde o momento em que o automóvel começava a ser produzido até o momento de ser entregue ao cliente.

Observando a cadeia, procurou identificar tudo aquilo que era feito e que não gerava resultados perceptíveis para o cliente final. Se alguma coisa assim fosse identificada, seria considerada um desperdício e, portanto, eliminada. A ênfase foi em tentar reduzir, tanto quanto possível, a quantidade de trabalho executada e os subprodutos envolvidos, de modo a concentrar esforços exclusivamente naquilo que pudesse gerar um resultado objetivo e perceptível para o cliente final.

> Desperdício é tudo aquilo que não adiciona valor ao produto, valor tal como percebido pelo cliente. (...) Se um componente está colocado em uma estante pegando poeira, isso é desperdício. Se um ciclo de desenvolvimento coletou requisitos em um livro que está pegando poeira, isso é desperdício. Se uma planta industrial produz mais coisas do que é imediatamente necessário, isso é desperdício. Se os desenvolvedores codificam mais funcionalidades que o imediatamente necessário, isso é desperdício, transferir o desenvolvimento de um

> grupo para outro é desperdício. O ideal é descobrir o que o cliente deseja e então fazer ou desenvolver isso e entregar exatamente o que ele quer, virtualmente de imediato. O que quer que atrapalhe a rápida satisfação da necessidade do cliente é um desperdício (POPPENDIECK & POPPENDIECK, 2003, p.xxv, tradução nossa).

O Sistema de Produção da Toyota também busca eliminar desperdícios fazendo com que o estoque seja mínimo, ou simplesmente inexistente, e as entregas sejam efetuadas com a maior velocidade possível. O objetivo disso é receber feedback rapidamente sobre o produto produzido. Acredita-se que quanto mais curto for o ciclo de feedback, maior será o aprendizado e mais chances existirão para aprimorar o produto e o processo de produção. Além disso, eventuais falhas poderão ser descobertas cedo, facilitando a correção e tornando-as menos custosas. Portanto, toda a produção é organizada em torno da idéia de aprendizado, melhoria contínua e detecção rápida de falhas.

4.3.2 Amplificar o aprendizado

Ao contrário da abordagem adotada por Taylor, a Toyota partiu da premissa de que a melhor forma de se executar um trabalho não é estática, mas sim dinâmica. Sendo assim, busca fazer com que os operários aprendam cada vez mais, se tornem cada vez mais habilidosos e, portanto, capazes de criar formas inovadoras e mais eficazes de realizar suas tarefas à medida que ganhem mais experiência e conhecimento. Além disso, acredita que ninguém tem mais informações para aprimorar o trabalho do chão de fábrica quanto as pessoas que estão lá executando o trabalho.

Ao fazer isso, a Toyota se afasta da idéia da separação entre planejamento e execução que faz parte do modelo Taylorista. Na Toyota, o operário é responsável por planejar, executar e aprimorar continuamente a forma de fazer ambas as coisas. O supervisor deixa de ser responsável pelo planejamento centralizado e assume o papel de

treinador. Ele busca assegurar que a equipe tenha o aprendizado mais rico possível ao longo do trabalho (POPPENDIECK & POPPENDIECK, 2003).

> Desenvolver é como criar uma receita, enquanto produzir é como preparar o prato. Receitas são criadas por *chefs* experientes que desenvolveram o instinto para o que funciona e a capacidade de adaptar os ingredientes disponíveis conforme a ocasião. Ainda assim, até mesmo os grandes *chefs* produzem inúmeras variações de um novo prato à medida que iteram na direção da receita que terá um excelente sabor e será fácil de reproduzir. Não se espera que os *chefs* atinjam a receita perfeita na primeira tentativa; espera-se que eles produzam diversas variações sobre o mesmo tema como parte natural do processo de aprendizagem (POPPENDIECK & POPPENDIECK, 2003, p.xxv, tradução nossa).

Visto que o desenvolvimento de software envolve experimentação e aprimoramento, a idéia de amplificar o conhecimento é bastante relevante. Existe ainda o desafio adicional de que equipes de desenvolvimento costumam ser numerosas e os resultados bem mais complexos do que receitas. Além disso, a rápida evolução tecnológica torna ainda mais essencial a adoção de formas de se amplificar o conhecimento em um projeto de software.

4.3.3 Adiar decisões ao máximo

Ao se projetar um novo produto, tal como um software, existem decisões que podem ser afetadas por mudanças que venham a ocorrer ao longo do projeto. Por exemplo, mudanças ocorridas na economia ou de regras legislativas podem resultar na necessidade de mudanças em um projeto de software que vinha sendo conduzido dentro de uma instituição bancária. O desenvolvimento de software tradicionalmente é afetado por diversas mudanças ao longo dos projetos.

O Sistema de Produção da Toyota trabalha com o princípio de adiar decisões até o último momento responsável, ou seja, aquele a partir do qual a não tomada da decisão traria prejuízos diretos ao projeto. O objetivo é aguardar até que informações mais

concretas estejam disponíveis para a equipe. Desta forma, procura-se reduzir a necessidade de re-trabalho em função de decisões tomadas cedo demais e que mais tarde possam ser forçadas a mudar em função de mudanças nas circunstâncias (POPPENDIECK & POPPENDIECK, 2003).

> Práticas de desenvolvimento que permitam adiar a tomada de decisões são eficazes em domínios que envolvem incerteza, porque elas provêm uma abordagem baseada em opções. Diante de incertezas, a maioria dos mercados econômicos desenvolve opções para prover uma forma de o investidor evitar se trancar em decisões até que o futuro esteja mais perto e mais fácil de prever. Adiar decisões é valioso porque é possível tomar melhores decisões quando elas são baseadas em fatos e não especulações. Em um mercado em evolução, manter decisões de design em aberto é mais valioso que se comprometer cedo demais. Uma estratégia chave para adiar compromissos durante o desenvolvimento de um sistema complexo é incorporar a capacidade de mudança no próprio sistema (POPPENDIECK & POPPENDIECK, 2003, p.xxvi, tradução nossa)

O Extreme Programming trabalha com este conceito evitando a implementação de funcionalidades baseadas em especulações. Além disso, usa o desenvolvimento iterativo para assegurar que a equipe se concentre apenas em um conjunto reduzido de funcionalidades a cada iteração, deixando que o tempo traga maiores informações sobre as funcionalidades futuras.

4.3.4 Entregar o mais rapidamente possível

Feedback é um conceito chave na filosofia *just-in-time*, porque quanto mais rico e mais rápido for o feedback, maior será o aprendizado. Quando uma equipe é capaz de fazer entregas rápidas, mesmo que cada uma se refira apenas a um conjunto reduzido de funcionalidades, é possível aprender e aprimorar o que está sendo produzido, bem como a forma de produção.

> No desenvolvimento, o ciclo de descoberta é crítico para o aprendizado: faça o design, implemente, obtenha feedback, melhore. Quanto mais curtos são estes ciclos, mais se pode aprender. A velocidade assegura que os clientes obtenham o que desejam agora e

> não aquilo que eles precisavam ontem. Isso também permite que eles adiem a tomada de decisões sobre o que eles realmente querem até que eles saibam mais (POPPENDIECK & POPPENDIECK, 2003, p.xxvi, tradução nossa).

É difícil adiar decisões quando as entregas demoram demais a ocorrer. Se uma equipe de desenvolvimento só é capaz de efetuar entregas a cada seis meses, uma vez que se defina o escopo de um período de seis meses de trabalho, o cliente pode vir a ter que aguardar no mínimo seis meses antes de ver qualquer mudança de idéia ser colocada em prática. Por outro lado, se a equipe faz entregas a cada duas semanas, mudanças de rumo podem ser incorporadas mais rapidamente, o que permite adiar decisões sem que isso gere conseqüências indesejáveis (POPPENDIECK & POPPENDIECK, 2003).

4.3.5 Delegar poder à equipe

Como pudemos observar anteriormente, é importante que o trabalhador do conhecimento possa definir a forma de executar suas tarefas. Ou seja, é essencial que ele tenha o domínio sobre o processo de desenvolvimento e, portanto, tenha a oportunidade de aprimorá-lo ao longo do tempo com o objetivo de obter a melhor qualidade possível.

> Executar atividades com a máxima qualidade depende de obter os detalhes corretamente e ninguém entende melhor dos detalhes que as pessoas que efetivamente executam o trabalho. Envolver desenvolvedores nos detalhes das decisões técnicas é fundamental para alcançar a excelência. As pessoas na linha de frente combinam o conhecimento do detalhe do último minuto com o poder de muitas mentes. Quando equipados com a qualificação técnica necessária e guiados por um líder, eles tomarão melhores decisões técnicas e melhores decisões de processo que qualquer um possa tomar por eles. Pelo fato de as decisões serem adiadas e a execução ser rápida, não é possível para uma autoridade central orquestrar as atividades dos trabalhadores (POPPENDIECK & POPPENDIECK, 2003, p.xxvi, tradução nossa).

Este é mais um princípio no qual a produção enxuta se afasta da idéia de divisão entre planejamento e execução. Ao invés disso, procura-se assegurar que o conhecimento gerado no chão de fábrica circule da melhor maneira possível, se enriqueça e retorne para o chão de fábrica rapidamente na forma de melhorias no processo.

4.3.6 Incorporar integridade

Outra forma de reduzir desperdícios é assegurar que o produto tenha elevada integridade perceptível e conceitual. A integridade perceptível é alcançada "quando um usuário pensa 'Isso! É exatamente o que eu quero. Alguém entrou na minha cabeça!' (POPPENDIECK & POPPENDIECK, 2003, p.xxvii, tradução nossa)." Ou seja, quando o software possui uma interface intuitiva e fácil de utilizar, cujos conceitos se relacionam de maneira harmônica.

Tal nível de integridade é importante para reduzir desperdícios na medida em que reduz ou elimina possíveis chamados dos usuários contendo dúvidas ou reclamações. Além disso, eleva a satisfação do usuário final.

A integridade conceitual, por sua vez, "significa que os conceitos centrais do sistema trabalham em conjunto como um todo harmônico e coeso; e é um fator crítico para a criação da percepção de integridade (POPPENDIECK & POPPENDIECK, 2003, p.xxvii, tradução nossa)." Em outras palavras, a partes se encaixam de maneira coerente, tornando fácil re-conectar os componentes e reduzindo as chances de defeitos. Ambos são fatores que geram redução de desperdícios.

> .
> Software íntegro possui uma arquitetura coerente, alcança uma pontuação elevada em usabilidade e adequação ao propósito e é manutenível, adaptável e extensível. Pesquisas revelam que a integridade deriva de liderança sábia, qualificação significativa, comunicação eficaz e disciplina sadia; processos, procedimentos e

> medições não são substitutos adequados (POPPENDIECK & POPPENDIECK, 2003, p.xxvii, tradução nossa).

Os demais princípios são essenciais para se alcançar integridade, na medida em que ajudam a reduzir os ciclos de feedback e, portanto, procuram amplificar o aprendizado.

4.3.7 Ver o todo

Para que um software alcance integridade perceptível e conceitual, é importante que o todo seja harmônico. Portanto, não é desejável que um determinado aspecto do projeto seja extremamente otimizado, enquanto outros tenham comportamentos deficientes. Para atingir integridade, o equilíbrio é mais importante que o comportamento individual das partes envolvidas. É necessário que a equipe de desenvolvimento seja capaz de ter a visão do todo permanentemente.

> Integridade em sistemas complexos requer profunda qualificação em muitas áreas diferentes. Um dos problemas mais intratáveis no desenvolvimento de produtos é que especialistas em qualquer área (banco de dados ou interface gráfica, por exemplo) têm a tendência de maximizar o desempenho de uma parte do produto representando a sua própria especialidade ao invés de focar no desempenho geral do sistema. Com bastante freqüência, o bem comum acaba sofrendo se as pessoas se comprometem primariamente com seus próprios interesses especializados. Quando indivíduos ou organizações são medidos de acordo com suas contribuições especializadas, ao invés do desempenho geral, é provável que ocorra sub-otimização (POPPENDIECK & POPPENDIECK, 2003, p.xxvii, tradução nossa).

Para solucionar os problemas de sub-otimização equipes *just-in-time* procuram elevar o nível das medições. Procura-se medir o resultado final e não as partes individuais, ou seja a atenção é voltada para o equilíbrio do conjunto. Além disso, utilizam-se os demais princípios para estabelecer um fluxo de comunicação rico que ajude a equipe a ter a visão do todo (POPPENDIECK & POPPENDIECK, 2003).

4.4 Processos de desenvolvimento de software

Como vimos no segundo capítulo, o termo "crise do software" acompanha a indústria de software desde 1968, quando ocorreu a conferência da OTAN que cunhou o nome Engenharia de Software. O termo Engenharia de Software, deu origem a uma disciplina dentro da área de computação, embora originalmente tivesse sido criado apenas como uma provocação, conforme os relatos de Peter Naur e Brian Randell.

> O termo "engenharia de software" foi escolhido deliberadamente para ser provocativo, cuja implicação era a necessidade de que a manufatura de software fosse baseada nos tipos de fundamentos teóricos e disciplinas práticas que são tradicionais em ramos estabelecidos da engenharia (MCBREEN, 2002, p.xv, tradução nossa).

A Engenharia de Software surgiu com o objetivo de atender às necessidades da OTAN para o desenvolvimento de grandes sistemas de defesa. Em princípio, seu escopo não envolvia projetos de aplicações comerciais que, de um modo geral, são bem menores e precisam entrar em produção em pouco tempo. Nestes casos, são raros os projetos desenvolvidos por equipes de mais de 20 pessoas e a maioria dos desenvolvedores trabalha em equipes com menos de 10 membros (MCBREEN, 2002, p.xvi, tradução nossa).

Segundo o *IEEE Standard Computer Dictionary*, a engenharia de software "é a aplicação de uma abordagem sistemática, disciplinada e mensurável para desenvolvimento, operação e manutenção de software; isto é, a aplicação da engenharia ao software (MCBREEN, 2002, p.7, tradução nossa)." O objetivo desta abordagem é alcançar na área de software o mesmo nível de previsibilidade, determinismo e acerto presente em outros ramos da engenharia.

A Engenharia de Software deu origem a diversos processos de desenvolvimento. O mais conhecido e antigo deles é o processo linear e seqüencial de desenvolvimento,

também conhecido como cascata. Ele organiza os projetos de software em quatro grandes etapas realizadas seqüencialmente: análise, design, codificação e testes. Ainda hoje, este á o modelo de desenvolvimento mais conhecido e amplamente utilizado (PRESSMAN, 1997).

Ele pode ser relacionado facilmente aos princípios da produção em massa utilizados por Taylor. A evolução seqüencial do projeto se assemelha a uma fábrica onde requisitos são tratados como matérias primas que são transformadas à medida que avançam pela linha de produção. Cada transformação gera um conjunto de artefatos a serem utilizados em etapas posteriores da fabricação. Procura-se assegurar que cada artefato seja produzido corretamente para que o resultado final possa ser alcançado com a mesma previsibilidade e determinismo de uma fábrica. Portanto, evitar variações é fundamental.

Dentro do projeto, grande parte dos artefatos é representada por documentos criados para direcionar o trabalho que será executado mais adiante na cadeia de produção, os quais são produzidos por profissionais especializados. Normalmente o processo de planejamento é feito no início da cadeia, enquanto espera-se que os artefatos produzidos ao longo do projeto permitam que as últimas etapas da cadeia sejam produzidas de forma cada vez mais mecânica e determinística. Portanto, observam-se dois princípios fundamentais da racionalização do trabalho manual: a especialização e a centralização.

> Durante um projeto, três habilidades diferentes são necessárias:
> - Analistas para documentar os requisitos;
> - Projetistas para criar a especificação do design e
> - Programadores para escrever o código.
>
> A cada etapa, os autores de cada documento têm que adicionar detalhes extras porque não sabem quem irá ler o documento mais adiante. Sem saber que tipo de conhecimento em comum pode ser assumido, a única coisa segura a fazer é adicionar o máximo de detalhes e referências cruzadas que o autor conheça. Os revisores

> precisam, então, analisar o documento para confirmar que ele esteja completo e sem ambigüidades. Documentação completa gera outro desafio: os membros da equipe precisam assegurar que os documentos permaneçam consistentes quando ocorrem mudanças nos requisitos ou mudanças de design feitas durante a codificação. (...) Como as mudanças se tornam muito caras devido à necessidade de atualização dos documentos, acabam sendo controladas e evitadas (MCBREEN, 2002, p.6, tradução nossa).

A utilização de um processo seqüencial, baseado na criação de artefatos, é uma tentativa de disciplinar o desenvolvimento e buscar os níveis de previsibilidade desejados. Entretanto, freqüentemente o resultado obtido é apenas formalismo e não disciplina. Em parte isso ocorre porque como já vimos, mudanças são constantes em projetos de software e o modelo seqüencial de desenvolvimento gera uma necessidade de controlá-las, o que, por sua vez, leva a um conflito de interesses entre clientes e equipes de desenvolvimento. Além disso, esse modelo revela poucas chances para a aquisição e utilização de feedback ao longo do desenvolvimento, elevando os riscos de que o projeto entregue algo que não resolva as necessidades dos usuários, mesmo seguindo um planejamento e controlando o processo.

DeMarco (DEMARCO & BOEHM, 2002) acredita que o foco na geração de artefatos, ao invés de ajudar, acabou se tornando uma forma de agravar os problemas da "crise do software". Segundo ele, "Depois de quase 20 anos de obsessão por processos (...), o processo que encontro tipicamente em empresas clientes se tornou excessivamente baseado em documentações, gerando a enxurrada de documentos que se tornou endêmica (...)."

Ao longo do tempo, a indústria de software criou alternativas para o processo de desenvolvimento em cascata. Algumas mantiveram parte das características do modelo seqüencial, enquanto outras se baseiam em formas completamente diferentes de se tratar os projetos de software.

O Rational Unified Process (RUP), por exemplo, traz avanços em relação ao desenvolvimento em cascata, embora mantenha conceitos herdados do taylorismo. Ele é um *framework* de processo bastante abrangente que, como tal, pode ser instanciado para atender às necessidades dos mais diversos tipos de projetos de software (JACOBSON, BOOCH et al., 1999).

O RUP é organizado em torno do conceito de "**melhores práticas**". Ele provê um vasto arcabouço de práticas que procuram indicar a melhor forma de se realizar diversos tipos de atividades nos projetos de software. Neste sentido, ele se aproxima da abordagem taylorista que também buscava identificar as melhores formas de estruturar o trabalho dentro de uma fábrica.

A proposta é que, ao iniciar um projeto que utilizará o RUP, a equipe de desenvolvimento selecione dentre as "melhores práticas" disponíveis no arcabouço, aquelas que fazem sentido para o projeto em questão. Embora seja uma proposta intuitivamente justificável e louvável, existem alguns problemas práticos que foram levantados por Boehm e Turner ((2003).

Em seu arcabouço, o RUP engloba também um vasto conjunto de artefatos, além de "melhores práticas". Determinar que práticas e artefatos devem ser adotados em um projeto é uma atividade que pode ser bem feita por pessoas com treinamento, conhecimento e experiência no RUP. Entretanto, é raro encontrar pessoas com essas características nos projetos de software. Na falta delas, os membros das equipes de projeto tendem a adotar um conjunto desnecessariamente abrangente de práticas e artefatos temendo retirar elementos que possam fazer falta mais tarde, o que freqüentemente leva à burocratização dos projetos.

Não é objetivo do RUP tornar os projetos “pesados” e “enfadonhos”, entretanto, o que Boehm e Turner observam é que abordagens baseadas em boas práticas como o RUP (entre outras), freqüentemente levam os projetos a se burocratizarem devido à forma como os membros da equipe adotam os conceitos. O desconhecimento e a inexperiência normalmente se tornam um convite para o excesso, embora este não seja o objetivo destes tipos de processos.

Outra questão contraditória na idéia de “melhores práticas” está no fato de que não existem “melhores práticas” em absoluto. A avaliação do que vem a ser uma melhor prática normalmente precisa levar em conta o contexto em que a prática é utilizada. Pois, enquanto em determinados casos a prática pode se mostrar perfeita, em outros, pode se revelar desastrosa devido a características do projeto.

Além de se basear em um conjunto de boas práticas, o RUP é orientado a casos de uso, centrado na arquitetura, iterativo e incremental (JACOBSON, BOOCH et al., 1999). Os casos de uso são utilizados pelas equipes para “capturar” os requisitos funcionais e indicar que atores utilizarão as funcionalidades implementadas.

Muitas equipes encaram os casos de uso como o mecanismo essencial para o levantamento dos requisitos, o que é problemático, porque significa que freqüentemente os casos de uso serão interpretados como elementos completos e suficientes para a implementação das funcionalidades. Basear a compreensão dos requisitos em canais de comunicação escritos é problemático, conforme veremos no próximo capítulo.

Ao ser centrado na arquitetura, o RUP também incentiva (direta ou indiretamente) as equipes a estabelecerem a arquitetura do software antes de começar a implementação do mesmo. Portanto, a utilização de casos de uso para os requisitos e a elaboração de documentos que descrevem a arquitetura antes da implementação

freqüentemente levam equipes a adotarem um modelo de desenvolvimento bastante semelhando ao cascata, embora exista uma diferença fundamental na proposta do RUP: o desenvolvimento iterativo e incremental.

Em princípio, o RUP estabelece o desenvolvimento iterativo e incremental como forma de incorporar feedback e aprendizado ao processo de desenvolvimento. Entretanto, é comum equipes adotarem o RUP com iterações muito longas ou simplesmente executarem o projeto inteiro em uma única iteração. Nestes casos, o que se observa é a equipe executando um projeto de acordo com o processo em cascata, mas basicamente usando os artefatos do RUP para organizar a documentação.

Uma das maiores falhas do processo de desenvolvimento em cascata e de outras propostas de desenvolvimento baseadas em princípios tayloristas é a incapacidade de levar em conta o fator humano de forma apropriada. Como vimos anteriormente, Taylor tornou produtivo o trabalho manual, mas seus princípios não são aplicáveis para o trabalho do conhecimento, em especial o desenvolvimento de software.

> Um bom software não se origina de ferramentas CASE, programação visual, prototipagem rápida ou tecnologia de objetos. Um bom software é resultado de pessoas. Assim como é o caso de softwares ruins. (...) já que software é criado por pessoas e usado por pessoas, uma melhor compreensão das pessoas – como executam o trabalho e como trabalham em conjunto – é a base para um melhor desenvolvimento de software e para criarmos softwares melhores (CONSTANTINE, 2001, p.xvii, tradução nossa).

Na década de 1990, alguns profissionais da indústria de software começaram a propor novos processos de desenvolvimento que fossem mais adequados para lidar com os aspectos humanos dos projetos de software. Em fevereiro de 2001, 17 profissionais experientes se reuniram em Utah (EUA) para discutir suas práticas de desenvolvimento e suas propostas alternativas para evitar os processos de desenvolvimento excessivamente baseados em documentações e formalismos. Naquele momento,

decidiram organizar suas propostas sob um nome comum: **desenvolvimento ágil de software**. Isto foi feito pelo lançamento do Manifesto pelo Desenvolvimento Ágil de Software[3].

O manifesto estabelece um conjunto de valores que são adotados nos projetos ágeis:

- **Indivíduos e interações** ao invés de processos e ferramentas;
- **Software funcionando** ao invés de documentação abrangente;
- **Colaboração com o cliente** ao invés de negociação de contratos e
- **Responder a mudanças** ao invés de seguir um plano.

A proposta é que, embora exista valor nos itens à direita, os processos ágeis valorizam mais os itens que estão à esquerda. Atualmente, estes são os principais processos ágeis conhecidos: *Scrum*, *Dynamic Systems Development Method (DSDM)*, *Crystal Methods*, *Feature-Driven Development (FDD)*, *Lean Development (LD)*, *Extreme Programming* e *Adaptative Software Development* (HIGHSMITH, 2002).

Uma das principais diferenças dos processos ágeis em relação aos seus antecessores é o conceito chamado de *barely sufficient*, ou seja, **mínimo necessário**. Enquanto abordagens como o RUP (entre outras) procuram estabelecer um arcabouço de "melhores práticas", os processos ágeis sugerem o uso de um conjunto bastante reduzido de práticas. Tal conjunto pode se revelar suficiente em muitos projetos comerciais que envolvam equipes reduzidas, tais como os que foram mencionados no início desta seção.

O objetivo é começar os projetos de software de forma simples, com poucas práticas e avaliar os resultados. Caso práticas ou artefatos faltem ao projeto, deve-se

---

[3] http://www.agilemanifesto.org.

buscá-los em outras propostas metodológicas. Com isso, procura-se evitar as chances de que um projeto seja iniciado com um conjunto desnecessariamente elevado de práticas e artefatos que possam torná-lo burocrático. A ênfase passa a ser de só trazer elementos novos para o projeto quanto os mesmos realmente se mostram necessários (BOEHM & TURNER, 2003; COCKBURN, 2002; HIGHSMITH, 2002). O próximo capítulo apresentará o Extreme Programming que representa um dos processos ágeis mais conhecidos e utilizados.

5 EXTREME PROGRAMMING

Os principais fundamentos do XP tiveram origem nas tradições do desenvolvimento em Smalltalk e datam de meados da década de 80, quando Kent Beck e Ward Cunningham trabalhavam na Tektronixs, Inc. Práticas, tais como, refatoração, programação em par, mudanças rápidas, feedback constante do cliente, desenvolvimento iterativo, testes automatizados, entre outras, são elementos centrais da cultura da comunidade Smalltalk. Olhando deste ponto de vista, XP pode ser considerado o modo de agir do Smalltalk generalizado para outros ambientes.

Entre 1986 e 1987, começaram a surgir algumas contribuições fundamentais para aquilo que viria a ser o XP, uma década depois. De 1986 a 1996, Kent e Ward desenvolveram um amplo conjunto de boas práticas que foram condensadas sucintamente no padrão de linguagem *Episodes*. Este padrão foi publicado em 1996 sob o título *Pattern Languages of Program Design 2*.

Ainda neste mesmo período, entre 1989 e 1992, surgiram importantes avanços em refatoração. Nesta área, destaca-se a tese de Bill Opdyke, *Refactoring Object-Oriented Frameworks*. Este trabalho demonstrou como pessoas como Kent e Ward obtinham ganhos de produtividade utilizando a refatoração.

Alguns anos mais tarde, por volta de 1996, Kent publicou o livro *Smalltalk Best Practices Patterns*. Este livro também apresentou boas técnicas de desenvolvimento, grande parte das quais foi combinada no trabalho de Martin Fowler et al. (2000).

O desenvolvimento orientado a testes é uma técnica que derivou diretamente das técnicas de refatoração. O primeiro artigo publicado sobre este conceito foi escrito por Kent Beck para a *SmalltalkReport*. Neste artigo, ele introduziu o *framework SmallUnit*.

A partir de então, o artigo mais importante sobre o assunto, intitulado "*Test Infected*", descreveu o JavaUnit na Dr. Dobb's.

Ainda nas questões técnicas, a comunidade XP é tradicionalmente conhecida por utilizar padrões (*patterns*) fortemente. Este é um outro aspecto importante no surgimento do XP, já que Kent e Ward começaram a aplicar os conceitos de padrões em 1987, quando escreveram um dos primeiros artigos sobre o assunto para a OOPSLA'87 (*Conference on Object-Oriented Programming, Systems, Languages, and Applications*).

Em 1996, todas as partes que foram sendo agregadas ao longo de uma década começaram a se fundir. No início daquele ano, Kent trabalhava como consultor para problemas de desempenho em SmallTalk, quando foi chamado para analisar o desempenho do projeto de conversão da folha de pagamento da Chrysler para SmallTalk. O sistema em questão, conhecido como C3, ou *Chrysler Comprehensive Compensation System* (Sistema de Compensação Abrangente da Chrysler), é conhecido como o berço do XP e foi onde Kent Beck utilizou pela primeira vez, em conjunto, as práticas que atualmente formam a estrutura do Extreme Proramming (BECK & ANDRES, 2005; TELES, 2004).

Neste capítulo e no seguinte, procuraremos mostrar que o conjunto de práticas e valores do Extreme Programming é coeso e possui características que permitem utilizar o XP como uma alternativa válida na busca por maiores taxas de sucesso nos projetos de software. Este capítulo apresenta cada valor e prática do XP com base em inúmeras publicações que apóiam o uso dos mesmos.

O enfoque deste capítulo é no **porquê** dos conceitos propostos pelo Extreme Programming. Atualmente existem boas publicações explicando **o que** são os valores e as práticas do XP, além de demonstrarem **como** utilizá-los. Sendo assim, decidimos

investir esforços sobretudo em explicar o **porquê** dos conceitos propostos pelo XP, apresentando brevemente o que são, mas sem entrar em detalhes sobre como utilizá-los. Caso contrário, o escopo deste capítulo seria excessivamente amplo e o texto maior que o necessário para o propósito desta dissertação.

No capítulo seguinte, apresentaremos um estudo de caso que valida, ao menos em um projeto em particular, que a proposta do XP realmente é capaz de gerar resultados positivos.

5.1 Valores

5.1.1 Feedback

A compreensão das necessidades dos usuários é uma das atividades mais difíceis e importantes de serem realizadas pelos desenvolvedores de um software, pois ela direciona todos os demais esforços. Entretanto, compreender os requisitos é freqüentemente difícil, bem como costuma ser complexo para os próprios usuários transmiti-los corretamente. Segundo Brooks (1987, tradução nossa), "nenhuma outra parte do trabalho conceitual é tão difícil quanto estabelecer detalhadamente os requisitos técnicos, incluindo todas as interfaces (...) Nenhuma outra parte é mais difícil de corrigir mais tarde."

> Portanto, a função mais importante que os construtores de software podem desempenhar por seus clientes é a extração e o refinamento iterativo dos requisitos do produto. Porque a verdade é que os clientes não sabem o que querem. Eles normalmente não sabem que questões precisam ser respondidas e eles quase nunca pensaram no problema no nível de detalhe que precisa ser especificado. (...) As dinâmicas das ações são difíceis de imaginar. Portanto (...) é necessário dar espaço para uma interação abrangente entre o cliente e o designer como parte da definição do sistema (BROOKS, 1987, tradução nossa).

Na opinião de Brooks, os clientes não têm como prever corretamente as funcionalidades de que necessitarão. Por isso, é fundamental que haja uma forte interação com os desenvolvedores ao longo do projeto.

> Eu daria um passo além e afirmaria que, na verdade, é impossível para os clientes, mesmo aqueles trabalhando com engenheiros de software, especificar completamente, precisamente e corretamente os requisitos exatos de um produto de software moderno antes de ter construído e tentado algumas versões do produto que estão especificando (BROOKS, 1987, tradução nossa).

A compreensão das necessidades dos usuários é um processo de aprendizado contínuo no qual os desenvolvedores aprendem sobre os problemas do negócio e os usuários tomam conhecimento das dificuldades e limitações técnicas. "Um princípio psicológico bem conhecido indica que para maximizar a taxa de aprendizado, a pessoa precisa receber feedback sobre quão bem ou mal ele está indo (WEINBERG, 1971, p.102, tradução nossa)."

Amplificar o aprendizado é importante porque ajuda a acelerar a **convergência entre as necessidades e a compreensão das mesmas** por parte dos desenvolvedores. Além disso, acelera o entendimento dos usuários sobre as possibilidades da tecnologia e suas limitações. A convergência é ainda mais rápida quando os ciclos de feedback são encurtados. "Um dos maiores pontos que aceleram a melhoria do desempenho de um sistema (...) é a minimização das [suas] defasagens (SENGE, 2002, p.119-121)."

> A psicologia do aprendizado ensina que o tempo entre uma ação e o correspondente feedback é crítico para o aprendizado. Experimentos com animais mostram que mesmo pequenas diferenças no tempo de feedback resultam em enormes diferenças de aprendizado. (...) Portanto, um dos princípios é obter feedback, interpretá-lo, e colocar o que foi aprendido de volta dentro do sistema o mais rapidamente possível. As pessoas de negócio aprendem como o sistema pode contribuir da melhor forma e retornam este aprendizado em dias ou semanas, ao invés de meses ou anos. Os programadores aprendem como fazer o design, implementar e testar o sistema da melhor forma possível e retornam este aprendizado em segundos ou minutos ao invés de dias, semanas ou meses (BECK, 2000, p.37, tradução nossa).

Por estas razões, o Extreme Programming é organizado em ciclos curtos de feedback que possibilitem aos usuários solicitar funcionalidades e aprender sobre elas através de software funcionando em prazos curtos. Esse processo envolve a priorização de poucas funcionalidades a serem implementadas de cada vez e a simplificação das mesmas na medida do possível. O objetivo se torna apresentar a funcionalidade ao usuário rapidamente, de modo que ele possa, cedo, detectar eventuais falhas, quando tende a ser mais barato corrigi-las. "A razão básica para estratégias incrementais e iterativas é permitir que os inevitáveis erros das pessoas sejam descobertos relativamente cedo e reparados de forma metódica (COCKBURN, 2002, p.49, tradução nossa)."

Trabalhando com ciclos de feedback curtos, o Extreme Programming procura assegurar que pouco trabalho seja efetuado e concluído de cada vez. A equipe segue adiante apenas se o resultado estiver correto. Caso surjam falhas, as mesmas são corrigidas logo, antes de iniciar o desenvolvimento de outras funcionalidades. A utilização de lotes reduzidos de trabalho assegura que eventuais falhas tenderão a ser corrigidas com maior rapidez exatamente porque o escopo do trabalho é reduzido, o que significa que menos coisas podem dar errado.

5.1.2 Comunicação

Projetos de software normalmente envolvem a presença de pelo menos duas pessoas, um usuário e um desenvolvedor, o que causa a necessidade de comunicação entre elas. No mínimo, cabe ao usuário comunicar o que necessita que seja produzido e ao desenvolvedor comunicar as considerações técnicas que afetam a solução e a velocidade de implementação da mesma.

Muitos projetos envolvem não apenas duas pessoas, mas sim grupos maiores que podem ser compostos por diversos usuários e desenvolvedores. Com freqüência, equívocos no processo de comunicação, causam desentendimentos ou compreensão incorreta de algum aspecto do projeto.

> Como os sociólogos sabem, comunicação é intrinsecamente difícil, mediada por códigos que são sempre contextuais, normas, culturas e percepções. (...)
> A construção de requisitos básicos, por exemplo, envolve um processo de comunicação de conhecimento tácito, o que explica grande parte da dificuldade no desenvolvimento de software. Traduzir conhecimento de um contexto para outro, como traduzir qualquer língua, não envolve apenas gramática básica e regras sintáticas, mas também questões de significado e intenção que são contextuais e subjetivas. (...)
> De um modo geral, software oferece um exercício de traduzir algoritmos existentes – na natureza, organizações ou práticas – para a forma digital. Grande parte deste conhecimento de domínio é tácito, indefinido, não codificado e desenvolvido ao longo do tempo, freqüentemente sem ser explícito até mesmo para os indivíduos que participaram do processo. Ainda mais importante, este conhecimento e as práticas são dinâmicos, evoluindo constantemente e se transformando (EISCHEN, 2002, p.39, tradução nossa).

A transmissão de conhecimento tácito representa um desafio significativo para as equipes de desenvolvimento, o qual pode ser solucionado de forma mais ou menos eficaz dependendo dos mecanismos de comunicação adotados no projeto. “A forma de se transmitir uma idéia exerce uma grande influência na compreensão correta da mesma (TELES, 2004, p.48).”

Quando uma pessoa está na presença de outra e transmite uma idéia através de um diálogo, o interlocutor tem acesso a vários elementos que compõem a comunicação, tais como expressões faciais, gestos, postura, palavras verbalizadas e tom de voz. A mesma conversa por telefone, seria desprovida de todos os elementos visuais. Portanto, a comunicação por telefone, por exemplo, é menos rica em elementos que um diálogo presencial, o que torna mais difícil compreender a idéia transmitida.

A riqueza do meio de comunicação exerce influência ainda maior quando se observa a transmissão de conhecimento tácito. Imaginemos uma situação na qual uma pessoa ao telefone tenta ensinar a seu interlocutor como dar um laço no cadarço de seu tênis. Usando o telefone, as chances de sucesso são reduzidas. Entretanto, se os interlocutores estivessem na presença um do outro, seria fácil demonstrar como dar um laço através de um exemplo. Além disso, à medida que o aprendiz fizesse algumas tentativas, o mentor poderia fornecer feedback de modo a corrigir eventuais erros.

Em muitos projetos, os usuários executam atividades cotidianas de modo tácito, com efeitos semelhantes ao exemplo acima. Sendo assim, têm dificuldades em explicar o que fazem usando um meio de comunicação pouco rico (como telefone, ou email, por exemplo), mas são capazes de mostrar o que fazem com alguma facilidade se a comunicação puder ocorrer através de um diálogo presencial. Essa é uma das razões pelas quais o Extreme Programming prioriza mecanismos de comunicação mais ricos.

> Uma equipe XP faz um excelente uso de comunicação osmótica, comunicação face-a-face, correntes de convecção no fluxo da informação e radiadores de informação nas paredes.
> A disponibilidade consistente de especialistas significa que o tempo entre uma pergunta e sua resposta é curto. O tempo e a energia gastos para descobrir uma informação demandada é baixo; a taxa de dispersão da informação é alta (COCKBURN, 2002, p.167, tradução nossa).

Projetos XP procuram envolver ativamente seus usuários (ou ao menos um representante dos mesmos) fazendo com que se tornem parte integrante da equipe de desenvolvimento. Na prática, isso significa que o usuário (ou seu representante) está presente no mesmo local onde os desenvolvedores trabalham, possibilitando que eles tenham acesso rápido e direto a um ou mais especialistas no domínio do negócio. Isso ajuda a acelerar o fluxo de informações e permite que a comunicação se baseie prioritariamente em diálogos presenciais.

A rapidez na comunicação é um aspecto relevante, pois "o ritmo de progresso de um projeto está ligado ao tempo que se leva para transmitir uma informação da mente de uma pessoa para outra (COCKBURN, 2002, p.77, tradução nossa)." Além disso, " os custos de um projeto crescem na proporção do tempo necessário para as pessoas se compreenderem (COCKBURN, 2002, p.81, tradução nossa)."

A proximidade dos participantes auxilia os processos de comunicação. Mas o XP também emprega outros mecanismos, como os **radiadores de informação** (COCKBURN, 2002) que, além de facilitar a comunicação, ajudam a colocar em prática um modelo de auto-direcionamento do processo de desenvolvimento.

> Um radiador de informação mostra informações onde transeuntes podem vê-la. Com radiadores de informação, eles não precisam ficar perguntando; a informação simplesmente chega a eles à medida que passam pelo local (COCKBURN, 2002, p.84, tradução nossa).

> [Radiadores de informação são uma forma de] *controle visual*, ou *gestão à vista*. Se o trabalho será autodirecionado, então todos precisam ser capazes de ver o que está acontecendo, o que precisa ser feito, que problemas existem, que progresso está sendo alcançado. O trabalho não pode ser auto-direcionado até que controles visuais simples, apropriados para o domínio, sejam colocados em uso, atualizados e usados para direcionar o trabalho (POPPENDIECK & POPPENDIECK, 2003, p.76, tradução nossa).

Outro fator que influencia a qualidade da comunicação é a quantidade de pessoas envolvidas. "Com cada aumento no tamanho [da equipe], torna-se mais difícil para as pessoas saber o que os outros estão fazendo e como não sobrepor, duplicar ou interferir no trabalho um do outro (COCKBURN, 2002, p.151, tradução nossa)." Por esta razão, projetos XP procuram contar com um número reduzido de participantes (freqüentemente menor que uma dúzia de pessoas) (BECK, 2000).

> O limite essencial não é o número de ferramentas ou falta de organização, mas sim comunicação. O aumento da quantidade de linhas de comunicação e a qualidade das mesmas, e não o número de

> pessoas, complica o desenvolvimento de software (EISCHEN, 2002, p.39, tradução nossa).

5.1.3 Simplicidade

Como vimos anteriormente, estudos do Standish Group revelam que 45 por cento das funcionalidades encontradas em um sistema típico jamais são usadas e 19 por cento raramente são utilizadas, totalizando 64 por cento de funcionalidades que poderiam nem sequer ter sido implementadas. Em outras palavras, os projetos de software freqüentemente investem grande parte dos recursos (tempo, pessoas e dinheiro) em esforços desnecessários.

Em diversos projetos observa-se a ocorrência de três fenômenos que ajudam a esclarecer as razões dos resultados acima:

- O escopo é fixado no início e alterações no mesmo são evitadas;
- Os desenvolvedores criam soluções genéricas para "facilitar" possíveis alterações que possam ocorrer no escopo e
- Os desenvolvedores produzem funcionalidades adicionais na tentativa de antecipar o que o usuário "certamente" irá solicitar no futuro.

Todos os casos acima derivam de preocupações legítimas e justificáveis dos clientes e desenvolvedores. No primeiro caso, o cliente teme não receber todas as funcionalidades que imagina necessitar, o que o leva a fixar o escopo. Nos últimos dois casos, o desenvolvedor teme que o cliente peça alterações tardias no sistema que possam gerar atrasos. Assim, procura tornar o software mais "flexível", de modo que mudanças possam ser implementadas ajustando-se parâmetros para evitar recodificar partes do sistema. Além disso, tenta antever funcionalidades que "provavelmente" serão

necessárias (embora o cliente ainda não perceba isso) e as implementa de modo que, quando o cliente as solicitar, já estarão prontas.

Estas preocupações são justificáveis, porém as abordagens utilizadas apresentam problemas. Fixar o escopo é arriscado, porque significa que tanto as funcionalidades realmente úteis, quanto as desnecessárias serão implementadas. A diferença entre elas só costuma ser notada ao longo do projeto, à medida que os usuários conseguem obter feedback do software. Portanto, uma alternativa é a adoção de um modelo de desenvolvimento iterativo, com iterações curtas, no início das quais o cliente possa priorizar as funcionalidades (POPPENDIECK & POPPENDIECK, 2003).

> Visto que os clientes normalmente não sabem direito o que desejam no início do projeto, eles tendem a pedir tudo o que acham que podem vir a precisar, especialmente se eles pensarem que só terão uma única chance de pedir. Esta é uma das melhores maneiras que conhecemos para aumentar o escopo do projeto muito além do necessário para alcançar a missão geral do projeto (POPPENDIECK & POPPENDIECK, 2003, p.32-33, tradução nossa).

Para que uma equipe de desenvolvimento possa trabalhar com iterações curtas, é necessário que ela seja capaz de receber um pequeno escopo de funcionalidades no início de cada iteração e implementá-las completamente dentro de um curto prazo de tempo. Isso cria a necessidade de concentrar esforços apenas no essencial para implementar as funcionalidades da iteração, evitando generalizações que ainda não se mostrem necessárias e a criação de funcionalidades que ainda não tiverem sido solicitadas pelos usuários. Existem vantagens nesta abordagem, pois "(...) adicionar suporte para futuras funcionalidades de forma desnecessária complica o design e eleva o esforço para desenvolver incrementos subseqüentes (BOEHM & TURNER, 2003, p.41, tradução nossa)."

> Simplicidade e comunicação possuem uma maravilhosa relação de apoio mútuo. Quanto mais você comunica, mais claramente você é capaz de ver o que precisa ser feito e mais confiança você tem sobre o que realmente não precisa ser feito. Quanto mais simples é o seu sistema, menos você precisa comunicar sobre ele, o que leva à comunicação mais completa, especialmente se você for capaz de simplificar o sistema suficientemente a ponto de necessitar de menos programadores (BECK, 2000, p.31, tradução nossa).

Os desenvolvedores de um projeto XP procuram implementar as funcionalidades priorizadas para cada iteração com a maior qualidade possível, porém com foco apenas naquilo que é claramente essencial. Generalizações que não se provem imediatamente necessárias são evitadas, pois "se você mantiver o sistema suficientemente simples o tempo todo, qualquer coisa que você coloque nele será inserido facilmente e na menor quantidade de lugares possível (JEFFRIES, ANDERSON et al., 2001, p.76, tradução nossa)."

Ao invés de tentar prever que mudanças o usuário solicitará e, portanto, que generalizações serão úteis, os desenvolvedores procuram simplificar o sistema, tornando-o mais fácil de ser alterado no futuro. "Como programadores, nos habituamos a antecipar problemas. Quando eles aparecem mais tarde, ficamos felizes. Quando não aparecem, nem notamos (BECK, 2000, p.104-105, tradução nossa)." Além disso, equipes XP se baseiam no princípio de que "funcionalidades extras adicionam complexidade e não flexibilidade (POPPENDIECK & POPPENDIECK, 2003, p.59, tradução nossa)." Sendo assim, procuram adiar a inclusão de qualquer funcionalidade até que ela realmente seja priorizada e solicitada pelo cliente.

> Pode parecer uma boa idéia colocar algumas funcionalidades extras no sistema para o caso de se tornarem necessárias. (...) Isso pode parecer inofensivo, mas, ao contrário, trata-se de um sério desperdício. Cada fragmento de código no sistema precisa ser rastreado, compilado, integrado e testado a cada vez que o código sofre uma intervenção, e então, precisa ser mantido durante toda a vida do software. Cada fragmento de código eleva a complexidade e é uma parte que pode

> falhar (POPPENDIECK & POPPENDIECK, 2003, p.6, tradução nossa).

5.1.4 Coragem

Existem temores que costumam assombrar os participantes de um projeto de software. Beck e Fowler (2001) destacam alguns destes medos que exercem influência significativa nos processos de desenvolvimento.

Clientes temem:

- Não obter o que pediram;
- Pedir a coisa errada;
- Pagar demais por muito pouco;
- Jamais ver um plano relevante;
- Não saber o que está acontecendo e
- Fixarem-se em suas primeiras decisões e não serem capazes de reagir a mudanças nos negócios.

Desenvolvedores, por sua vez, temem:

- Ser solicitados a fazer mais do que sabem fazer;
- Ser ordenados a fazer coisas que não façam sentido;
- Ficar defasados tecnicamente;
- Receber responsabilidades, sem autoridade;
- Não receber definições claras sobre o que precisa ser feito;
- Sacrificar a qualidade em função de prazo;
- Ter que resolver problemas complicados sem ajuda e

- Não ter tempo suficiente para fazer um bom trabalho.

Equipes XP reconhecem estes temores e buscam formas de lidar com eles de maneira corajosa. Ter coragem em XP significa ter **confiança** nos mecanismos de segurança utilizados para proteger o projeto. Ao invés de acreditar que os problemas não ocorrerão e fazer com que a coragem se fundamente nesta crença, projetos XP partem do princípio de que problemas irão ocorrer, inclusive aqueles mais temidos. Entretanto, a equipe utiliza redes de proteção que possam ajudar a reduzir ou eliminar as conseqüências destes problemas.

O cliente teme não obter o que pediu, ou ainda pior, pedir a coisa errada. Para protegê-lo, o XP adota iterações curtas (normalmente de uma a três semanas) e fixas (se a equipe opta por duas semanas, por exemplo, todas as iterações do projeto terão sempre esta duração). Desta forma, ao final de cada iteração, é possível avaliar se a equipe implementou o que foi pedido e se o que foi pedido realmente fazia sentido.

O problema não é eliminado em função do desenvolvimento iterativo. O cliente pode ter solicitado algo errado no início da iteração ou a equipe pode ter implementado de forma incorreta, mas isso é descoberto cedo. Como a iteração é curta, poucas funcionalidades são implementadas. Portanto, caso haja um erro, o mesmo se refere a um conjunto reduzido de funcionalidades, o que facilita eventuais correções e evita que a equipe invista muitos recursos em funcionalidades incorretas, caso o cliente tenha errado ao solicitá-las (POPPENDIECK & POPPENDIECK, 2003).

O desenvolvimento iterativo também ajuda a lidar com o medo que o cliente tem de pagar demais por muito pouco. Ao receber funcionalidades com freqüência, em prazos curtos, o cliente passa a ter diversas oportunidades de avaliar o trabalho da

equipe com base em feedback concreto: software executável. Assim ele pode decidir se continua ou não a empregar aquela equipe ou se é preferível trocar (TELES, 2004). Além disso, o feedback constante, produzido ao longo das iterações, faz com que o cliente possa saber exatamente o que está acontecendo no projeto.

Finalmente, o processo de planejamento não é estático. A cada início de iteração o planejamento geral do projeto é revisado e atualizado com base em informações mais recentes. Isto é, o processo de planejamento é contínuo e procura incorporar feedback ao longo do tempo. Isso permite a elaboração de planos para cada iteração que têm maiores chances de acerto. Além disso, no processo de priorização, o cliente pode incorporar novas decisões de negócios de forma natural (BECK & FOWLER, 2001).

Desenvolver software de forma iterativa e incremental não tem apenas vantagens. Também gera alguns riscos, como por exemplo o de introduzir falhas em algo que vinha funcionando corretamente. Por isso, o XP adota a prática de desenvolvimento orientado a testes como mecanismo básico de proteção. "O desenvolvimento orientado a testes é uma forma de lidar com o medo durante a programação (BECK, 2003, p.x, tradução nossa)."

Ele leva os desenvolvedores a criar uma base de testes automatizados que possam ser executados toda vez que um novo fragmento de código é adicionado ao sistema. Embora isso não impeça a ocorrência de erros, representa um instrumento útil para detectá-los rapidamente, o que agiliza a correção e evita que eventuais bugs se acumulem ao longo do tempo.

Os desenvolvedores temem não saber solucionar alguns problemas e não serem capazes de se atualizar tecnicamente. O XP utiliza a programação em par para permitir que os membros da equipe de desenvolvimento aprendam continuamente uns com os

outros. Além disso, a possibilidade de contar sempre com a ajuda imediata de um colega gera maior confiança na capacidade de resolver os desafios apresentados pelo cliente. Finalmente, a programação em par estabelece um processo permanente de inspeção de código, o que serve como uma rede de proteção adicional contra eventuais erros cometidos durante a codificação de novas funcionalidades ou alteração de outras previamente existentes (WILLIAMS & KESSLER, 2003).

Outra preocupação permanente dos desenvolvedores é não ter tempo suficiente para realizar um trabalho de qualidade. O XP trata essa questão dividindo claramente a responsabilidade por decisões técnicas e de negócio. O cliente tem soberania nas decisões de negócio. Portanto, ele decide que funcionalidades devem ser implementadas e em que ordem. Os desenvolvedores, por sua vez, têm autoridade e responsabilidade sobre as decisões técnicas. Portanto, são eles que estimam os prazos. Isso ajuda a lidar com o medo de ter que cumprir prazos impossíveis impostos por pessoas que não possuam a qualificação técnica para estimar o esforço de um determinado trabalho de programação (BECK & FOWLER, 2001).

5.2 Práticas

5.2.1 Cliente Presente

Imaginemos que uma pessoa esteja muito acima de seu peso ideal e decida buscar a orientação de uma nutricionista para formular uma dieta e ajudar a reduzir o peso. Na primeira consulta, a pessoa pergunta à nutricionista quanto tempo será necessário para que ela perca dez quilos. Infelizmente não existe uma resposta exata para esta pergunta, pois o resultado final depende do trabalho de ambas as partes. Da mesma forma que a nutricionista precisa ser capaz de elaborar uma dieta adequada, o

paciente deve ter a disciplina de segui-la. O resultado só é alcançado se ambos fizerem sua parte corretamente.

O desenvolvimento de software possui características semelhantes. Tanto o cliente, quanto os desenvolvedores têm um papel a cumprir. O melhor e mais participativo cliente não será capaz de obter o software desejado se a equipe de desenvolvimento não implementar corretamente o que é pedido e a melhor equipe não será capaz de produzir o software certo se o cliente não for capaz de especificá-lo adequadamente e prover feedback ao longo do projeto. Infelizmente, deficiências na participação do cliente têm sido apontadas como um dos principais fatores a gerar falhas nos projetos de software.

> A falta de envolvimento dos usuários tradicionalmente tem sido a causa número um das falhas nos projetos. No sentido oposto, a contribuição número um para o sucesso de um projeto tem sido envolvimento dos usuários. Mesmo quando entregue no prazo e dentro do orçamento, um projeto pode falhar se não tratar das necessidades e expectativas dos usuários (THE STANDISH GROUP INTERNATIONAL, 2001, p.4, tradução nossa).

O XP se preocupa com esta questão e procura solucioná-la trazendo o cliente para fazer parte da equipe de desenvolvimento. Em termos práticos, isso significa colocar o cliente fisicamente próximo aos desenvolvedores ou mover os desenvolvedores para próximo do cliente.

> Ter especialistas de domínio à disposição o tempo todo significa que o tempo de feedback entre a solução ser imaginada e depois avaliada é o mais curto possível, freqüentemente de minutos ou algumas poucas horas.
> Tal rapidez no feedback significa que a equipe de desenvolvimento ganha uma compreensão mais profunda das necessidades e hábitos dos usuários e comete menos erros quando cria novas idéias. Eles tentam mais idéias diferentes, o que faz com que o produto final fique melhor. Havendo boa dose de colaboração, os programadores irão testar as idéias dos especialistas de domínio e oferecer contra-propostas. Isso irá aperfeiçoar as próprias idéias do cliente sobre como o sistema deve se parecer (COCKBURN, 2002, p.179, tradução nossa).

A presença do cliente ao longo do desenvolvimento viabiliza o ciclo contínuo de feedback entre ele e os desenvolvedores. Este ciclo permite que pequenas mudanças sejam feitas ao longo do desenvolvimento, de forma rápida. É importante lembrar que o tempo de feedback é fundamental. Ou seja, se a equipe recebe feedback do cliente rapidamente, ela aprende com mais precisão a idéia que o cliente está transmitindo e vice-versa. Para que isso funcione, é necessário que haja proximidade física entre clientes e desenvovedores. "Existem muitos estudos que mostram que a comunicação é reduzida enormemente pela separação. Mover colegas um andar abaixo reduz a comunicação quase tanto quanto se os movêssemos para o outro lado do mundo (JEFFRIES, ANDERSON et al., 2001, p.18, tradução nossa)."

A redução da distância entre cliente e desenvolvedores também produz outro efeito benéfico: a melhoria na relação de confiança entre as partes envolvidas. Estar próximo fisicamente permite que o cliente perceba mais facilmente os esforços da equipe de desenvolvimento. Além disso, a redução no tempo para a obtenção de resultados também ajuda a elevar a satisfação do cliente, o que também melhora os relacionamentos (TELES, 2004).

Envolver o cliente na equipe, embora tenha efeitos positivos nos projetos, nem sempre é possível ou fácil. Boehm e Turner (2003), Emam (2003) e Teles (2004) apresentam situações nas quais pode ser difícil ou inviável a introdução desta prática.

5.2.2 Jogo do Planejamento

Como se observou anteriormente, estatísticas demonstram que muitos projetos dedicam esforços significativos à implementação de funcionalidades que não são

utilizadas mais tarde. Portanto, decidir o que implementar é uma das atividades mais importantes a serem conduzidas durante o desenvolvimento de um sistema.

> Tempo é o inimigo de todos os projetos. Visto que o escopo impacta na duração do projeto, ambos estão associados. Minimizando o escopo, o tempo é reduzido e portanto as chances de sucesso crescem (THE STANDISH GROUP INTERNATIONAL, 2001, p.4, tradução nossa).

Os estudos do Standish Group demonstram a importância do conceito de **triagem** para o desenvolvimento de software. Segundo Yourdon (2004), as funcionalidades de um sistema podem ser categorizadas em "**tem que ser feita**", "**deveria ser feita**" e "**poderia ser feita**". Cabe aos membros do projeto assegurar que as funcionalidades que têm que ser feitas sejam implementadas em primeiro lugar.

> Assumindo que a regra familiar "80-20" seja verdadeira, a equipe de projeto pode ser capaz de entregar 80 por centro do "benefício" do sistema implementando 20 por cento dos requisitos – *se* ela implementar os 20 por cento corretos (YOURDON, 2004, p.117, tradução nossa).

O planejamento é usado em XP para assegurar que a equipe esteja sempre trabalhando no mais importante, a cada momento do projeto.

> Não planejamos para prever o futuro. Os negócios e o software mudam rápido demais para que previsões sejam possíveis.
> Planejamos porque:
> - Precisamos assegurar que estamos sempre trabalhando naquilo que é a coisa mais importante que precisamos fazer;
> - Precisamos coordenar com outras pessoas e
> - Quando eventos inesperados acontecem, precisamos compreender as conseqüências para os dois primeiros (BECK & FOWLER, 2001, p.2-3, tradução nossa).

O XP considera o planejamento uma atividade contínua a ser desempenhada ao longo de todo o projeto. O maior valor não está nos planos gerados, mas sim no exercício de criá-los. "Como muitos, nós gostamos da citação de Eisenhower: 'Na preparação para a batalha, descobri que planos são inúteis, mas planejar é indispensável'

(BECK & FOWLER, 2001, p.xiii, tradução nossa).” Por esta razão, os planos e as prioridades são revisados com freqüência.

Projetos XP procuram dividir o tempo disponível para o projeto utilizando dois conceitos: **releases** e **iterações**. “O XP tem releases que tomam alguns poucos meses, que se dividem em iterações de duas semanas, que se dividem em tarefas que tomam alguns dias (BECK & FOWLER, 2001, p.21, tradução nossa).”

O planejamento aloca histórias (fragmentos de funcionalidades) em releases e iterações. Elas são registradas em pequenos cartões, fáceis de serem manipulados pelo cliente e pelos desenvolvedores. “No planejamento do release, o cliente escolhe histórias equivalentes a alguns poucos meses de trabalho, tipicamente se concentrando em um lançamento público (BECK & FOWLER, 2001, p.39, tradução nossa).” Esta idéia é ilustrada na figura 5.1.

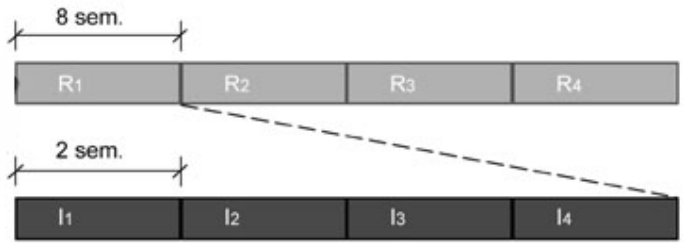

Figura 5.1: release e iterações em um projeto XP.

> Em cada ciclo de release, o cliente controla o escopo, decidindo o que fazer e o que adiar, de modo a prover o melhor release possível na data acertada. O trabalho se encaixa no cronograma baseado no valor para o negócio, dificuldade e a velocidade de implementação da equipe (JEFFRIES, ANDERSON et al., 2001, p.55, tradução nossa).

Um release representa um marco no tempo no qual um conjunto coeso de funcionalidades é finalizado e lançado para consumo de seus usuários. No espaço de tempo de um release (que normalmente é de meses) a equipe implementa

funcionalidades em iterações curtas e fixas que fornecem cadência ao processo de desenvolvimento.

> Uma iteração é um incremento de software útil que é projetado, programado, testado, integrado e entregue durante um espaço de tempo curto e fixo. (...) Este software será aprimorado em iterações futuras, mas se trata de código funcionando, testado e integrado desde o início. Iterações fornecem um aumento dramático de feedback em relação ao desenvolvimento de software seqüencial, portanto, promovendo comunicação muito mais ampla entre clientes e desenvolvedores, e entre as várias pessoas que possuem interesses no sistema. (...) Problemas de design são expostos cedo, e à medida que mudanças acontecem, tolerância a mudanças é construída no sistema (POPPENDIECK & POPPENDIECK, 2003, p.28, tradução nossa).

O final de uma iteração representa um ponto de sincronização no projeto, um momento no qual o cliente e os desenvolvedores avaliam as funcionalidades produzidas e re-avaliam as prioridades para as iterações seguintes. Além disso, permite que eventuais falhas sejam detectadas antes de seguir adiante com o desenvolvimento.

> Que as pessoas cometam erros, em princípio, não nos surpreende. De fato, é exatamente por isso que o desenvolvimento *iterativo* e *incremental* foi inventado. (...)
> A razão para utilizar estratégias incrementais e iterativas é permitir que os inevitáveis erros das pessoas sejam descobertos relativamente cedo e corrigidos de forma apropriada (COCKBURN, 2002, p.47-48, tradução nossa).

As funcionalidades são representadas através de histórias, que refletem necessidades do cliente e são suficientemente pequenas para que os programadores possam implementar um pequeno conjunto delas a cada iteração. “Uma história deve ser compreensível pelo cliente e pelos desenvolvedores, testável, valiosa para o cliente (...) (BECK & FOWLER, 2001, p.45, tradução nossa).”

> A maioria dos métodos ágeis expressa os requisitos em termos de *histórias informais ajustáveis*. Métodos ágeis contam com seus ciclos rápidos de iterações para determinar as mudanças necessárias nas funcionalidades desejadas e para corrigi-las na iteração seguinte. Determinar o conjunto de requisitos mais prioritários a ser incluído na iteração seguinte é feito de forma colaborativa por clientes e

> desenvolvedores. Os clientes expressam suas necessidades mais fortes e os desenvolvedores avaliam que combinação de funcionalidades é viável de ser incluída na iteração seguinte (...). Negociações estabelecem os conteúdos da iteração seguinte (BOEHM & TURNER, 2003, p.37-38, tradução nossa).

Como já foi mencionado anteriormente, o XP atribui aos clientes a responsabilidade de priorizar as histórias e aos desenvolvedores a responsabilidade de estimá-las. A estimativa representa o custo esperado para uma determinada história. Esta informação é importante para que o cliente possa priorizar de maneira adequada. "O valor de negócio depende daquilo que você obtém, quando obtém e quanto custa. Para decidir o que fazer e quando, os clientes precisam saber o custo daquilo que pedem (JEFFRIES, ANDERSON et al., 2001, p.14-15, tradução nossa)."

> O cliente:
> - Define as histórias;
> - Decide qual o valor de negócio de cada história e
> - Decide que histórias serão construídas no release.

> Os programadores:
> - Estimam quanto tempo será necessário para construir cada história;
> - Advertem o cliente sobre riscos técnicos significativos e
> - Medem o progresso da equipe para fornecer um orçamento geral para o cliente (BECK & FOWLER, 2001, p.40, tradução nossa).

As estimativas são geradas com base em experiências passadas dos desenvolvedores. Ou seja, eles observam as histórias que ainda precisam ser implementadas e procuram identificar outras que já tenham sido finalizadas no passado e que sejam semelhantes às histórias que ainda serão desenvolvidas no futuro.

> O melhor guia para estimar o futuro é olhar alguma coisa que aconteceu no passado que seja parecida com a coisa futura. Então simplesmente assuma que a história se repetirá. Freqüentemente isso acontece (BECK & FOWLER, 2001, p.57-58, tradução nossa).

Segundo Beck e Fowler (2001, p.95, tradução nossa), "a única coisa que se sabe sobre um plano é que as coisas não sairão de acordo com ele. Portanto, é preciso

monitorar a iteração em intervalos regulares.” Atrasos, por exemplo, podem ser detectados cedo caso haja um processo permanente de monitoramento, pois o fato é que “o projeto atrasa, um dia de cada vez (BROOKS, 1995, p.154, tradução nossa).”

O progresso da iteração é monitorado diariamente utilizando-se ferramentas como o **quadro de acompanhamento diário** (ver seção 6.13) proposto por Teles (2004). Um dos principais objetivos de seu uso é determinar a **velocidade** da equipe em uma iteração e a carga de trabalho que cada história efetivamente consumiu.

A velocidade representa basicamente a quantidade de trabalho que a equipe é capaz de entregar em uma iteração. No início de uma iteração, o modelo de planejamento do XP assume que a equipe será capaz de entregar a mesma quantidade de histórias efetivamente implementadas na iteração anterior. Se ao final, isso não acontecer, o número é ajustado, passando a refletir mais uma vez o número de histórias que a equipe efetivamente entregou (BECK & FOWLER, 2001).

A velocidade é utilizada como um orçamento que é apresentado ao cliente no início de cada iteração, quando os desenvolvedores declaram, por exemplo, que o cliente poderá alocar até seis histórias na iteração que se inicia. Ao longo do projeto a velocidade tende a convergir rapidamente para um valor que se mantém quase constante durante todo o desenvolvimento. Variações existem, mas costumam ser reduzidas.

O desenvolvimento baseado em iterações curtas é uma forma de elevar as chances de convergir para um sistema que efetivamente atenda às necessidades de seus usuários dentro do tempo disponível para o projeto. Além disso, os processos de re-planejamento baseados em feedback e na velocidade da equipe geram maior previsibilidade sobre os resultados do projeto.

> Uma boa estratégia para atingir convergência é trabalhar nos itens de mais alta prioridade primeiro, deixando os menos prioritários irem

> para baixo na lista de pendências. Entregando funcionalidades de alta prioridade primeiro, é provável que você entregue a maior parte do valor de negócio bem antes de toda a lista ser cumprida (...).
> Esta abordagem para a gerência do projeto pode parecer que irá levar a resultados imprevisíveis, mas na prática é exatamente o oposto que acontece. Uma vez que se estabeleça um histórico de entregas de software funcionando, é fácil projetar a quantidade de trabalho que será feito em cada iteração à medida que o projeto avança (POPPENDIECK & POPPENDIECK, 2003, p.32-33, tradução nossa).

5.2.3 Stand up meeting

Um projeto de desenvolvimento de software pode ser compreendido como um sistema humano composto por elementos tais como clientes, desenvolvedores, gerentes, entre outros. O bom funcionamento do projeto depende do bom funcionamento de cada um de seus componentes, bem como da interação entre os mesmos.

> Um sistema consiste de partes interdependentes que interagem em conjunto para atingir um propósito. Um sistema não é apenas a soma de suas partes – é o produto de suas interações. As melhores partes não necessariamente formam o melhor sistema; a habilidade de um sistema de atingir o seu propósito depende de quão bem as partes trabalham em conjunto, não apenas quão bem atuam individualmente (POPPENDIECK & POPPENDIECK, 2003, p.153, tradução nossa),

Com o objetivo de assegurar que as partes trabalhem bem em conjunto, o XP utiliza uma breve reunião diária chamada de *stand up meeting*[4]. Ela procura alinhar os membros da equipe informando os resultados obtidos no dia anterior e permitindo que os participantes priorizem as atividades do dia que se inicia.

"Um dia de trabalho de uma equipe XP sempre começa com um *stand up meeting*. (...) Primeiramente, ele serve para que todos os membros da equipe comentem rapidamente o trabalho que executaram no dia anterior (TELES, 2004, p.87)." Isso gera visibilidade de tudo o que ocorre na equipe para cada um de seus membros, o que permite identificar rapidamente problemas e soluções que foram encontrados no dia

[4] O termo em inglês significa "reunião em pé". Fazer com que as pessoas permaneçam em pé durante a reunião é uma forma de incentivá-las a concluir a reunião rapidamente.

anterior. É uma forma de disseminar conhecimento e assegurar que as pessoas tenham acesso às informações mais recentes à medida que o projeto prossegue.

Projetos XP procuram assegurar que a equipe trabalhe sempre naquilo que é mais prioritário primeiro. Por isso, existem diversos ciclos de planejamento e o *stand up meeting* é usado para o planejamento diário das atividades da equipe. Além de disseminar informações sobre o dia anterior, a equipe prioriza o trabalho a ser feito no dia que se inicia.

> No fim do stand up meeting, cada membro da equipe sabe o que deverá fazer ao longo do dia. É importante notar que a decisão sobre o que fazer ao longo do dia não é tomada por uma única pessoa. Essa decisão é tomada em equipe. Isso faz sentido, porque quando todos se reúnem, é possível ter uma visão de todo o desenvolvimento e não apenas da uma parte. Desta forma, é factível decidir com mais eficácia quais são as prioridades do dia (TELES, 2004, p.88).

O *stand up meeting* é uma reunião que força uma aproximação dos desenvolvedores de forma diária e contínua. Ele diminui os tempos de feedback, na medida em que cada desenvolvedor reporta as atividades executadas no dia anterior. Isso permite que toda a equipe tenha conhecimento rapidamente dos desafios que foram enfrentados por cada membro, das soluções criadas, das idéias colocadas em prática e dos problemas que precisam ser tratados com urgência.

> Estas reuniões curtas são excelentes para desenvolver o espírito de equipe e comunicar, bem como para determinar quem irá fazer par com quem durante o dia, a manhã ou a tarde (WILLIAMS & KESSLER, 2003, p.5, tradução nossa).

Tal reunião só pode ser feita com freqüência diária, se todos os desenvolvedores estiverem trabalhando próximos uns aos outros e compartilhando um mesmo ambiente. A redução do tempo de feedback dentro da equipe evita que problemas sejam prolongados, visto que o *stand up meeting* também é usado para que a equipe priorize em conjunto as atividades que devem ser executadas a cada dia.

> Percebemos que reuniões diárias curtas são inestimáveis para dar a cada pessoa uma idéia do que os outros estão fazendo. (...) Cada pessoa diz brevemente o que fez no dia anterior e o que está fazendo hoje. Problemas e anúncios importantes para a equipe também são passados. (...) O objetivo do *stand up meeting* é comunicar problemas e não resolvê-los. (...) Tudo que demandar qualquer coisa além de um breve anúncio deve ser reservado para outra sessão onde apenas aqueles interessados no assunto devem estar presentes (BECK & FOWLER, 2001, p.105, tradução nossa).

5.2.4 Programação em Par

Williams e Kessler (2003, p.3, tradução nossa) definem a programação em par como sendo "um estilo de programação no qual *dois* programadores trabalham lado a lado em um computador, continuamente colaborando no mesmo design, algoritmo, código e teste." A programação em par é utilizada por todos os desenvolvedores durante toda a duração de um projeto XP.

Esta técnica implementa uma das diversas redes de proteção que os projetos XP utilizam para reduzir os riscos de eventuais falhas. Quando um programador desenvolve em par, ele conta com a presença de outro desenvolvedor que faz uma inspeção imediata de todo o código que é produzido.

Isso é importante porque "numerosos experimentos confirmaram que o olho tem uma tendência de ver o que ele espera ver (WEINBERG, 1971, p.162, tradução nossa)." Sendo assim, possui uma "capacidade quase infinita de não ver o que não quer ver. (...) Programadores, se deixados com seus próprios aparelhos, irão ignorar os erros mais escandalosos (...) que qualquer outra pessoa consegue ver em um instante (WEINBERG, 1971, p.56, tradução nossa)."

Na medida em que "um único caractere faltando ou incorreto pode consumir literalmente dias para ser encontrado (WILLIAMS & KESSLER, 2003, p.3, tradução

nossa),” a oportunidade de identificar problemas cedo pode significar economia no tempo dedicado à depuração nos projetos e é capaz de reduzir a incidência de bugs.

> Inspeções de código foram introduzidas há mais de 20 anos como uma maneira econômica de detectar e remover defeitos de software. Resultados e estudos empíricos (Fagan, 1976) consistentemente proferem a eficácia das revisões. Entretanto, a maioria dos programadores acha as inspeções desagradáveis e não satisfatórias. (...) A teoria sobre porque as inspeções são eficazes se baseia no conhecimento proeminente de que quanto mais cedo um defeito é encontrado em um produto, mais barato é o seu conserto (WILLIAMS & KESSLER, 2003, p.27, tradução nossa).

A programação em par torna o processo de inspeção parte natural do dia-a-dia da equipe de desenvolvimento. Isso permite que ela consiga utilizar inspeções com freqüência sem incorrer nos problemas que tornam as inspeções tradicionalmente desagradáveis.

> Revisões em pares (também conhecidas como inspeção do código) representam uma das formas mais eficazes para encontrar defeitos em software. As evidências que dão suporte às revisões em pares remontam a mais de duas décadas e ninguém mais questiona seus benefícios. (...)
> (...) apesar das evidências em relação ao valor das inspeções, existem indicações de que inspeções de software tradicionais não são tão prevalentes na prática. A programação em par é uma forma de institucionalizar as revisões em pares dentro dos projetos de software, o que seria uma grande melhoria para a maioria dos projetos (EMAM, 2003, p.12, tradução nossa).

Alguns números ilustram o valor de se identificar defeitos cedo. Watts Humphrey coletou dados da indústria (WILLIAMS & KESSLER, 2003) que demonstram que o tempo normalmente alocado a depuração de um único bug costuma ser elevado, de modo que identificá-lo cedo pode reduzir significativamente os gastos com depuração.

> Tipicamente, durante os testes do sistema leva-se metade (Humphrey, 1995) ou dois (Humphrey, 1997) dias de trabalho para corrigir cada defeito. Dados da indústria relatam que de 33 a 88 horas são gastas em cada defeito encontrado no campo (Humphrey, 1995). Quando cada

> defeito evitado durante o desenvolvimento do código pode poupar um tempo de correção que varia em 0,5 e 88 horas, a programação em par rapidamente se transforma em uma alternativa que poupa tempo e dinheiro (WILLIAMS & KESSLER, 2003, p.39, tradução nossa).

Mesmo quando eventuais problemas aparecem no sistema, a programação em par ajuda a acelerar a depuração. O simples fato de ter uma pessoa ao lado freqüentemente ajuda a encontrar a solução do problema mais rapidamente.

> [Uma] técnica eficaz é explicar o seu código para outra pessoa. Isso normalmente fará com que você explique o bug para si mesmo. (...) Uma outra pessoa irá fazer perguntas e provavelmente fará com que você se explique freqüentemente. As perguntas e respostas podem levar a grandes revelações (WILLIAMS & KESSLER, 2003, p.28, tradução nossa).

Outro aspecto importante da programação em par é o fato freqüentemente observado de que duas pessoas pensando sobre um mesmo problema conseguem explorar mais cenários de soluções e adotar aqueles mais simples e eficazes. Durante o desenvolvimento em par, os programadores mantém um diálogo constante, isto é, "eles discutem novas idéias e abordagens continuamente (WILLIAMS & KESSLER, 2003, p.18, tradução nossa)."

> Os pares consideram muito mais soluções possíveis para um problema e convergem mais rapidamente para a solução que será implementada. De acordo com os pesquisadores Nick Flor e Edwin Hutchins, o feedback, debate e troca de idéias entre parceiros reduzem significativamente a probabilidade de proceder com um design ruim (WILLIAMS, KESSLER et al., 2000, p.20, tradução nossa).

Ao explorar mais cenários e escolher soluções mais simples, os desenvolvedores ajudam a manter o sistema mais simples como um todo. O que ajuda a torná-lo mais fácil de compreender, manter e modificar ao longo do tempo. Além disso, optar por soluções mais simples freqüentemente leva a redução no tempo de desenvolvimento das funcionalidades, o que ajuda a acelerar o progresso da equipe.

A programação em par também é útil no processo de aprendizado dos desenvolvedores, o que é relevante visto que "o negócio da programação se baseia mais do que qualquer outro em aprendizado sem fim (WEINBERG, 1971, p.193, tradução nossa)." Durante o trabalho dos pares, o "conhecimento é passado constantemente entre os parceiros, desde dicas de utilização das ferramentas até regras da linguagem de programação (...) os pares se revezam no papel de professor e aluno (WILLIAMS & KESSLER, 2003, p.29, tradução nossa).

> A revisão contínua que decorre da programação colaborativa cria um mecanismo educacional único porque os parceiros vivenciam um aprendizado sem fim. "O processo de analisar e criticar artefatos de software produzidos por outras pessoas é um método poderoso de aprendizado sobre linguagens, domínios de aplicação e assim por diante" (Johnson, 1998). O aprendizado que transcende estas revisões contínuas previne a ocorrência de defeitos futuros – e a prevenção de defeitos é mais eficaz que qualquer forma de remoção de defeitos. A revisão contínua que decorre da programação colaborativa, na qual cada parceiro trabalha sem cessar para identificar e resolver problemas, gera tanto eficiência ótima em termos de remoção de defeitos, quanto o desenvolvimento de habilidades para a prevenção dos mesmos (WILLIAMS & KESSLER, 2003, p.29, tradução nossa).

O aprendizado através da programação em par é particularmente eficaz "porque não está isolado. Ele está ocorrendo dentro do contexto de um problema maior que você precisa resolver (TELES, 2004, p.95)." Segundo Weinberg (1971, p.195-196, tradução nossa), "nenhum momento será mais propício para aprender do que aquele no qual a *necessidade* do aprendizado é sentida de maneira mais forte (...)."

A existência de um problema, dentro de um contexto bem definido, atua como uma forte motivação para que o programador aprenda algo para que seja capaz de solucionar o problema. A presença imediata de um colega que tenha conhecimentos importantes sobre o problema em questão torna o processo de aprendizado imediato. O problema que gerou a necessidade do aprendizado faz com que o aprendiz dedique

grande atenção e interesse àquilo que está sendo ensinado. Isso eleva a capacidade de absorção do conteúdo, bem como a fixação dos conceitos. Outro fator que exerce influência é a própria contribuição que o programador está dando ao projeto com a solução que busca construir.

> Lave e Wenger (1991) estudaram vários tipos de aprendizagem. Eles enfatizam a importância de que o aprendiz participe ativamente, que ele tenha um trabalho legítimo para fazer, e que ele trabalhe na periferia se movendo consistentemente para um trabalho mais elevado. Lave e Wenger enfatizam a importância de que o aprendiz trabalhe dentro da "linha de visão" do especialista (WILLIAMS & KESSLER, 2003, p.29, tradução nossa).

De forma semelhante, a programação em par "(...) também é uma ótima estratégia de gestão do conhecimento – uma excelente maneira de passar conhecimento tácito pela equipe (WILLIAMS & KESSLER, 2003, p.16, tradução nossa)." Como observamos anteriormente, a disseminação de conhecimento tácito é essencial para o bom andamento de um projeto.

> (...) programação em par funciona para o XP porque encoraja a comunicação. Gosto de pensar na analogia com um copo d'água. Quando uma informação importante é aprendida por alguém na equipe, é como se colocássemos uma gota de tintura na água. Já que os pares se revezam o tempo todo, a informação se difunde rapidamente através da equipe, da mesma forma que a tintura se espalha através da água. Ao contrário da tintura, entretanto, a informação vai ficando mais rica e mais intensa à medida que se espalha e é enriquecida pela experiência e idéias de todas as pessoas da equipe (BECK, 2000, p.101, tradução nossa).

A programação em par também eleva a robustez da equipe, permitindo que ela supere melhor a perda ou a adição de um novo membro. "Com a programação em par, o risco do projeto associado à perda de um programador chave é reduzido porque existem várias pessoas familiarizadas com cada parte do sistema (WILLIAMS & KESSLER, 2003, p.41, tradução nossa)." Além disso, novos desenvolvedores podem começar a contribuir mais cedo.

> Tradicionalmente, pessoas novas em uma organização são apresentadas a diferentes partes de um sistema por uma pessoa experiente da equipe. Este tempo dedicado ao treinamento custa diversas horas valiosas do membro experiente. Durante estas horas, nem a pessoa nova, nem o treinador estão contribuindo no sentido de completar o projeto (WILLIAMS & KESSLER, 2003, p.42, tradução nossa).

Em uma equipe XP, o novo membro irá trabalhar em par com diversas pessoas da equipe, em várias partes do sistema. Nestes momentos, a pessoa que estiver atuando como seu mentor irá produzir mais lentamente, porém continuará produzindo. O novo programador, por sua vez, tenderá a aprender os conceitos mais rapidamente devido às razões explicadas anteriormente sobre a eficácia da programação em par para o processo de aprendizado. Além disso, a programação em par ajuda a assegurar que o novo membro aprenda rapidamente o método de trabalho da equipe.

> De um modo geral, nós nos comportamos da mesma forma que vemos as pessoas se comportarem a nossa volta, portanto, um grupo de programação funcionando tenderá a socializar novos membros para sua filosofia de programação (WEINBERG, 1971, p.61, tradução nossa).

A programação em par também tende a gerar maior aderência ao processo de desenvolvimento escolhido pela equipe. Isso vale tanto para desenvolvedores novos, quanto para aqueles que já estão na equipe há mais tempo. Em XP, isso é particularmente relevante, pois embora seja um processo de desenvolvimento simples e com pouca formalidade, é um processo significativamente disciplinado (COCKBURN, 2002).

> Outra característica poderosa da programação em par é que algumas das práticas não funcionariam sem ela. Sob estresse, as pessoas recuam. Elas deixariam de escrever os testes, não iriam refatorar[5] e

[5] Do inglês *refactor*. A palavra não existe em português, porém alguns livros de XP recentes trazem a expressão refatorar como um neologismo, que é adotado nesta dissertação para facilitar a leitura, evitar o uso do termo em inglês e se alinhar às demais publicações que já utilizam o termo.

> evitariam integrar. Com o seu parceiro olhando, contudo, existem boas chances de que mesmo que você esteja ignorando uma destas práticas, o seu parceiro não estará. (...) as chances de ignorar o seu compromisso com o restante da equipe é bem menor trabalhando em par que se você estivesse trabalhando sozinho (BECK, 2000, p.102, tradução nossa).

Um efeito conhecido como **pressão do par** costuma estar presente durante a programação em par e exerce influência sobre a produtividade dos desenvolvedores. A responsabilidade compartilhada com outra pessoa faz com que os programadores se concentrem mais e melhor naquilo que estão produzindo. "Os programadores admitem trabalhar com mais afinco e de forma mais inteligente nos programas porque não querem decepcionar seus parceiros (WILLIAMS & KESSLER, 2003, p.21, tradução nossa)."

> (...) ter um parceiro trabalhando com você durante a maior parte do dia minimiza distrações e interrupções. Um desenvolvedor dificilmente irá bater papo sobre questões de lazer ao telefone enquanto uma pessoa estiver sentada ao seu lado esperando que termine. Da mesma forma, alguém dificilmente irá interromper duas pessoas no meio de uma discussão. Portanto, a quantidade de tempo não produtivo que é recuperado pode levar a um aumento de produtividade (EMAM, 2003, p.12, tradução nossa).

Além de elevar a concentração e reduzir interrupções por terceiros, a pressão do par também evita os bloqueios mentais e distrações "seja porque um parceiro mantém o outro na trilha certa (e não é distraído) ou porque um dos parceiros consegue ultrapassar o bloqueio mental (WILLIAMS & KESSLER, 2003, p.18-19, tradução nossa)." Isso também ajuda a recuperar um tempo que, de outra forma, seria improdutivo.

Embora a programação em par ofereça oportunidades de melhoria nos projetos, "a reação intuitiva de muitas pessoas é rejeitar a idéia (...) porque assumem que haverá um aumento de cem por cento de programador-hora colocando dois programadores para

fazer o trabalho que um pode fazer (WILLIAMS & KESSLER, 2003, p.38, tradução nossa).” Entretanto, pesquisas demonstram que isso não ocorre na prática.

Williams e Kessler (2003) executaram experimentos, com resultados estatisticamente significativos, demonstrando que a programação em par eleva o número de programador-hora em apenas 15 por cento. Entretanto, conseguiram “validar estatisticamente os relatos que alegam que a programação em par é um meio acessível de produzir software de mais elevada qualidade (WILLIAMS & KESSLER, 2003, p.9, tradução nossa).” Ou seja, apesar da ligeira elevação no consumo de recursos, os resultados mostraram que os desenvolvedores que trabalharam em par geraram um número significativamente menor de defeitos, produziram menos código para solucionar o mesmo problema e tiveram resultados mais consistentes (WILLIAMS, KESSLER et al., 2000).

> Trabalhando em sintonia, os pares completaram suas atribuições de 40% a 50% mais rapidamente.
> No mercado atual, obter um produto de qualidade tão rapidamente quanto possível é uma vantagem competitiva que pode até significar a própria sobrevivência. Consertar defeitos depois de lançar os produtos para os clientes pode custar muito mais que encontrá-los e consertá-los durante o desenvolvimento. Os benefícios de ter um produto lançado mais rapidamente, com gastos menores de manutenção e que melhora a satisfação dos clientes supera qualquer aumento na quantidade de programador-hora que possa resultar dos pares (WILLIAMS, KESSLER et al., 2000, p.22-24, tradução nossa).

Finalmente, a programação em par também ajuda a elevar a motivação dos desenvolvedores, tornando o trabalho mais divertido e facilitando a comunicação. “Em nossa recente pesquisa pela *Web*, perguntamos, ‘Que benefícios você encontrou na programação em par?‘ A resposta mais comum foi: ‘É muito mais divertido’ (WILLIAMS & KESSLER, 2003, p.21, tradução nossa).” Naturalmente, quanto mais

agradável e divertido for o trabalho de programação, mais produtivo o mesmo tenderá a ser.

> Na pesquisa on-line com programadores profissionais que trabalham em par, 96% afirmaram que gostavam mais do trabalho em relação a quando programavam sozinhos. Nós entrevistamos seis vezes os 41 programadores que trabalharam de forma colaborativa em pares no experimento da universidade. Consistentemente, mais de 90% deles afirmaram que gostavam mais de trabalhar em par que sozinhos. Adicionalmente, virtualmente todos os programadores profissionais pesquisados afirmaram que ficavam mais confiantes em suas soluções quando programavam em par. Quase 95% dos estudantes ecoaram esta afirmação.
> Existe uma correlação natural entre estas duas medidas de satisfação. Isto é, os pares gostavam mais se seu trabalho porque ficavam mais confiantes com o resultado final (WILLIAMS, KESSLER et al., 2000, p.24, tradução nossa).

O trabalho diário executado com diferentes pessoas freqüentemente ajuda a aproximá-las e a quebrar barreiras de comunicação. Isso envolve inclusive a aproximação no nível pessoal que também gera benefícios ao longo do desenvolvimento.

> À medida que os membros da equipe se conhecem melhor, eles se tornam muito mais propensos a falar entre eles sobre questões pessoais e técnicas. As barreiras de comunicação entre cada pessoa começam a cair. As pessoas passam a considerar as outras muito mais acessíveis. (...) O relacionamento e a confiança construído entre os membros da equipe geram em cada um a coragem de pedir conselhos e orientação sem se sentirem vulneráveis e insuficientes. Além disso, eles se sentem melhor sobre o trabalho que fazem porque conhecem seus companheiros de equipe no nível pessoal (WILLIAMS & KESSLER, 2003, p.43, tradução nossa).

Apesar de seus benefícios, a programação em par pode se revelar problemática em pelo menos duas circunstâncias: a presença de programadores com ego excessivamente elevado e competição entre os membros da equipe.

> Um programador que genuinamente veja seu programa como uma extensão do seu próprio ego não irá tentar encontrar todos os erros naquele programa. Ao contrário, tentará provar que o programa está correto – mesmo que isso signifique fazer vista grossa para erros que

> são monstruosos para outros olhos (WEINBERG, 1971, p.55, tradução nossa).

Além disso, fazer par com uma pessoa de ego excessivamente elevado costuma ser difícil e propenso a conflitos. Por sua vez, programar em par em ambientes competitivos é pouco viável porque o processo de ensino e aprendizado fica bloqueado em função da competição.

> O ato de ensinar simplesmente não pode ocorrer se as pessoas não se sentirem seguras. Em uma atmosfera de competição, você seria maluco se deixasse alguém ver você aprendendo com outra pessoa; seria uma clara indicação de que você conhece menos que o seu mentor sobre um determinado assunto. Da mesma forma, você seria louco de ensinar à outra pessoa, já que esta pessoa poderia eventualmente usar seus ensinamentos para passar a sua frente (DEMARCO & LISTER, 1999, p.182-183, tradução nossa).

5.2.5 Código Coletivo

Complementando a programação em par, equipes XP também praticam o conceito de código coletivo: "todas as classes e métodos pertencem à equipe e qualquer membro da equipe pode melhorar o que for necessário (JEFFRIES, ANDERSON et al., 2001, p.122, tradução nossa)." O desenvolvedor não apenas programa em par com diferentes pessoas ao longo do tempo, como também tem acesso a todas as partes do código, inclusive aquelas que ele não programou. Além disso, tem o direito de fazer qualquer alteração que considerar necessária sem ter que pedir permissão. Naturalmente, seus colegas também podem alterar o seu código sem lhe pedir (BECK, 2000).

Isso significa que os desenvolvedores têm a oportunidade de trabalhar com pessoas diferentes e em partes diferentes do código. Existe um revezamento diário em ambos os aspectos. Assim se estabelece mais uma rede de proteção, visto que os programadores podem revisar e alterar o código escrito em diferentes partes do sistema,

por diferentes pessoas. Como alterações no código podem causar erros, a prática de código coletivo só pode ser adotada com segurança se a equipe investir em testes automatizados. “Código coletivo é aquela idéia aparentemente maluca de que qualquer pessoa pode alterar qualquer fragmento de código no sistema a qualquer momento. Sem os testes, você estaria morto tentando fazer isso (BECK, 2000, p.99-100, tradução nossa).”

Esta prática também protege o projeto ajudando a tornar a equipe mais robusta, na medida em que os desenvolvedores se habituam a trabalhar nas mais variadas partes do sistema. Sendo assim, “todo o trabalho fica menos suscetível de ser afetado se alguém ficar doente (...) ou faltar (...). Isso não apenas reduz as variações no cronograma, mas também eleva as chances de ter alguém por perto quando o código tiver que ser modificado (WEINBERG, 1971, p.59-60, tradução nossa).”

A adoção do código coletivo também permite que a equipe avance mais rapidamente “porque ninguém precisa esperar até que outra pessoa apareça para consertar alguma coisa. O código permanece mais limpo porque os programadores não são forçados a contornar uma deficiência em um objeto criando um outro (JEFFRIES, ANDERSON et al., 2001, p.122, tradução nossa).” Além disso, a equipe tem razões genuínas para manter o código simples ao longo do projeto.

> Um dos efeitos do código coletivo é que código complexo não sobrevive por muito tempo. Visto que todos estão acostumados a olhar todas as partes do sistema, tal código será encontrado logo, ao invés de tardiamente. Quando for encontrado, alguém irá tentar simplificá-lo. (...)
> A propriedade coletiva do código tem a tendência de evitar que códigos complexos entrem no sistema em primeiro lugar. Se você sabe que outra pessoa irá olhar para o que você está escrevendo dentro de pouco tempo (em algumas horas), você irá pensar duas vezes antes de colocar no sistema uma complexidade que você não conseguirá justificar (BECK, 2000, p.99-100, tradução nossa).

5.2.6 Código Padronizado

Em projetos XP, os programadores codificam seguindo um padrão de código acordado. “Não importa muito o formato. O que realmente importa é que todos os membros da equipe adotem o padrão e o utilizem sempre. (...) não são as especificidades (...) que contam; é a sua familiaridade com elas (JEFFRIES, ANDERSON et al., 2001, p.79, tradução nossa).

> Padrões de código (...) são importantes porque levam a uma maior consistência dentro do seu código e o código de seus colegas de equipe. Mais consistência leva a um código mais simples de compreender, o que por sua vez significa que é mais simples desenvolver e manter (AMBLER, 2000, p.1, tradução nossa).

A adoção de um padrão ajuda a simplificar a comunicação, a programar em par e a tornar o código coletivo. Da mesma forma, a própria programação em par e a prática de código coletivo ajudam a assegurar que a equipe irá seguir o padrão adotado. Como o código passa por vários níveis de revisão, é fácil detectar e corrigir qualquer código fora do padrão. Porém para assegurar que o padrão realmente seja usado, “qualquer padrão envolvido deve surgir e evoluir no nível das próprias pessoas que realizam o trabalho. A propriedade do padrão deve estar nas mãos daqueles que realizam o trabalho (DEMARCO, 2001, p.109, tradução nossa).”

5.2.7 Design Simples

Existem pelo menos quatro coisas que os usuários de um software esperam dele:

- Que faça a coisa certa;
- Que funcione;
- Que seja fácil de utilizar e
- Que possa evoluir com o tempo.

As práticas do XP se complementam formando um conjunto que busca atingir estes objetivos. Para assegurar que o sistema esteja fazendo a coisa certa e esteja funcionando, o desenvolvimento é executado em iterações curtas nas quais se implementa um pequeno conjunto de histórias. O feedback gerado ao final de cada iteração permite que os usuários avaliem se o sistema realmente está fazendo a coisa certa e se está funcionando.

Adotar iterações curtas, portanto, é um mecanismo importante para atingir parte destes objetivos, entretanto, impõe um desafio. Como fazer a análise, design, implementação, teste e depuração de um conjunto de funcionalidades em uma ou duas semanas? Isso só é possível se os desenvolvedores mantiverem o design da aplicação suficientemente simples (BECK, 2000).

A facilidade de uso, por sua vez, é conquistada quando se consegue desenvolver um software que tenha **integridade perceptível** e **integridade conceitual**. "Um programa limpo e elegante tem que apresentar a cada um de seus usuários um modelo mental coerente (...). A integridade conceitual, como percebida pelo usuário, é o fator mais importante na facilidade de uso (BROOKS, 1995, p.255-256, tradução nossa)."

> Integridade conceitual significa que os conceitos centrais de um sistema operam em conjunto como um todo equilibrado e coeso. Os componentes se encaixam e funcionam bem em conjunto; a arquitetura alcança um equilíbrio eficaz entre flexibilidade, manutenibilidade, eficiência e presteza. A arquitetura de um sistema de software se refere à forma como o sistema é estruturado para prover as funcionalidades e características desejadas. Uma arquitetura eficaz é o que leva um sistema a ter integridade conceitual (POPPENDIECK & POPPENDIECK, 2003, p.135, tradução nossa).

Criar uma arquitetura de software que possua integridade conceitual é um feito importante que pode ser ameaçado caso o sistema sofra alterações freqüentes. "Algumas mudanças válidas de objetivos (e de estratégia de desenvolvimento) são inevitáveis e é

melhor estar preparado para elas que assumir que elas não virão (BROOKS, 1995, p.241, tradução nossa).” Portanto, embora o desenvolvimento iterativo seja útil para alcançar alguns dos objetivos do projeto, parece ser uma proposta arriscada quando se deseja construir um sistema fácil de ser utilizado e que também possa evoluir ao longo do tempo. Na prática, entretanto, as iterações provêm oportunidades para que a arquitetura seja revisada, aprimorada e amadureça com o tempo.

> (...) clientes de um sistema de software geralmente não são capazes de definir o que irão perceber como sendo íntegro, assim como eles não sabem descrever de forma precisa o que desejam em um carro. Os clientes sabem quais são seus problemas, mas com bastante freqüência, não conseguem descrever a solução. Eles saberão reconhecer um bom sistema quando o encontrarem, mas eles não conseguem visualizá-lo previamente. Para piorar as coisas, à medida que suas circunstâncias mudarem, também mudarão suas percepções sobre a integridade do sistema (POPPENDIECK & POPPENDIECK, 2003, p.130-131, tradução nossa).

A arquitetura de um sistema precisa ser capaz de evoluir porque a necessidade de mudanças é a única constante encontrada no desenvolvimento, visto que o software “se encontra inserido em uma matriz cultural de aplicações, usuários, leis e equipamentos de hardware (...) [que] muda continuamente e (...) força mudanças no software (BROOKS, 1995, p.184, tradução nossa).”

> À medida que novas funcionalidades são adicionadas a um sistema para manter a integridade perceptível, as características subjacentes da arquitetura para acomodar as funcionalidades de maneira coesa também precisam ser modificadas (POPPENDIECK & POPPENDIECK, 2003, p.128-129, tradução nossa).

“Dados da indústria indicam que entre 50 e 70 por cento de todo o esforço gasto em um programa serão gastos depois que o mesmo for entregue para o usuário pela primeira vez (PRESSMAN, 1997, p.18, tradução nossa).” Isso representa uma razão adicional para que a equipe de desenvolvimento encontre mecanismos para facilitar mudanças futuras. Uma forma de fazer isso é tentar prever o que precisará mudar e

tornar o sistema mais genérico de modo a se ajustar mais facilmente a tais mudanças previstas. Entretanto, generalizações freqüentemente elevam a complexidade do sistema e muitas vezes as previsões não se materializam. Outra forma de resolver essa questão é utilizar o feedback das iterações para detectar as generalizações necessárias e tornar a arquitetura naturalmente tolerante a mudanças.

> A maior parte dos sistemas de software é dinâmica ao longo do tempo; bem mais da metade dos gastos em uma aplicação ocorrerão depois de ela entrar em produção. Os sistemas devem ser capazes de se adaptar a mudanças tanto de negócio, quanto técnicas de forma econômica.
> Uma das abordagens chave para incorporar mudanças em uma infra-estrutura de informação é assegurar que o próprio processo de desenvolvimento incorpore mudanças continuamente. (...) Um processo de design tolerante a mudanças tem mais chances de criar como resultado um sistema tolerante a mudanças (POPPENDIECK & POPPENDIECK, 2003, p.134, tradução nossa).

Equipes XP procuram assegurar que o design seja mantido simples e eventuais flexibilidades sejam adiadas até que se tornem efetivamente necessárias em função das histórias solicitadas pelos usuários. Portanto, os desenvolvedores não tentam prever mudanças futuras, nem procuram antecipar as flexibilidades que serão necessárias. Eles esperam o feedback das iterações para receber informação concreta sobre onde a arquitetura precisa ser sofisticada.

> Observar onde as mudanças ocorrem ao longo do desenvolvimento iterativo dá uma boa indicação de onde o sistema provavelmente precisará de flexibilidade no futuro. (...)
> Se um sistema é desenvolvido permitindo que o design vá emergindo através de iterações, o design será robusto, adaptando-se mais prontamente para os tipos de mudança que ocorrem ao longo do desenvolvimento. Ainda mais importante, a habilidade de se adaptar será construída dentro do sistema de modo que à medida que mais mudanças ocorram após o seu lançamento, elas possam ser prontamente incorporadas. (POPPENDIECK & POPPENDIECK, 2003, p.52, tradução nossa).

Temendo a incapacidade de adaptar o sistema no futuro, diversas equipes de desenvolvimento criam soluções desnecessariamente complexas para problemas que possivelmente não irão nem aparecer no futuro. Nestes casos, os desenvolvedores enveredam em um trabalho especulativo que consome recursos do projeto, eleva a complexidade do código e o potencial de ocorrência de erros. Além disso, "um design complicado é mais difícil de comunicar que um simples. Devemos, portanto, criar uma estratégia de design que gere o design mais simples possível, consistente com nossos demais objetivos (BECK, 2000, p.103, tradução nossa)."

> (...) cada programa tem um nível apropriado de cuidado e sofisticação dependendo dos usos para os quais será colocado em prática. Trabalhar acima deste nível é, em certo sentido, ainda menos profissional que trabalhar abaixo. (...)
> Freqüentemente, contudo, o programador não consegue ajustar suas atividades para o problema que tem nas mãos porque ele não sabe exatamente que problema tem nas mãos. Isto é, ele *assume* que certas coisas são desejadas – talvez com base naquilo que ele sabe fazer, ou com base naquilo que foi desejado no último trabalho que ele programou – mas ele nunca descobre o que era desejado até o trabalho ser finalizado (WEINBERG, 1971, p.127, tradução nossa).

As equipes XP procuram adiar decisões de design tanto quanto possível. Ou seja, implementam o design mais simples e comunicativo possível para os problemas de hoje. Sofisticações que possam ser necessárias para acomodar necessidades futuras não são implementadas até que se atinja um ponto no desenvolvimento no qual a equipe irá implementar de fato as histórias que demandem tais sofisticações. Desta forma, se reduz as chances de adicionar código desnecessário e elevar a complexidade do sistema cedo demais (POPPENDIECK & POPPENDIECK, 2003). O problema desta abordagem é que algumas alterações podem acabar se revelando custosas no futuro.

> Em 1987, Barry Boehm escreveu "Encontrar e corrigir um problema de software após a entrega custa 100 vezes mais que encontrar e corrigir o problema em fases iniciais de design." Esta observação se tornou a lógica por traz de colocar a análise de requisitos e o design no início do projeto, embora o próprio Boehm encorajasse o

> desenvolvimento incremental. Em 2001, Boehm notou que para sistemas pequenos o fator de escalada pode ser algo mais próximo de 5:1 do que 100:1; e mesmo em sistemas grandes, boas práticas arquiteturais podem reduzir significativamente o custo de mudança confinando funcionalidades que são prováveis de sofrerem alterações em áreas pequenas e bem encapsuladas (POPPENDIECK & POPPENDIECK, 2003, p.50, tradução nossa).

Se o custo de alterar o software se elevar exponencialmente ao longo do tempo, a abordagem iterativa pode ser arriscada e muito custosa. Entretanto, se for possível mantê-lo baixo, tal abordagem pode ser incorporada nos projetos de software com resultados positivos.

> Nem todas as mudanças são iguais. Existem algumas poucas decisões arquiteturais básicas nas quais você precisa acertar no começo do desenvolvimento, porque elas fixam restrições do sistema para toda a vida do mesmo. Exemplos podem ser a escolha da linguagem, decisões de camadas a serem usadas na arquitetura ou a escolha de interagir com um banco de dados existente também utilizado por outras aplicações. Estes tipos de decisões podem ter uma taxa de escalada no custo de 100:1. (...)
> A maior parte das mudanças em um sistema não tem necessariamente que ter um alto fator de escalada do custo. (...)
> Uma única curva ou fator de escalada do custo é enganoso. Ao invés de um único gráfico mostrando uma única tendência para todas as mudanças, um gráfico mais apropriado tem ao menos duas curvas da escalada do custo, como mostrado na figura [5.2] (POPPENDIECK & POPPENDIECK, 2003, p.49-51, tradução nossa).

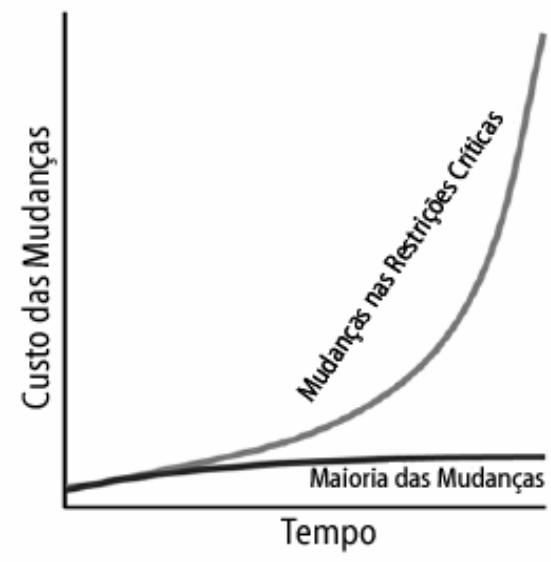

Figura 5.2: custo de uma mudança em um software ao longo do tempo.

Equipes XP se baseiam na idéia de que "em determinadas circunstâncias, o crescimento exponencial no custo de mudança do software ao longo do tempo pode se tornar linear (BECK, 2000, p.21, tradução nossa)." Tal idéia se fundamenta, entre outros, nos seguintes fatores:

- Avanços ocorridos na microinformática;
- A adoção da orientação a objetos;
- O uso da refatoração para aprimorar e simplificar o design; e
- A adoção de testes automatizados.

> A revolução do microcomputador mudou a forma como se constrói software. (...) Computadores individuais velozes agora são ferramentas rotineiras do desenvolvedor de software (...). O computador pessoal de hoje em dia não é apenas mais veloz que o supercomputador de 1960, é mais rápido que as estações Unix de 1985. Tudo isso significa que compilar é rápido até mesmo na máquina mais humilde, e grandes quantidades de memória eliminaram esperas por montagens baseadas em acessos a disco. Grandes quantidades de memória também tornam razoável manter tabelas de símbolos na memória com o código objeto, de modo que depuração em alto nível sem re-compilação se tornou rotina (BROOKS, 1995, p.281-282, tradução nossa).

Computadores velozes permitem, entre outras coisas, executar testes automatizados em poucos segundos cada vez que uma nova funcionalidade ou modificação é inserida no sistema. Isso significa que é possível obter feedback mais rapidamente e detectar falhas mais cedo. Além disso, inúmeras tecnologias importantes passaram a ser utilizadas nas últimas décadas na tentativa de reduzir o custo da mudança, tais como a orientação a objetos, "melhores linguagens, melhor tecnologia de banco de dados, melhores práticas de programação, melhores ambientes e ferramentas, bem como novas notações (BECK, 2000, p.23, tradução nossa)."

Programadores de equipes XP desenvolvem o hábito de refatorar o código permanentemente, com o objetivo de mantê-lo simples ao longo do tempo. As oportunidades de refatoração são identificadas utilizando-se as revisões freqüentes que são causadas pelas práticas de programação em par e código coletivo. Finalmente os testes automatizados ajudam a equipe a identificar rapidamente se uma mudança no sistema causou algum tipo de erro. Quanto mais rápido o erro é identificado, menor é o tempo de depuração e menor é o custo associado à mudança. Portanto, o XP se baseia na idéia de que a curva de mudança se mantém constante ao longo do tempo e a adoção de suas práticas colabora para que essa idéia seja verdadeira (BECK, 2000).

> (...) quando o código é "suficientemente simples," tudo o que escrevemos deve:
> 1. Passar em todos os testes;
> 2. Expressar todas as idéias que temos que expressar;
> 3. Dizer tudo uma vez e apenas uma vez; e
> 4. Ter o número mínimo de classes e métodos que seja consistente com as recomendações acima (JEFFRIES, ANDERSON et al., 2001, p.83, tradução nossa).

5.2.8 Desenvolvimento Orientado a Testes

Sistemas computacionais e projetos de software costumam vivenciar diversas dificuldades ao longo do tempo. Um dos problemas mais caros e recorrentes é a incidência de defeitos.

> De acordo com a mais recente estimativa do Departamento de Comércio americano, os softwares defeituosos custam 60 bilhões de dólares por ano só à economia americana. No Brasil, não há dados confiáveis, mas especialistas acreditam que uns 8 bilhões de reais – ou cerca de 0,6% do PIB – não seria um número muito distante da realidade.(...)
> Há quatro anos a *Mars Climate Orbiter* foi perdida quando entrava na atmosfera de Marte. Mal programado, o software embutido na sonda misturou medidas em pés com metros e, por um problema tão simples, provocou um prejuízo de 125 milhões de dólares. (TEIXEIRA, 2004).

O problema ocorrido no software da *Mars Climate Orbiter*, que é relatado com maiores detalhes em (ISBELL, HARDIN et al., 1999) e (SORID, 1999), é um exemplo

de quão custosas podem ser as conseqüências dos defeitos em sistemas computacionais. Embora tais conseqüências normalmente sejam mais críticas depois que o sistema já se encontra em produção, elas também têm efeitos danosos no dia-a-dia dos projetos.

Quando um defeito é identificado, faz-se necessário depurar o software. "Depuração é a parte difícil e lenta da programação de sistemas, e ciclos [de feedback] demorados representam o pior problema da depuração (BROOKS, 1995, p.245, tradução nossa)." Ela é demorada porque os programadores "se esquecem do que fizeram. Se escrevo um programa hoje e o testo em uma semana, minhas recordações sobre como o escrevi terão se perdido (...). Portanto (...) não serei capaz de imaginar rapidamente onde está o problema (JEFFRIES, ANDERSON et al., 2001, p.32, tradução nossa)."

> Se você observar como os programadores passam o tempo, irá descobrir que codificar na verdade representa uma pequena fração. Algum tempo é gasto tentando descobrir o que precisa ser feito, algum tempo é gasto projetando, mas a maior parte do tempo é gasta depurando. (...) Consertar o bug normalmente é rápido, mas descobrir o que o está causando é um pesadelo. Então, quando você corrige um bug, sempre existe a chance de que outro apareça e que você não venha a notar até que já tenha se passado bastante tempo (FOWLER, 2000, p.89, tradução nossa).

Por isso, é essencial que as equipes de desenvolvimento sejam capazes de reduzir a incidência de bugs e o custo associado à depuração e eliminação dos mesmos. Isso é válido durante o projeto, porém é ainda mais relevante durante a manutenção do sistema.

> O problema fundamental com a manutenção de programas é que consertar um defeito tem uma chance substancial (20-50 por cento) de introduzir outro. Assim, o processo todo é dois passos para frente e um para trás.
> Por que os defeitos não são corrigidos de forma mais limpa? Primeiro, mesmo um defeito sutil se revela como uma falha local de algum tipo. De fato, ele freqüentemente possui ramificações ao redor do sistema, normalmente nada óbvias. Qualquer tentativa de repará-lo com o mínimo de esforço irá corrigir a parte local e óbvia, mas os efeitos do

> reparo que alcançam outras partes do sistema serão ignorados. Segundo, aquele que faz o reparo normalmente não é a pessoa que escreveu o código, e freqüentemente é um programador júnior ou um programador em treinamento (BROOKS, 1995, p.122, tradução nossa).

Equipes XP lidam com estes problemas utilizando uma técnica conhecida como **desenvolvimento orientado a testes**. Ela consiste em escrever um mecanismo de teste automatizado antes de codificar cada história e cada método do sistema (BECK, 2000). Trata-se de uma técnica preventiva utilizada durante todo o projeto e a manutenção. A prevenção é usada porque "(...) antes da questão de como resolver problemas vem a questão de *evitar* problemas. (...) um programador que evita problemas é mais 'inteligente' que aquele que traz problemas para si mesmo (WEINBERG, 1971, p.164-165, tradução nossa)."

> Ao invés de esperar até o final [do desenvolvimento], é muito mais barato, no longo prazo, adotar um modelo do tipo "pague à medida que for usando." Escrevendo testes com o código à medida que você avança, não há nenhuma correria no final e você experimenta menos bugs ao longo do projeto na medida em que está sempre trabalhando com código testado. Reservando um pouquinho mais de tempo o tempo todo, você minimiza o risco de necessitar de uma quantidade enorme de tempo no final (HUNT & THOMAS, 2003, p.9, tradução nossa).

Os testes automatizados procuram comprovar que as solicitações dos usuários estão sendo atendidas de forma correta. "De tudo o que podemos esperar de um programa, primeiro e mais importante, é que ele esteja correto. Em outras palavras, deveria gerar as saídas corretas para cada entrada possível (WEINBERG, 1971, p.17, tradução nossa)." Além disso, os testes verificam se as histórias continuam funcionando ao longo do tempo, pois "(...) além de assegurar que o código faça o que você quer, você precisa assegurar que o código faça o que você quer o tempo todo (HUNT & THOMAS, 2003, p.5, tradução nossa)."

> Testes são mais valiosos quando o nível de estresse se eleva, quando as pessoas estão trabalhando muito, quando o julgamento humano começa a falhar. Portanto, os testes têm que ser automatizados – retornando uma indicação sobre o funcionamento do sistema do tipo "polegar para cima" ou "polegar para baixo" (BECK, 2000, p.116, tradução nossa).

Os desenvolvedores se concentram em dois aspectos durante a programação. O código deve se manter limpo e precisa funcionar.

> O objetivo é código limpo que funcione por uma série de razões:
> - Código limpo que funcione é uma forma previsível de desenvolver. Você sabe quando terminou sem ter que se preocupar com uma longa lista de bugs;
> - Código limpo que funcione lhe dá a chance de aprender todas as lições que o código tem para ensinar (...);
> - Código limpo que funcione melhora a vida dos usuários do nosso software;
> - Código limpo que funcione permite que seus colegas de equipe possam confiar em você e que você possa confiar neles e
> - Escrever um código limpo que funcione faz com que o desenvolvedor se sinta melhor (BECK, 2003, p.viii, tradução nossa).

O XP se concentra sobretudo em dois tipos de testes: o **teste de unidade** e o **teste de aceitação**. O primeiro tenta assegurar que cada componente do sistema funcione corretamente. O segundo procura testar a interação entre os componentes, as quais dão origem às funcionalidades (BECK, 2000).

Os testes de unidade são escritos antes de cada método produzido no sistema. É a forma utilizada pelos desenvolvedores para saber se o método irá funcionar ou não. "Quando um desenvolvedor codifica, ele deve obter feedback imediato para saber se o código funciona como desejado. Em outras palavras, deve haver um teste para cada mecanismo que o desenvolvedor implementar (POPPENDIECK & POPPENDIECK, 2003, p.146, tradução nossa)."

Feedback imediato é importante porque evita que os defeitos passem muito tempo despercebidos. Como se observou anteriormente, depurar costuma ser custoso porque o desenvolvedor esquece *o que fez*, *por que fez* e *o contexto* em torno da funcionalidade que se encontra defeituosa. Ao depurar tudo isso precisa ser recordado. Sendo assim, quanto mais tempo se passa entre o momento em que o erro é introduzido e o momento em que é identificado, maior tende a ser o tempo de depuração (TELES, 2004). O feedback imediato gerado pelos testes de unidade ajuda a reduzir este tempo e, portanto, freqüentemente consegue acelerar a depuração. Mas, isso tudo só é possível se os testes forem automatizados.

> Se os testes não forem automatizados ou se tomarem muito tempo, eles não serão executados com suficiente freqüência. Grandes lotes de mudanças serão implementados antes de testar, o que tornará muito mais provável a ocorrência de falhas e será bem mais difícil identificar que mudança fez com que os testes falhassem (POPPENDIECK & POPPENDIECK, 2003, p.147, tradução nossa).

Os testes de aceitação (também conhecidos como **testes funcionais**) "são escritos para assegurar que o sistema como um todo funciona. (...) Eles tipicamente tratam o sistema inteiro como uma caixa preta tanto quanto possível (FOWLER, 2000, p.97, tradução nossa)." Estes testes são escritos pelo cliente ou com a orientação do cliente, pois ele é a pessoa que conhece o negócio e, portanto, os resultados que devem ser produzidos pelo sistema.

> Os clientes escrevem teste para uma história de cada vez. A pergunta que eles precisam se fazer é: "O que necessitaria ser verificado antes que eu tivesse a confiança de que esta história está finalizada?" Cada cenário que eles imaginam se transforma em um teste, neste caso, um teste funcional (BECK, 2000, p.118, tradução nossa).

Como os clientes normalmente não são capazes de escrever e automatizar os testes de aceitação por conta própria, "uma equipe XP de qualquer tamanho carrega

consigo pelo menos um analista de teste dedicado. O trabalho dele é traduzir as idéias de testes, às vezes vagas, do cliente em testes reais, automatizados e isolados (BECK, 2000, p.118-119, tradução nossa)."

Por maiores que sejam os investimentos em testes unitários e funcionais, equipes XP eventualmente encontram bugs que não são detectados pelos testes. Nestes casos, os desenvolvedores criam testes que exponham os defeitos antes de corrigi-los. Desta forma, se protegem do caso em que um mesmo bug volte a ocorrer no futuro. Se ocorrer, será identificado rapidamente.

> Como boa prática, o programa de teste deveria ser construído *antes* que a "correção" fosse feita no programa de produção. Em primeiro lugar, haverá uma tendência absolutamente humana de esquecer sobre o problema assim que o programa de produção estiver funcionando corretamente, portanto temos que impor uma pequena disciplina sobre nós mesmos. Possivelmente mais importante, entretanto, é a chance de que o mero ato de construir o caso de teste nos faça descobrir o problema (WEINBERG, 1971, p.196-197, tradução nossa).

Novas funcionalidades e eventuais correções inseridas em um código têm o potencial de adicionar novos defeitos no mesmo. Por essa razão, "após cada correção [ou adição de funcionalidade], deve-se executar a base de casos de teste inteira para assegurar que o sistema não foi danificado de uma forma obscura (BROOKS, 1995, p.242-243, tradução nossa)." Ainda segundo Brooks (1995, p.246, tradução nossa), "vale a pena construir diversas proteções de depuração e código de teste, talvez até mesmo 50 por cento do produto que está sendo depurado." Isso é particularmente verdadeiro no caso de se desenvolver o software de forma iterativa.

> Se você desenvolve software em *iterações* (...) você irá fazer mudanças sérias no código após ele ter sido escrito. Isso é perigoso (...). Para fazer mudanças de forma segura, é necessário haver uma forma imediata para encontrar e corrigir conseqüências indesejáveis. A forma mais eficaz de facilitar mudanças é ter uma base de testes automatizados (...). Uma base de testes irá encontrar conseqüências indesejáveis imediatamente e se for boa, também irá indicar a causa

> do problema (POPPENDIECK & POPPENDIECK, 2003, p.147-148, tradução nossa).

Infelizmente, "é impossível testar absolutamente tudo, sem que os testes se tornem tão complicados e propensos a erros quanto o código de produção. É suicídio não testar nada. (...) Você deve testar coisas que possam vir a quebrar (BECK, 2000, p.116-117, tradução nossa)." A idéia de Beck é que ao longo do tempo, e com a prática, os desenvolvedores de um projeto se tornem cada vez mais capazes de identificar que tipo de situações tendem a gerar mais erros.

São nelas que eles irão concentrar os maiores esforços de teste. É uma abordagem com boas chances de funcionar, especialmente levando-se em conta que "(...) 80 por centro dos defeitos podem ser rastreados para 20 por cento de todas as possíveis causas (...) (PRESSMAN, 1997, p.203, tradução nossa)."

> Os programadores escrevem testes método após método. Um programador escreve um teste sob as seguintes circunstâncias:
> - Se a interface de um método não é muito clara, você escreve um teste antes de escrever o método;
> - Se a interface for clara, mas você imagina que a implementação será um tanto complicada, você escreve um teste antes de escrever o método;
> - Se você imaginar uma circunstância incomum na qual o código deva funcionar, você escreve um teste para comunicar a circunstância;
> - Se você encontrar um problema mais tarde, você escreve um teste que isole o problema e
> - Se você estiver prestes a refatorar algum código, não tiver certeza sobre a forma como ele deve funcionar, e ainda não houver um teste para o aspecto do comportamento em questão, você escreve um teste primeiro (BECK, 2000, p.117-118, tradução nossa).

O processo básico de criação de testes e implementação de funcionalidades é simples. Ele se baseia nos passos descritos adiante:

1. Rapidamente adicione um teste;
2. Execute todos os testes e veja o novo falhar;
3. Faça uma pequena mudança;

4. Execute todos os testes e veja-os funcionar e
5. Refatore para remover duplicações (BECK, 2003, p.5, tradução nossa).

Cada vez que um novo elemento precisa ser inserido no código de produção, o programador começa escrevendo um teste que deve falhar enquanto o novo fragmento de código de produção não for inserido. Ele verifica se o teste realmente falha e em seguida adiciona o novo código de produção. Se tudo correr bem, o novo teste deve, então, funcionar, bem como todos os demais que existiam anteriormente no sistema. Havendo uma falha nos testes, o programador irá depurar o código e corrigi-lo até que todos os testes funcionem.

Para fazer o código funcionar, é permitido cometer qualquer tipo de pecado. Uma vez funcionando, o programador faz uma revisão do novo código em busca de duplicações e oportunidades de refatoração. Desta forma, "primeiro, resolvemos a parte 'que funciona' do problema. Depois resolvemos a parte do 'código limpo' (BECK, 2003, p.15, tradução nossa)."

Escrever um teste antes de cada história também representa uma forma de aprimorar a análise sobre as características dela. Pois a necessidade de implementar um teste força o desenvolvedor a buscar maiores detalhes sobre o comportamento da funcionalidade (BECK, 2000). Trata-se de um aspecto importante, visto que "a tarefa crucial é fazer com que o produto seja definido. Muitas, muitas falhas se relacionam exatamente com aqueles aspectos que nunca foram bem especificados (BROOKS, 1995, p.142, tradução nossa)."

O desenvolvimento orientado a testes também produz efeitos positivos no design da aplicação. Isso ocorre porque o desenvolvedor se coloca no papel de consumidor de

seu próprio código e a necessidade de implementar o teste o força a simplificar a implementação do código de produção.

> (...) escrevendo os testes primeiro, você se coloca no papel do *usuário* do seu código, ao invés de *desenvolvedor* do seu código. A partir desta perspectiva, você normalmente consegue obter um sentimento muito melhor sobre como uma interface será realmente utilizada e pode ver oportunidades para melhorar seu design (HUNT & THOMAS, 2003, p.115, tradução nossa).

A necessidade de automatizar os testes leva à necessidade de tornar o design mais fácil de ser testado. Normalmente, isso leva a melhorias no design. Ou seja, a simples necessidade de criar um teste automatiza é uma maneira de forçar os desenvolvedores a tornarem o código mais simples e freqüentemente melhor elaborado.

> Quanto mais complexo um objeto for, mais difícil será testá-lo completamente. Mas o inverso também é verdade: quanto mais simples um objeto ou método for, mais fácil será testá-lo.
> Quando estamos testando um objeto e podemos identificar coisas que podem quebrar, mas parecem impossíveis de serem testadas, deixamos a pressão por testar tudo que possa quebrar fluir de volta para o design. Talvez possamos simplificar este objeto, quebrá-lo e torná-lo mais fácil de testar.
> Normalmente podemos simplificar o código e torná-lo mais fácil de testar. No processo, tornamos o código mais simples de entender e mais provável de estar correto. Ganhamos em todos os lados. Ganhamos um código melhor e testes melhores (JEFFRIES, ANDERSON et al., 2001, p.234-235, tradução nossa).

Outro aspecto que ajuda a melhorar o design é a freqüente utilização de uma técnica conhecida como **programação por intenção** associada à criação dos testes automatizados. "A idéia principal da programação por intenção é a comunicação, especificamente a comunicação da nossa *intenção* para qualquer um que esteja lendo o código (ASTELS, 2003, p.45, tradução nossa)." Sendo assim, "codifique *o que* você deseja e não *como* fazer o que você deseja. (...) [Os desenvolvedores] executam uma pequena tarefa criando os testes primeiro, sempre expressando a intenção no código, ao invés do algoritmo (JEFFRIES, ANDERSON et al., 2001, p.107, tradução nossa)."

Ao escrever o teste primeiro, o programador age como se ele já pudesse encontrar no código de produção todos os elementos (classes, métodos etc) de que irá precisar. Isso certamente não é verdade, porque o teste é escrito antes de se implementar o código da funcionalidade. Entretanto, agindo como se o código de produção já estivesse lá e escrevendo da forma mais natural que lhe pareça, o desenvolvedor procura expressar claramente suas intenções utilizando as interfaces que lhe pareçam mais razoáveis para resolver o problema em questão. Em princípio, tal código de teste provavelmente não será nem mesmo compilável. Só depois de expressar adequadamente a sua intenção é que o desenvolvedor irá tentar implementar as interfaces que imaginou dentro do código de produção.

Além de proteger o sistema, os testes automatizados também ajudam a equipe de desenvolvimento a documentá-lo. Os testes representam exemplos de utilizações do código e, portanto, podem ser bastante úteis para ajudar a compreendê-lo. Além disso, "ao contrário da documentação escrita, o teste não perde o sincronismo com o código (a não ser, naturalmente, que você pare de executá-lo) (HUNT & THOMAS, 2003, p.6-7, tradução nossa)."

> Não chega a ser surpresa que seja difícil, se não impossível, manter a documentação precisa sobre como o software foi construído. (...) Entretanto, se um sistema possui uma base de testes abrangente que contenha tanto testes dos desenvolvedores, quanto testes dos clientes, estes testes serão, de fato, um reflexo preciso de como o sistema foi construído. Se os testes forem claros e bem organizados, eles serão um recurso valioso para compreender como o sistema funciona a partir do ponto de vista do desenvolvedor e do cliente (POPPENDIECK & POPPENDIECK, 2003, p.148-149, tradução nossa).

O desenvolvimento orientado a testes, embora contenha idéias relevantes para melhorar os projetos de software, pode se revelar um problema caso reduza a velocidade de desenvolvimento. Em princípio, isso é o que parece ocorrer, na medida em que a

automação dos testes é um exercício que não é executado em grande parte dos projetos de software, os quais já sofrem problemas de atraso sem considerar a criação destes testes. Imagine-se o que aconteceria se tivessem também que criá-los.

A prática, entretanto, revela que a adoção de testes automatizados acelera o processo de desenvolvimento e o torna mais previsível. “Descobri que escrever bons testes acelera enormemente a minha programação(...). Isso foi uma surpresa para mim e é contra-intuitivo para a maioria dos programadores (FOWLER, 2000, p.89, tradução nossa).” Em termos gerais, equipes XP acreditam que “a única forma de se mover rapidamente e com confiança é tendo uma rede de testes, tanto de unidade, quanto de aceitação (JEFFRIES, ANDERSON et al., 2001, p.33, tradução nossa).”

> Os testes exercem diversos papéis fundamentais durante o processo de desenvolvimento de software. Primeiro, os testes comunicam sem ambigüidade como as coisas devem funcionar. Segundo, eles provêm feedback indicando se o sistema realmente funciona da forma esperada. Terceiro, os testes fornecem os mecanismos de proteção que possibilitam aos desenvolvedores fazer mudanças ao longo do processo de desenvolvimento (...). Quando o desenvolvimento termina, a base de testes fornece uma representação precisa de como o sistema efetivamente foi construído (POPPENDIECK & POPPENDIECK, 2003, p.145-146, tradução nossa).

5.2.9 Refatoração

Sistemas mudam ao longo do tempo, especialmente quando são desenvolvidos de forma iterativa. Nesta obra já tivemos a oportunidade de identificar inúmeras razões que levam à necessidade de alterações em um software, as quais normalmente são acompanhadas de riscos e oportunidades. Podem ser úteis para resolver novos problemas dos usuários, mas podem fazer com que partes do sistema deixem de funcionar ou que sua estrutura se deteriore.

Tradicionalmente, “os reparos tendem a destruir a estrutura, elevar a entropia e a desordem de um sistema (BROOKS, 1995, p.243, tradução nossa).” Além disso, “sem melhoria contínua, qualquer sistema de software irá sofrer. As estruturas internas irão se calcificar e se tornar frágeis (POPPENDIECK & POPPENDIECK, 2003, p.141, tradução nossa).” Por esta razão, projetos XP investem diariamente no aprimoramento do design e na identificação de pontos da arquitetura que estejam se degradando. À medida que são encontrados, são corrigidos através de uma técnica conhecida como **refatoração**.

A “refatoração é o processo de fazer mudanças em um código existente e funcional sem alterar seu comportamento externo. Em outras palavras, alterar *como* ele faz, mas não *o que* ele faz. O objetivo é aprimorar a estrutura interna (ASTELS, 2003, p.15, tradução nossa).” Tal processo é conduzido permanentemente por todos os desenvolvedores como “uma forma disciplinada de limpar o código minimizando a chance de introduzir bugs (FOWLER, 2000, p.xvi, tradução nossa).”

> A necessidade de refatoração aparece à medida que a arquitetura evolui, amadurece e novas funcionalidades são solicitadas pelos usuários. Novas funcionalidades podem ser adicionadas ao código uma de cada vez, mas geralmente elas estarão relacionadas umas com as outras e freqüentemente será melhor adicionar um mecanismo arquitetural para dar suporte ao novo conjunto de funcionalidades. Isso comumente acontece de forma natural como uma refatoração para remover duplicação quando você adiciona o segundo ou terceiro de um conjunto de itens relacionados (POPPENDIECK & POPPENDIECK, 2003, p.141-142, tradução nossa).

Durante a refatoração de um código, “cada passo é simples, até mesmo simplista. (...) Apesar disso, o efeito cumulativo destas pequenas mudanças podem melhorar significativamente o design (FOWLER, 2000, p.xvi-xvii, tradução nossa).” Fazendo isso, os desenvolvedores procuram evitar situações nas quais os programas se tornem difíceis de serem modificados, pois ”o compilador não se importa se o código é

feio ou limpo. Mas, quando mudamos o sistema, existe um humano envolvido e humanos se importam. Um sistema mal projetado é difícil de alterar (FOWLER, 2000, p.6, tradução nossa)."

Portanto, o objetivo é elaborar programas que "sejam fáceis de ler, tenham toda a lógica em um e apenas um lugar, não permitam que mudanças ameacem o comportamento existente e permitam que lógicas condicionais sejam expressas da maneira mais simples possível (FOWLER, 2000, p.60, tradução nossa)."

É importante observar que a integridade conceitual sofre à medida que o design se deteriora. Como vimos anteriormente, a integridade conceitual é um fator essencial para tornar o sistema fácil de ser utilizado. Além disso, "(...) [a] integridade conceitual de um produto não apenas o torna mais fácil de usar, como também facilita a sua construção e o torna menos sujeito a bugs (BROOKS, 1995, p.142, tradução nossa)." Sendo assim, a adoção da prática de refatoração procura manter a integridade conceitual fazendo com que o sistema mantenha as seguintes características ao longo do tempo:

> 1. **Simplicidade** – Em quase todas as áreas, um design simples e funcional é o melhor design. (...)
> 2. **Clareza** – O código deve ser fácil de entender por todos aqueles que irão eventualmente trabalhar com ele. (...)
> 3. **Adequação ao uso** – Todo design deve alcançar o propósito para o qual foi criado. (...)
> 4. **Ausência de repetição** – Código idêntico jamais deveria existir em dois ou mais lugares. (...)
> 5. **Ausência de funcionalidades extras** – Quando o código não é mais necessário, o desperdício envolvido em mantê-lo é elevado (...) (POPPENDIECK & POPPENDIECK, 2003, p.142-143, tradução nossa).

A evolução do design ao longo das iterações com base no uso da refatoração é uma forma diferente de abordar o desenvolvimento de software, visto que tradicionalmente espera-se que o design seja criado antes de se iniciar a implementação do sistema. Entretanto, utilizando as práticas do XP "você descobre que o design, ao

invés de ocorrer todo no início, ocorre continuamente durante o desenvolvimento. Você aprende construindo o sistema como melhorar o design (FOWLER, 2000, p.xvi-xvii, tradução nossa)."

> O historiador da engenharia Henry Petroski escreveu extensivamente sobre como o design realmente acontece. Os engenheiros começam com alguma coisa que funcione, aprendem com suas fraquezas e melhoram o design. As melhorias não se originam apenas na tentativa de atender às demandas dos clientes ou da adição de funcionalidades; as melhorias também são necessárias porque sistemas complexos possuem efeitos que não são bem compreendidos durante o período de design. Escolhas sub-otimizadas constituem uma parte intrínseca do processo de criação de designs complexos no mundo real. Não é razoável esperar um design perfeito que preveja todas as possíveis contingências e efeitos em cascata decorrentes de simples mudanças. O pesquisador de design Donald Norman nota que é necessário cinco ou seis tentativas para realmente atingir um produto correto (POPPENDIECK & POPPENDIECK, 2003, p.140, tradução nossa).

Equipes XP normalmente não alocam um tempo especial do projeto para refatorar. Diversas pequenas refatorações são executadas diariamente, à medida que novas funcionalidades são produzidas, sempre que os desenvolvedores identificam a necessidade de melhorar o código. Existem três situações que são particularmente críticas:

> 1. Quando existe duplicação;
> 2. Quando percebemos que o código e/ou a sua intenção não está clara e
> 3. Quando detectamos *code smells*, isto é, sutis (ou não tão sutis) indicações da existência de um problema (ASTELS, 2003, p.15, tradução nossa).

Duplicação de código costuma ser um problema recorrente em diversos projetos e uma das principais razões para o design se degradar rapidamente. Além disso, reduzem significativamente o ritmo de trabalho da equipe de desenvolvimento.

> Códigos mal projetados normalmente demandam mais código para fazer a mesma coisa, freqüentemente porque o código literalmente faz a mesma coisa em diversos lugares. Portanto, um importante aspecto de melhoria do design é eliminar duplicação no código. A importância

> disso reside nas modificações futuras. Reduzir a quantidade de código não fará o sistema rodar mais rapidamente (...) entretanto, faz uma grande diferença na hora de uma modificação (FOWLER, 2000, p.55, tradução nossa).

Por essa razão, equipes XP utilizam um princípio conhecido como *DRY – Don't Repear Yourself* (não se repita). Trata-se de "uma técnica fundamental que exige que cada fragmento de conhecimento no sistema tenha uma única, não ambígua e definitiva representação (HUNT & THOMAS, 2003, p.32, tradução nossa)."

Os desenvolvedores também dedicam atenção significativa à semântica do código. Por esta razão, procuram utilizar nomes que façam sentido, evitam abreviações, adotam um código padronizado e seguem outras diretrizes que ajudam a melhorar a forma como o código comunica sua intenção (FOWLER, 2000). Por sua vez, "o conceito de *code smells* é amplamente utilizado pela comunidade XP para se referir a características do código que indicam qualidade inferior à aceitável (ASTELS, 2003, p.18, tradução nossa)."

> Quando você descobre que tem que adicionar uma funcionalidade ao programa e o código do programa não está estruturado de uma forma conveniente para adicioná-la, primeiro refatore o programa para tornar mais fácil a adição da funcionalidade, em seguida a adicione (FOWLER, 2000, p.7, tradução nossa).

A atenção constante com a melhoria do código também é usada como forma de torná-lo auto-explicativo, reduzindo os investimentos necessários em documentação e os riscos de que a mesma se torne obsoleta. "Para fazer com que a documentação seja mantida, é crucial que ela seja incorporada ao programa fonte, ao invés de mantida em um documento separado (BROOKS, 1995, p.249, tradução nossa)."

Embora os programadores possam utilizar comentários para documentar o código fonte, os mesmos são evitados. "Quando você sentir a necessidade de escrever

um comentário, primeiro tente refatorar o código de modo que qualquer comentário se torne supérfluo (FOWLER, 2000, p.88, tradução nossa).” Essa recomendação se baseia na idéia de que a maioria dos “comentários é desnecessária se o código for escrito de modo que a intenção esteja clara. Se e quando escrevermos comentários, devemos assegurar que eles comuniquem o *porquê* e não o *como* (ASTELS, 2003, p.54, tradução nossa).”

> Uma heurística que seguimos é que sempre que sentimos a necessidade de comentar alguma coisa, escrevemos um método ao invés do comentário. Este método contém o código sobre o qual se faria o comentário mas é nomeado de acordo com a intenção do código, ao invés de indicar como ele faz o que faz (FOWLER, 2000, p.77, tradução nossa).

Apesar dos aparentes benefícios da refatoração, um problema que pode surgir é a velocidade do desenvolvimento. Em princípio, o esforço de refatoração parece representar um re-trabalho e um custo adicional para o projeto, na medida em que seria preferível projetar o design correto desde o início. Como os projetos de software normalmente sofrem com prazos curtos, pode ser que simplesmente não haja tempo disponível para refatorar.

> Nós argumentamos que não há tempo para *não* refatorar. O trabalho apenas avançará mais lentamente à medida que o código se torne mais complexo e obscuro. Como sugerido na figura [5.3], incorrer em débitos de refatoração irá aniquilar a produtividade da equipe.

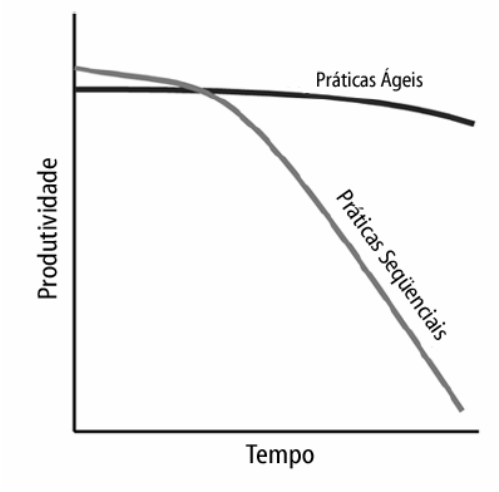

Figura 5.3: melhoria contínua do design sustenta a produtividade.

> Refatoração não é desperdício; ao contrário, é um método chave para evitar desperdícios provendo valor de negócio para os clientes. Uma base de código bem projetada é a fundação de um sistema que consiga responder às necessidades dos clientes tanto durante o desenvolvimento, quanto durante a vida útil do sistema (POPPENDIECK & POPPENDIECK, 2003, p.144-145, tradução nossa).

Equipes XP acreditam que refatorar diariamente ajuda a manter um ritmo acelerado de desenvolvimento. Há uma percepção de que seja relativamente fácil avançar rapidamente sem refatoração no início do projeto, mas quanto mais a base de código cresce, mais a produtividade é comprometida. A falta de tempo é freqüentemente usada como desculpa para não refatorar, entretanto a refatoração que não é feita imediatamente acaba tendo que ser feita mais tarde, quando ela se torna mais custosa e a pressão de tempo se torna ainda maior (HUNT & THOMAS, 2000).

> (...) um bom design é essencial para o desenvolvimento rápido do software. De fato, o objetivo básico de se ter um bom design é permitir o desenvolvimento rápido. Sem um bom design, você consegue avançar rapidamente por um tempo, mas logo o design ruim começa a te atrasar. Você gasta tempo buscando e corrigindo bugs ao invés de adicionar novas funcionalidades. As mudanças consomem mais tempo à medida que você tenta compreender o sistema e encontrar o código duplicado. Novas funcionalidades precisam de mais codificação (FOWLER, 2000, p.57, tradução nossa).

Por maiores que sejam os benefícios aparentes da refatoração, toda mudança no código traz consigo o potencial de que algo deixe de funcionar. Por essa razão, a adoção da prática de refatoração só pode ocorrer em projetos que produzam testes automatizados. “Uma ferramenta chave que anda de mãos dadas com a refatoração, e de fato a torna viável são os testes automatizados (POPPENDIECK & POPPENDIECK, 2003, p.144-145, tradução nossa).

5.2.10 Integração Contínua

Equipes XP normalmente são compostas por diversos programadores, trabalhando em pares de acordo com a prática de código coletivo. Isso cria dois problemas práticos. O primeiro é que “sempre que diversos indivíduos estão trabalhando na mesma coisa, ocorre uma necessidade de sincronização (POPPENDIECK & POPPENDIECK, 2003, p.34-35, tradução nossa).” O segundo é que os pares precisam ser capazes de evoluir rapidamente sem interferir no trabalho uns dos outros. Portanto, enquanto desenvolve, “você quer fingir que é o único programador no projeto. Você quer marchar adiante em velocidade máxima ignorando a relação entre as mudanças que você efetua e as mudanças que os demais estão fazendo (BECK, 2000, p.97-98, tradução nossa).”

Esta questão é resolvida no XP utilizando-se uma prática conhecida como **integração contínua**. Os pares trabalham de forma isolada, porém integram o que produzem com a versão mais recente do código de produção, diversas vezes ao dia. Isto é, os pares se sincronizam com freqüência à medida que terminam pequenas atividades de codificação (BECK, 2000).

Toda vez que um par integra seu código, há um risco de que identifique um erro na integração. Isto é, existe a chance, por exemplo, de que dois pares distintos tenham efetuado alterações conflitantes em uma mesma linha do código.

> O esforço necessário para resolver as colisões não pode ser muito grande.
> Isso não é um problema. A refatoração constante tem o efeito de dividir o sistema em muitos objetos e métodos pequenos. Isso reduz as chances de que dois pares de programadores mudem a mesma classe ou método ao mesmo tempo. Se eles fizerem isso, o esforço necessário para reconciliar as mudanças será pequeno, porque cada um representa apenas algumas horas de desenvolvimento (BECK, 2000, p.98-99, tradução nossa).

De um modo geral, os pares procuram descobrir estes eventuais conflitos tão cedo quanto possível, pois "quanto mais esperamos, pior as coisas vão ficando. (...) um bug introduzido ontem é bem fácil de encontrar, enquanto dez ou cem introduzidos semanas atrás podem se tornar quase impossíveis de serem localizar (JEFFRIES, ANDERSON et al., 2001, p.78-79, tradução nossa).

Integrando rapidamente, os pares também asseguram que o lote de trabalho a ser integrado será pequeno. Afinal, não se consegue produzir muita coisa em um espaço de tempo curto, como por exemplo de uma ou duas horas. Desta forma, se houver um erro na integração, o mesmo será referente a um lote pequeno de trabalho, onde menos coisas podem falhar. Portanto, o tempo para corrigir eventuais problemas tende a ser pequeno (POPPENDIECK & POPPENDIECK, 2003).

> *Builds*[6] mais freqüentes são melhores; eles fornecem feedback muito mais rápido. *Builds* e testes executados durante os *builds* devem ser automatizados. Se não forem, o próprio processo de *build* irá introduzir erros e a quantidade de trabalho manual tornará proibitiva a execução suficientemente freqüente dos *builds*. (...)
> O princípio geral é que se os *builds* e os teste tomarem muito tempo, eles não serão utilizados, portanto, invista em torná-los rápidos. Isso gera uma preferência por *builds* mais freqüentes, com testes menos

[6] Build é o processo de compilar, montar e empacotar o programa.

> abrangentes, mas ainda assim é importante executar todos os testes à noite ou nos finais de semana (POPPENDIECK & POPPENDIECK, 2003, p.35, tradução nossa).

O processo de integração é serial, isto é, somente um par pode integrar o seu código de cada vez. Isso assegura que eventuais erros de integração estarão sempre relacionados a um único par: aquele que está integrando no momento. Somente após assegurar que a integração está perfeita, todos os testes executam com sucesso e o sistema encontra-se em um estado consistente poderá outro par fazer a integração (BECK, 2000).

5.2.11 Releases Curtos

O XP considera que "um projeto de software é um investimento. O cliente investe uma certa quantidade de recursos na expectativa de obter um retorno dentro de certo prazo (TELES, 2004, p.185)." O volume de retorno depende do valor de negócio produzido e do tempo em que o mesmo é entregue.

O XP procura maximizar o retorno dos projetos assegurando que o maior valor de negócio possível seja entregue ao final de cada release e que cada release tenha uma duração curta. Isso é feito através do processo contínuo de priorização que seleciona sempre as histórias de maior valor para serem implementadas primeiro. Além disso, procura antecipar o retorno entregando software rapidamente.

Neste sentido, trabalha com a prática de **releases curtos**, que consiste em colocar o sistema em produção com freqüência, em prazos curtos, normalmente de dois ou três meses. Isso costuma ser bem-vindo, porque "clientes gostam de entregas rápidas. (...) entrega rápida normalmente se traduz em aumento de flexibilidade no negócio (POPPENDIECK & POPPENDIECK, 2003, p.70-71, tradução nossa)."

> Entrega rápida é uma abordagem baseada em opções para o desenvolvimento de software. Permite que você mantenha as suas opções em aberto até que você tenha reduzido incertezas e possa tomar decisões mais informadas e baseadas em fatos (POPPENDIECK & POPPENDIECK, 2003, p.70-71, tradução nossa).

Todo investimento implica no consumo de algum recurso na esperança de que o retorno seja maior, de modo a pagar o que foi consumido e gerar uma sobra (o retorno líquido). No caso de software, por exemplo, os projetos consomem dinheiro na forma de pagamentos efetuados para a equipe de desenvolvimento, entre outros. Espera-se, naturalmente, que o software produzido gere um valor superior ao montante de dinheiro gasto ao longo do projeto. As despesas de um projeto implicam a existência de um fluxo de caixa, ou seja, uma agenda de pagamentos e eventuais retornos (TELES, 2004).

Trabalhando com releases curtos e priorização permanente, a equipe procura assegurar que os primeiros releases gerem a maior parte do valor do sistema (por conterem, em princípio, as funcionalidades com maior valor para o negócio). Portanto, espera-se que o maior retorno esteja concentrado mais próximo do início do projeto, quando o cliente normalmente ainda não efetuou a maior parte dos desembolsos.

Colocando releases em produção rapidamente, o cliente tem a oportunidade de receber feedback concreto sobre o real potencial de retorno do projeto. Portanto, tem a chance de decidir cedo, quando ainda não gastou muito, se continua ou não com o projeto e particularmente se continua ou não investindo na mesma equipe de desenvolvimento. Portanto, a adoção de releases curtos funciona como uma estratégia de gestão de risco do projeto (TELES, 2004).

> Entregar rapidamente significa (...) que as empresas têm menos recursos atrelados a trabalho em andamento (...)
> Uma grande pilha de trabalho em andamento contém riscos adicionais, além da obsolescência. Problemas e defeitos, ambos pequenos e grandes, freqüentemente se mantêm escondidos em pilhas de trabalho

> parcialmente finalizado. Quando os desenvolvedores criam uma grande quantidade de código sem testar, os defeitos se empilham. Quando o código é desenvolvido, mas não é integrado, a parte mais arriscada do esforço continua lá. Quando um sistema está finalizado, mas não está em produção, o risco continua. Todos estes riscos podem ser significativamente reduzidos encurtando-se a cadeia de valor (POPPENDIECK & POPPENDIECK, 2003, p.70-71, tradução nossa).

Entre outras vantagens, "releases curtos e freqüentes provêm benefício cedo para o cliente enquanto fornecem feedback rápido para os programadores (JEFFRIES, ANDERSON et al., 2001, p.49, tradução nossa)." Eles têm a chance de aprender o que a comunidade de usuários, como um todo, pensa a respeito do sistema. Ao longo do desenvolvimento de um projeto XP, é comum os programadores interagirem diariamente com um representante dos usuários. Mas, sempre existe a possibilidade de que esta pessoa não represente adequadamente o interesse geral de toda a comunidade de usuários (BOEHM & TURNER, 2003). Uma forma de descobrir isso é colocando o software em produção o quanto antes. Pois, se houver falhas na representação, serão mais fáceis e rápidas de serem corrigidas se o release não contiver um número excessivamente grande de funcionalidades.

> Uma das coisas mais importantes que você pode fazer em um projeto XP é liberar releases cedo e com freqüência. (...) Você não quer perder a chance de aprender o que os usuários realmente querem. Você não quer perder o aumento na confiança que virá quando você mostrar às pessoas que fez alguma coisa útil (JEFFRIES, ANDERSON et al., 2001, p.50, tradução nossa).

5.2.12 Metáfora

Uma equipe de desenvolvimento formada por diversos programadores convive, entre outros, com o desafio de manter a integridade conceitual do sistema mesmo havendo diversos projetistas criando estruturas novas na arquitetura ao longo do tempo.

Isso pode ser resolvido caso exista um mecanismo capaz de alinhar o pensamento dos diversos projetistas assegurando que todos compartilhem uma visão única de como adicionar e manter as funcionalidades do sistema. O XP utiliza o conceito de **metáfora** para atingir este objetivo.

> (...) a maioria dos sistemas reflete falta de unidade conceitual (...). Usualmente isso se origina (...) da separação do design em muitas tarefas feitas por muitas pessoas.
> (...) a integridade conceitual é a consideração mais importante do design de um sistema. É melhor que um sistema omita certas funcionalidades anômalas e melhoramentos, e refletir um conjunto uniforme de idéias de design do que ter um sistema que contenha muitas idéias boas, mas independentes e disjuntas (BROOKS, 1995, p.42, tradução nossa).

Cockburn (2002, p.227, tradução nossa) sugere que "(...) a programação deveria ser vista como uma atividade através da qual os programadores formam ou atingem uma certa idéia, uma teoria, sobre os problemas que têm nas mãos." Ele se baseia no trabalho de Peter Naur que vê a programação como sendo a *construção de uma teoria*. "Usando as idéias de Naur, o trabalho do designer não é passar adiante 'o design', mas passar adiante 'as teorias' que serviram para direcionar o design (COCKBURN, 2002, p.227, tradução nossa)."

Tais teorias representam a visão geral que rege as decisões sobre como as estruturas serão organizadas no software. Segundo Naur:

> 1. O programador, tendo a teoria do programa, consegue explicar como a solução se relaciona com as questões do mundo real que ele procura tratar. (...)
> 2. O programador, tendo a teoria do programa, consegue explicar *porque* cada parte do programa é o que é (...).
> 3. O programador, tendo a teoria do programa, é capaz de responder construtivamente a qualquer requisição de modificação do programa, de modo a suportar as questões do mundo real de uma nova maneira (COCKBURN, 2002, p.231-232, tradução nossa).

No XP, este mesmo conceito é utilizado, mas procura envolver o uso de metáforas, visto que elas "(...) exercem um papel ativo no pensamento e na cognição. Em particular, as metáforas são vistas (...) como aspectos cruciais na disseminação e compreensão de novas idéias e conceitos (BRYANT, 2000, tradução nossa)."

> Kent Beck sugeriu que é útil para uma equipe de design simplificar o design geral de um programa para se adequar a uma única metáfora. Exemplos podem ser, "Este programa realmente se parece com uma linha de produção, com as coisas sendo adicionadas ao chassis ao longo da linha" (...)
> Se a metáfora for boa, as muitas associações que os projetistas criam em torno dela se revelam apropriadas para a situação de programação deles.
> Esta é exatamente a idéia de Naur de passar adiante a teoria do design. (...)
> Uma metáfora compartilhada apropriada permite que uma pessoa suponha precisamente onde outra pessoa da equipe adicionou código e como se encaixar com o código dela (COCKBURN, 2002, p.239, tradução nossa).

Esta forma de utilização de metáforas é um mecanismo poderoso para manter a integridade conceitual de um sistema e, portanto, torná-lo mais fácil de ser utilizado. Um exemplo recorrente de uso eficaz de uma metáfora são as interfaces gráficas baseadas em janelas. Elas representam "um exemplo sublime de uma interface de uso que possui integridade conceitual, conquistada pela adoção de um modelo mental familiar, a metáfora da escrivaninha [*desktop*] (...) (BROOKS, 1995, p.260-261, tradução nossa)."

5.2.13 Ritmo Sustentável

Segundo Brooks (1995, p.14, tradução nossa), "mais projetos de software saem de curso por falta de tempo do que por qualquer das outras causas combinadas." Quando isso acontece, os projetos normalmente adotam duas estratégias: a utilização de recursos no limite máximo e a adoção de horas-extras de trabalho.

O XP, por sua vez, evita as duas abordagens utilizando a prática de **ritmo sustentável**. Em síntese, ela recomenda que os membros da equipe de desenvolvimento trabalhem apenas durante o tempo regulamentar, ou seja, oito horas por dia, e evitem horas-extras tanto quanto possível. Além disso, sugere-se que os prazos sejam mantidos fixos e as cargas de trabalho sejam ajustadas (através de priorização) para se adequar aos prazos (BECK & FOWLER, 2001).

No livro *Slack*, Tom DeMarco (2001) faz uma análise aprofundada sobre a busca cada vez mais intensa das empresas por eficiência máxima, que se traduz em empregar o menor número de funcionários possível e mantê-los plenamente ocupados o tempo todo. Ele apresenta inúmeras evidências demonstrando que os efeitos obtidos acabam sendo o inverso do desejado. Na prática as organizações precisam incorporar algum nível de folga em suas estruturas para que possam operar de maneira adequada. O mesmo ocorre com os projetos de software. "Jamais executaríamos os servidores das nossas salas de computação com utilização plena – por que não aprendemos essa lição no desenvolvimento de software (POPPENDIECK & POPPENDIECK, 2003, p.81, tradução nossa)?"

> (...) utilização plena não provê nenhum valor para a cadeia de valor como um todo; de fato, normalmente faz mais mal que bem. (...)
> Assim como uma rodovia não consegue prover serviços aceitáveis sem alguma folga na sua capacidade, você provavelmente não estará provendo aos seus clientes o nível mais elevado de serviço se não tiver alguma folga em sua organização (POPPENDIECK & POPPENDIECK, 2003, p.81, tradução nossa).

A adoção de horas-extras, por sua vez, se baseia na idéia subjacente de que os seres humanos possuem um comportamento linear e consistente. Isso não se materializa na prática.

> Se humanos fossem lineares, poderíamos dobrar a produção de uma pessoa dobrando alguma entrada. Como a natureza determina,

> entretanto, nem dobrar as recompensas oferecidas, nem a ameaça de punição ou mesmo o tempo utilizado tem um efeito confiável de dobrar a qualidade do pensamento de uma pessoa, velocidade de pensamento, produtividade da programação ou motivação (COCKBURN, 2002, p.44, tradução nossa).

Seres humanos não se comportam como máquinas, portanto se cansam e produzem resultados indesejáveis em função da fadiga. Além disso, "hora-extra é como acelerar em uma corrida: faz algum sentido nos últimos cem metros da maratona (...), mas se você começar a acelerar no primeiro quilômetro, estará simplesmente perdendo tempo (DEMARCO & LISTER, 1987, p.16, tradução nossa)." Dentre os efeitos colaterais, destacam-se:

- Qualidade reduzida;
- Esgotamento pessoal;
- Maior rotatividade e
- Uso ineficaz do tempo durante as horas regulares de trabalho (DEMARCO, 2001, p.64, tradução nossa).

Apesar destes efeitos e de eles serem facilmente observáveis, horas-extras são usadas excessivamente. É como se representassem a única solução viável para atingir os prazos de um projeto.

> Historicamente, tem sido aceito semanas de trabalho de 80 horas, vidas pessoais sacrificadas e ficar até tarde no escritório como características essenciais de sucesso. (...)
> Estes hábitos inevitavelmente levam ao esgotamento, e para as empresas de software, o custo pode ser substancial. Empregados que experimentem esgotamento extremo podem deixar seus empregos ou ter sérios problemas de saúde, o que cria custos de substituição quando os empregados que estão saindo são experientes. Existe evidência demonstrando que engenheiros de software que passam por altos níveis de estresse mental e físico tendem a produzir mais defeitos. Isso resulta em menor qualidade de software (EMAM, 2003, p.6, tradução nossa).

Para muitos gerentes de projeto, a adoção de horas-extras é uma solução intuitiva para elevar a produtividade da equipe de desenvolvimento. Entretanto, "hora-

extra por um longo período é uma técnica de redução de produtividade. Reduz o efeito de cada hora trabalhada (DEMARCO, 2001, p.63, tradução nossa)."

> Como sugerido pela figura [5.4], a produtividade líquida pode eventualmente *aumentar* durante as primeiras 20 horas de hora-extra. Mas cedo ou tarde, todo mundo alcança um ponto em que os resultados diminuem; e, em algum ponto, a produtividade começa a diminuir devido a erros crescentes e falta de concentração. De fato, existe um ponto em que o membro da equipe se torna um "produtor negativo líquido," porque o esforço de re-trabalho causado por erros e defeitos excede a contribuição positiva de novo software desenvolvido (YOURDON, 2004, p.99-100, tradução nossa).

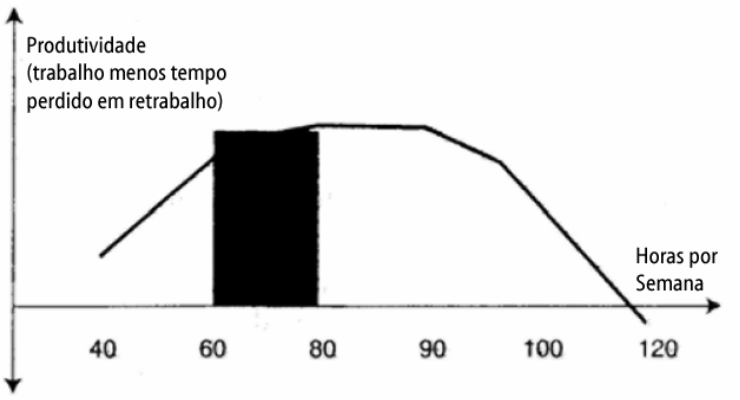

Figura 5.4: produtividade líquida versus horas trabalhadas por semana.

Por estas razões, o XP procura assegurar que a equipe trabalhe apenas durante as horas regulamentares. As pressões de tempo são tratadas através do processo de priorização e o ajuste do escopo de cada iteração para a real capacidade de entrega da equipe. Como essa capacidade pode variar ao longo do tempo, ela é permanentemente monitorada e ajustada à medida que o projeto avança. "Quando estiver sobrecarregado, não pense nisso como se não tivesse tempo suficiente; pense nisso como tendo coisas demais para fazer. Você não pode se conceder mais tempo, mas pode se permitir fazer menos coisas (BECK & FOWLER, 2001, p.25, tradução nossa)."

> Repito que a melhor maneira para se obter uma produtividade mais alta numa empresa, e uma melhor qualidade de vida fora dela, é deixando o escritório assim que acaba o horário de expediente normal e não oferecendo aos respectivos chefes mais tempo do que aquele estipulado pelo contrato e pago pela empresa (MASI, 2000, p.183).

6 ESTUDO DE CASO

O estudo de caso representa o resultado de um projeto de desenvolvimento de software que utilizou os valores e práticas do Extreme Programming durante um período de quatorze meses. O projeto teve teor comercial e foi desenvolvido no Brasil.

Por se tratar de um sistema comercial protegido por um acordo de sigilo, fizemos algumas adaptações de modo que fosse possível relatar os acontecimentos e os resultados da adoção do XP sem revelar aspectos que pudessem comprometer o sigilo das informações. Nosso acesso às informações reflete a participação do autor da dissertação no projeto, durante toda a sua duração. Isso permitiu vivenciar seu dia-a-dia e forneceu acesso aos artefatos que serviram de base para coletar os dados aqui apresentados.

As informações específicas sobre as características do sistema e do cliente não serão apresentadas. Ao invés disso, usaremos uma metáfora com o objetivo de descrever características do projeto sem apresentar o escopo real do mesmo, mas mostrando a influência do Extreme Programming sobre sua condução. Sendo assim, os nomes utilizados deste ponto em diante são pseudônimos, embora a metáfora tenha sido escolhida de modo a estabelecer uma relação próxima com as características reais do projeto.

6.1 Introdução

O Comitê Olímpico da Rússia é responsável por todos os atletas da Rússia que participam de jogos olímpicos ou estão sendo preparados para tal. Ele congrega dez mil atletas que estão distribuídos em diversas cidades da Rússia.

Durante décadas, a antiga União Soviética dominou os resultados dos jogos olímpicos se posicionando quase sempre como a primeira ou segunda colocada em número de medalhas de ouro conquistadas. Mais recentemente, com o seu desmembramento, a Rússia herdou a maior parte dos atletas e começou a sofrer a concorrência de outras nações que vêm se fortalecendo rapidamente, como a China. Como resultado, nas últimas olimpíadas a Rússia perdeu a sua tradicional liderança para a China, ficando em terceiro lugar no ranking de medalhas de ouro.

Esse fato desencadeou a reação do Comitê Olímpico da Rússia com o objetivo de reverter este resultado nos jogos olímpicos de 2008 e os próximos que se seguirão. Diversos estudos foram realizados e geraram iniciativas que já foram ou estão sendo colocadas em prática. Dentre elas se encontra o desenvolvimento de um **Portal de Aprimoramento Esportivo**.

6.2 Portal de Aprimoramento Esportivo

O comitê, através dos estudos realizados, percebeu a necessidade de estabelecer um processo contínuo de aferição das habilidades dos atletas atrelado a um planejamento de treinos para melhorar deficiências. O objetivo era aferir todos os atletas a cada seis meses, identificar suas principais deficiências e traçar um plano de treinos para melhorar suas respectivas habilidades específicas.

Esse objetivo esbarrava em diversos problemas práticos:

- Como aferir de maneira uniforme um total de dez mil atletas dispersos geograficamente?
- Como aferir habilidades específicas em uma gama de 35 esportes distintos?

- Como adequar o processo de aferição às modalidades particulares de cada esporte?
- Como permitir que os próprios atletas e seus treinadores participassem do processo de aferição e criação dos planos de condicionamento físico sem envolvimento de um órgão centralizador?

O comitê notou que havia a necessidade de utilizar um sistema computacional baseado em interface *web* que fosse capaz de organizar e dar acesso a informações de modo a solucionar as dificuldades mencionadas. Assim, iniciou uma busca por produtos comerciais que já estivessem sendo usados por outros comitês olímpicos para resolver essas questões.

Foram encontrados diversos pacotes comerciais que permitiam fazer o planejamento dos treinos, entretanto não se encontrou nenhum que fosse capaz de oferecer o nível de aferição desejável. Em especial, nenhum se mostrou em conformidade com a necessidade de aferir habilidades específicas em esportes distintos. Sendo assim, o comitê adquiriu um pacote para a parte de treinos e contratou os serviços de uma empresa de consultoria para implementar um sistema para fazer aferições.

Seguindo esta solução, o **Portal de Aprimoramento Esportivo** se tornou o resultado da integração de dois sistemas distintos: o **Sistema de Aferições** e o **Sistema de Treinos**. A consultoria contratada pelo comitê adotou o Extreme Programming para o desenvolvimento do Sistema de Aferições, bem como para a integração do mesmo com o Sistema de Treinos e outros sistemas legados. O projeto de desenvolvimento do Sistema de Aferições é o escopo deste estudo de caso.

Sistema de Aferições

Este sistema foi desenvolvido sobre a plataforma J2EE e interagia com outros dois sistemas do Comitê Olímpico: o Sistema de Atletas e o Sistema de Treinos. O **Sistema de Atletas** armazena todo o cadastro de atletas e respectivos treinadores categorizados por esportes, modalidades específicas dentro de cada esporte e equipes. O **Sistema de Treinos** representa o pacote adquirido pelo Comitê Olímpico para organizar os treinos disponíveis para os atletas.

6.3 Extreme Programming no projeto

A adoção do Extreme Programming neste projeto foi sugerida pela empresa de consultoria contratada para implementá-lo. Inicialmente, o assunto se mostrou controverso e não houve completa aceitação da idéia. Os representantes da área de esportes do comitê se mostraram receptivos, enquanto os membros da área de tecnologia da informação (TI) resistiram à idéia. Dentro da consultoria também havia resistência. Dois de seus sócios apoiaram o uso do XP, enquanto a equipe de desenvolvedores se mostrou cética.

Através de um acordo envolvendo todos os interessados, decidiu-se experimentar as práticas do XP durante seis semanas. Caso funcionassem, poderiam ser mantidas durante todo o projeto. Se falhassem seriam substituídas.

O projeto adotou iterações fixas de duas semanas. Embora houvesse um escopo definido, o contrato não o fixou, permitindo com isso que eventuais alterações pudessem ser incorporadas. As próximas seções apresentarão situações específicas do projeto que foram selecionadas para ilustrar como o Extreme Programming colaborou para os resultados alcançados.

6.4 Modelo Conceitual de Aferição de Habilidades

Na reunião inaugural do projeto, foram discutidos diversos assuntos entre a equipe de desenvolvimento e representantes do Comitê Olímpico. Em especial, a ata desta reunião revela que o Gerente de Esportes e sua equipe haviam feito uma revisão completa da especificação do sistema. Segundo ele, a especificação estava em ordem e todas as definições conceituais estavam corretas e finalizadas.

Ao longo do projeto, um funcionário da área de esportes foi designado para dar apoio diário aos desenvolvedores, atuando como representante de todos os usuários. Ele era conhecimento como **requerente**. Por sua vez, uma pessoa da área de tecnologia da informação foi designada para auxiliar os desenvolvedores nos aspectos de tecnologia do Comitê Olímpico. Era chamado de **requerente de TI**.

Na reunião de planejamento da primeira iteração, o requerente escreveu cartões contendo as histórias que deveriam ser implementadas com base na especificação do sistema. Depois de escrever histórias sobre todos os aspectos mais importantes do projeto, os participantes voltaram suas atenções para os cartões que seriam prováveis candidatos de serem implementados na primeira iteração. Deste ponto em diante, passaram a reler tais cartões e rediscutir em detalhes cada uma das histórias.

Os desenvolvedores não conheciam os requisitos do projeto até então. Portanto, o requerente não apenas escreveu histórias, como também explicou cada uma delas diversas vezes para os desenvolvedores. Isso foi útil, porque eles começaram a identificar diversos detalhes que ainda não haviam sido previstos pelo requerente e para os quais ainda não havia uma definição por parte dele.

Nesta reunião, o requerente e outras pessoas do comitê obtiveram um volume significativo de feedback por parte dos desenvolvedores e todos aprenderam mais sobre

as funcionalidades. A partir das discussões, vários detalhes foram esclarecidos pelo requerente, o qual priorizou um conjunto de histórias para a primeira iteração.

Elas envolviam os aspectos essenciais dos processos de aferição de habilidades específicas e não-específicas. Cada tipo de habilidade tinha um modelo próprio de aferição e coube aos desenvolvedores implementar os dois modelos logo na primeira iteração.

Esta priorização não foi a mais natural do ponto de vista técnico, pois inseria já na primeira iteração funcionalidades que dependiam de dados que ainda não existiam, pois nem sequer havia funcionalidades previamente implementadas para o cadastramento dos mesmos. Como exemplo, podemos citar a falta do cadastro de esportes, atletas, habilidades, entre outros.

Para lidar com essa questão, a equipe criou as tabelas que fossem necessárias e as preencheu diretamente com dados fictícios (temporários) que pudessem ser utilizados para permitir a execução do processo de aferição. Foram criadas tabelas para entidades tais como habilidades (específicas e não-específicas), modalidades esportivas, atletas, entre outras. Com esta simplificação, a equipe viabilizou a implementação das histórias mais prioritárias, de modo que o requerente pudesse receber feedback rápido sobre as mesmas.

Ao final da iteração, houve uma reunião de encerramento com a presença dos requerentes. A equipe de desenvolvimento conseguiu implementar todas as histórias acordadas e apresentou o sistema para o requerente, que se mostrou satisfeito com o trabalho e o resultado final.

Avaliando o sistema cuidadosamente, ele se deu conta de diversos detalhes que não haviam sido previstos inicialmente, mas que ficavam evidentes diante da

possibilidade de uso das funcionalidades. Tais detalhes foram listados por ele, que solicitou mudanças para a equipe de desenvolvimento.

Além dos detalhes de implementação, o requerente percebeu um problema mais sério. Embora o modelo conceitual de aferição de habilidades tivesse sido implementado tal como determinado na especificação, utilizando o sistema o requerente notou que havia equívocos conceituais neste modelo, os quais colocariam o sistema em risco se não fossem tratados.

Inicialmente, ele não sabia que soluções deveriam ser implementadas para corrigir o modelo conceitual, sendo assim levou a discussão de volta para o Comitê Olímpico, de modo que pudesse ser debatida por toda a equipe de esportes. E para que a equipe de desenvolvimento não ficasse sem trabalho na iteração seguinte, o requerente priorizou algumas histórias menos importantes.

Dentro do comitê, os problemas do modelo conceitual foram debatidos entre esportistas espalhados por diversas cidades da Rússia. De um modo geral, todos perceberam os problemas apontados pelo requerente, cuja identificação foi possível graças ao uso do sistema. Como o assunto não tinha soluções triviais, a discussão se alongou por um tempo maior que o imaginado inicialmente.

Durante a segunda iteração o requerente informou à equipe de desenvolvimento que as novas definições não estariam prontas antes da reunião de planejamento da iteração seguinte. Em outras palavras, não seria possível implementar as correções do modelo conceitual na terceira iteração. Na melhor das hipóteses, isso só seria viável a partir da quarta iteração. Até lá, a equipe teria que se dedicar a histórias menos prioritárias.

Na quarta iteração a equipe recebeu do requerente as mudanças do modelo conceitual, as quais foram priorizadas. Durante essa iteração, o novo modelo conceitual foi apresentado ao Diretor de Esportes, que não o aprovou. Ele sugeriu outras mudanças significativas e, no meio da iteração, a equipe de desenvolvimento foi informada deste fato. Finalmente, a quinta e a sexta iterações acabaram sendo utilizadas para implementar o novo modelo conceitual que se revelou mais coerente e aderente à necessidade do comitê.

Este episódio gerou um atraso em relação ao cronograma planejado no início do projeto. Entretanto, é importante notar o efeito do feedback e do aprendizado no resultado final do sistema. No início do projeto, o Gerente de Esportes tinha bastante convicção de que o modelo conceitual estava correto. Afinal, já havia sido investido um grande esforço durante o ano anterior para desenvolvê-lo, inclusive com a participação de consultorias especializadas.

Ao contrário do trabalho efetuado no ano anterior, entretanto, ao longo das primeiras seis iterações o requerente teve a oportunidade de receber feedback concreto através do uso do sistema. Assim, teve condições de notar falhas no modelo conceitual que passaram despercebidas anteriormente, levando o comitê a corrigi-lo e aprimorá-lo. Com isso, o processo de aferição de habilidades acabou se tornando mais intuitivo para os atletas.

Estes acontecimentos demonstram a importância da utilização de iterações curtas e feedback rápido. Outro fator que também foi relevante nas primeiras iterações foi a proximidade física entre os desenvolvedores e o requerente, cujo local de trabalho se situava a menos de cinqüenta metros do escritório da equipe de desenvolvimento. Sempre que o requerente avançava nas definições conceituais, as mesmas eram

apresentadas para a equipe de desenvolvimento. Isso permitiu a realização de inúmeros debates sobre o novo modelo conceitual, antes de começar o seu desenvolvimento.

Nestes debates, os desenvolvedores tiveram papel importante ao identificar aspectos que não eram notados pelo requerente. Em determinado momento da quarta iteração, por exemplo, o requerente passou quase três dias discutindo e aprimorando o modelo conceitual com os desenvolvedores. A quantidade de itens discutidos e acertados foi elevada. Caso os envolvidos não estivessem próximos e tivessem que usar outros mecanismos de comunicação, ao invés do diálogo pessoal, face-a-face, certamente teria sido mais demorado convergir para uma solução final em função da quantidade e complexidade dos detalhes que mereciam a atenção de todos.

6.5 Tratamento de Mudanças no Sistema de Atletas

O funcionamento do Sistema de Aferições depende de informações do Sistema de Atletas. Por exemplo, para aferir um atleta é necessário identificar em que esporte ele atua e em qual modalidade. Essas informações são essenciais para determinar que habilidades específicas o sistema irá utilizar para aferir o atleta. Por sua vez, quando um treinador deseja aferir um integrante de sua equipe, o software busca no Sistema de Atletas quais são os atletas subordinados àquele treinador.

Na décima iteração do projeto, logo após a realização de um piloto que contou com a participação de 80 usuários, foi detectada uma falha no sistema quando um atleta mudava de modalidade, enquanto a sua aferição ainda não havia sido finalizada. É comum um atleta treinar em modalidades parecidas, embora esteja associado a uma única modalidade principal. Às vezes, ele evolui tanto em uma modalidade paralela que esta acaba se tornando a principal.

Nestes casos, faz-se uma mudança no Sistema de Atletas para registrar o novo direcionamento do desportista. Isso cria um problema para o Sistema de Aferições quando a mudança de modalidade principal ocorre enquanto o atleta está no meio de uma aferição, que normalmente pode levar dias, ou até semanas, visto que o ciclo completo envolve a utilização do sistema pelo atleta e seu treinador de forma assíncrona.

Nestas situações, o sistema apresentava falha porque a aferição havia sido iniciada em uma modalidade e, de repente, o sistema notava que o atleta não se encontrava mais naquela modalidade principal, impedindo que o mesmo continuasse a ser aferido. Esta falha chamou a atenção do Requerente e da equipe de desenvolvimento sobre a possibilidade de problemas semelhantes acontecerem quando o sistema já estivesse definitivamente em produção.

Depois de vários estudos, todos notaram que o problema realmente poderia ocorrer em produção toda vez que um atleta mudasse de modalidade principal ou fosse transferido para outra equipe, por exemplo. A identificação deste problema foi importante, visto que a incidência do mesmo se mostrou bastante elevada quando o sistema entrou em produção. Afinal, eram dez mil atletas sendo aferidos durante um período de dois meses. Muitos mudavam de modalidade principal e de equipe neste período de tempo.

Inicialmente, nem os desenvolvedores, nem o requerente conseguiram propor uma solução para esta questão. Sendo assim, ela foi registrada e levada para discussão interna no Comitê Olímpico. Nada foi feito a este respeito, nem na décima, nem na décima primeira iteração. Já na décima segunda, a equipe de desenvolvimento realizou inúmeros testes para identificar os problemas que as mudanças no Sistema de Atletas

poderiam acarretar nas funcionalidades que já estavam implementadas no Sistema de Aferições. Esse mapeamento foi repassado para o Comitê, de modo que o requerente pudesse definir internamente as ações a serem realizadas para a correção do problema em cada uma das funcionalidades afetadas.

Tal definição consumiu bastante tempo, de modo que, durante várias iterações, o problema foi deixado de lado, tendo sido finalmente priorizado para a décima sétima iteração. A solução, por sua vez, só ficou concluída na décima oitava iteração. Apesar da demora, em grande parte decorrente da complexidade do problema e da dificuldade de traçar soluções efetivas para o mesmo, a correção foi implementada antes de o sistema entrar em produção.

Uma vez em produção, ela provou que funcionava, mas era ineficiente tamanha a quantidade de ocorrências do problema no período reservado para o ciclo de aferições. Assim, o requerente aprimorou a abordagem e solicitou que a mesma fosse implementada na vigésima quarta iteração, perto do fim do projeto. Esta última solução se mostrou eficiente e resolveu por definitivo o problema que já vinha acompanhando todos os envolvidos no projeto há quatorze iterações.

Sobre este episódio é relevante notar que o problema das mudanças no Sistema de Atletas e sua respectiva solução não foram previstos na especificação do projeto, ou seja, não faziam parte do escopo original. Apesar disso, tratava-se de um problema grave que, segundo o requerente, poderia ter tornado o Sistema de Aferições inútil caso não fosse considerado e tratado.

Isso demonstra que o feedback freqüente, dentro de um modelo de desenvolvimento iterativo pode ser útil para identificar problemas graves e, portanto, ajudar a reduzir os riscos de que o software implementado não atenda às reais

necessidades de seus usuários. Também mostra a importância de permitir que o escopo incorpore o aprendizado gerado ao longo do projeto, o que só é possível quando existe algum nível de flexibilidade no tratamento dele.

6.6 Integração com o Sistema de Treinos

Quando a aferição de um atleta é concluída, inicia-se a preparação do **Plano de Treinos**. Trata-se de um planejamento dos treinos aos quais o atleta será submetido nos meses seguintes com o intuito de melhorar a sua proficiência em habilidades específicas e não-específicas que tenham sido priorizadas pelo seu treinador.

Ao final do processo de aferição, é comum o atleta encontrar uma ou mais habilidades nas quais o seu nível de proficiência encontra-se aquém do desejado. Quando isso acontece, o seu treinador prioriza as habilidades mais críticas, de modo que a montagem do Plano de Treinos possa envolver a escolha de exercícios que estejam associados diretamente às habilidades que foram priorizadas.

Enquanto o Sistema de Aferição cuida da aferição de habilidades e da geração do Plano de Treinos, o Sistema de Treinos faz a gestão de todos os treinos disponibilizados para os atletas. Ele armazena os treinos nos quais um determinado atleta está registrado, quanto já foi realizado, qual o desempenho obtido, bem como o histórico de todos os treinos já realizados pelo atleta. Portanto, é necessária a integração entre os dois sistemas para elaborar o Plano de Treinos de maneira completa, fazendo com que suas informações sejam automaticamente refletidas no Sistema de Treinos.

A integração entre o Sistema de Aferição e o Sistema de Treinos já era prevista desde o início do projeto. Entretanto, os trabalhos referentes a esta parte não começaram desde o início. Nas primeiras sete iterações, a equipe de desenvolvimento não implementou nenhuma funcionalidade que tivesse relação com o Sistema de Treinos.

Neste primeiro momento, a aferição de habilidades foi priorizada em detrimento do Plano de Treinos.

A equipe de desenvolvimento poderia ter seguido outra abordagem na qual fizesse ao menos a preparação de uma infra-estrutura que pudesse acomodar as funcionalidades do Plano de Treinos e as conseqüentes necessidades de integração. Entretanto, seguiu as práticas do XP de manter o design simples e fazer com que a arquitetura evoluísse com base nas necessidades das histórias priorizadas para cada iteração. Com isso, aguardou até a oitava iteração para iniciar os trabalhos relativos à integração com o Sistema de Treinos.

As primeiras funcionalidades relativas ao Plano de Treinos começaram a ser implementadas na oitava iteração, quando o modelo conceitual de aferição de habilidades já se encontrava estável. Isso beneficiou a implementação do Plano de Treinos, pois durante as discussões sobre o modelo conceitual de aferição, o requerente identificou e tratou de diversos pontos importantes sobre o modelo de geração do Plano de Treinos. O feedback obtido pelo requerente em cada iteração facilitou a elaboração e o refinamento do modelo conceitual de geração do Plano de Treinos.

Os principais objetivos da integração já estavam definidos desde a segunda iteração. Entretanto, a forma de implementar tais objetivos sofreu inúmeras alterações ao longo do projeto. Desde a segunda iteração a empresa responsável pela implantação do Sistema de Treinos começou a estudar a possibilidade de implantar uma versão do Sistema de Treinos no escritório da equipe de desenvolvimento do Sistema de Aferições.

Na nona iteração, a equipe de desenvolvimento ainda não conhecia o Sistema de Treinos, pois não tinha acesso ao mesmo. Para contornar este problema, seria necessário

ter acesso ao Sistema de Treinos instalado no Comitê Olímpico ou instalar uma cópia do mesmo no escritório da equipe de desenvolvimento do Sistema de Aferições. Em função de restrições de segurança do comitê, a segunda opção foi adotada. Mas, infelizmente o Sistema de Treinos não foi instalado imediatamente.

Para contornar a falta de acesso a ele, a equipe de desenvolvimento começou a implementar as funcionalidades relativas ao Plano de Desenvolvimento utilizando o conceito de *mock objects* (objetos fictícios que simulam o comportamento de objetos reais) (HUNT & THOMAS, 2003). Inicialmente, cada funcionalidade que envolvia acesso ao Sistema de Treinos utilizava uma interface que simulava o acesso ao mesmo, sempre fornecendo as respostas corretas. Desta forma, mais tarde, a implementação desta interface seria substituída pelo acesso real ao Sistema de Treinos.

Através da simulação do acesso ao Sistema de Treinos, os desenvolvedores contornaram a ausência do mesmo e permitiram que o requerente recebesse feedback rápido sobre as funcionalidades do Plano de Treinos. Isso foi importante, pois permitiu que inúmeros refinamentos fossem feitos, muito antes de ter a integração efetivamente operando.

O Sistema de Treinos era um pacote adquirido pelo comitê e implantado por uma empresa de consultoria. Durante o projeto de implementação do Sistema de Aferições, o Sistema de Treinos estava sendo instalado e personalizado em paralelo.

A equipe de implantação do Sistema de Treinos não trabalhava fisicamente próxima à equipe de desenvolvimento do Sistema de Aferições. Sendo assim, a comunicação entre as equipes era feita basicamente através de telefone ou e-mail, embora tenham ocorrido também algumas reuniões presenciais ao longo do projeto. Nas

poucas vezes em que elas ocorreram, não envolveram todos os desenvolvedores do Sistema de Aferições.

As atas demonstram que a partir da nona iteração, à medida que o processo de integração entre os sistemas se intensificava, cada vez surgiam mais dificuldades em função de equívocos de compreensão das informações. A integração vinha sendo feita através da construção de uma API (*Application Programming Interfaces* – Interface de Integração entre Aplicativos) que era dividida em duas partes: uma cliente, sendo implementada pelos desenvolvedores do Sistema de Aferições e outra servidora, implementada pela equipe do Sistema de Treinos.

Em diversos momentos a equipe do Sistema de Treinos implementou funcionalidades incorretas na API de integração devido aos erros de comunicação. Além disso, a distância física levou a dificuldades de sincronização entre as equipes, pois freqüentemente as requisições da equipe do Sistema de Aferições não eram atendidas a tempo, ou ficavam sem resposta por um período longo.

No início da décima quarta iteração, enquanto ainda havia diversas funcionalidades a serem implementadas que dependiam da integração com o Sistema de Treinos, a equipe de desenvolvimento do Sistema Aferições foi informada de que o contrato entre o comitê e a consultoria responsável pela implantação do Sistema de Treinos estava para terminar. Sendo assim, era necessário planejar da forma mais precisa possível quais seriam as necessidades de integração pendentes para que a equipe do Sistema de Treinos pudesse adicionar os itens pendentes à API de integração.

Essa situação foi problemática para todos os envolvidos no projeto, visto que o requerente ainda tinha dúvidas sobre algumas das partes do sistema que envolviam a integração. A tentativa de prever tudo se revelou difícil e preocupante.

Embora a equipe de desenvolvimento do Sistema de Aferições já pudesse contar com o Sistema de Treinos instalado em seu escritório a partir de certo ponto do projeto, existiam diferenças entre o que estava instalado no comitê e no escritório da equipe de desenvolvimento. No comitê, foram implementadas diversas configurações personalizadas. Infelizmente, tais configurações demoravam a ser instaladas na versão que estava sendo usada no escritório da equipe de desenvolvimento do Sistema de Aferições. Conseqüentemente, a integração deixava de funcionar em alguns casos devido a incompatibilidade de versões. Este tipo de problema começou a ser vivenciado durante a décima terceira iteração e continuou até a décima quinta.

No início da décima sexta o requerente percebeu a necessidade de fazer uma pequena alteração no processo de finalização da preparação do Plano de Treinos. Entretanto, tal mudança envolvia mudanças na API de integração, o que tornou-a inviável, visto que o contrato entre o comitê e a equipe de implantação do Sistema de Treinos já havia sido finalizado.

Na décima sexta iteração teve início um conjunto de testes de integração para verificar se a API estava funcionando a contento. Nesta iteração, a equipe teve sérios problemas com o Sistema de Treinos e não teve acesso ao seu suporte. Foi necessário reinstalar o Sistema de Treinos e o cliente teve de estender o contrato com a sua equipe de implantação.

Durante a décima sétima, décima oitava e décima nona iterações a equipe de desenvolvimento continuou fazendo acertos e testes na integração com o Sistema de Treinos. Na décima nona iteração foi identificado um problema de lentidão na integração. Nesta iteração, finalmente se conseguiu mais ajuda da equipe de implantação do Sistema de Treinos.

Em função dos problemas que ocorreram com o Sistema de Treinos, não foi possível colocar em produção as funcionalidades de geração do Plano de Treinos, o que desagradou bastante o requerente. Ao final da décima nona iteração o desempenho da integração ainda era insatisfatório. Um dos problemas é que a equipe fazia testes no ambiente de desenvolvimento que continha poucos dados. Quando a utilização se dava em produção, com uma grande quantidade de dados cadastrados, o desempenho da API de integração caia drasticamente.

Na vigésima iteração a integração ficou estável e as funcionalidades do Plano de Treinos puderam entrar em produção. Na vigésima primeira iteração, mais uma vez, o requerente solicitou uma funcionalidade que envolvia mudança na integração com o Sistema de Treinos. A equipe mostrou que isso implicaria em mudanças na API de integração. Sendo assim, o requerente mais uma vez acabou tendo que desistir da mudança, embora ela fosse importante.

Esses acontecimentos demonstram como é comum que detalhes passem despercebidos quando se está especificando um sistema ou mesmo parte dele. Ao tornar necessário que o requerente e a equipe de desenvolvimento previssem todas as necessidades de integração muito cedo, criou-se um problema, visto que vários detalhes não foram previstos e precisaram ser tratados mais adiante. A impossibilidade de alterar a API acabou prejudicando o comitê.

Outro problema deste episódio é a distância física entre a equipe do Sistema de Aferições e a equipe do Sistema de Treinos. Era mais difícil e demorado convergir sem que todos estivessem próximos e utilizassem canais de comunicação mais ricos.

6.7 Cadastros

O Sistema de Aferições englobava diversas histórias, dentre as quais a elaboração de formulários para cadastramento de dados. Embora tais cadastros fossem importantes para viabilizar as demais funcionalidades do sistema, eles não foram priorizados pelo requerente nas primeiras iterações.

No início do projeto, o Requerente tinha particular interesse na implementação de funcionalidades que representassem os modelos de aferição de habilidades específicas e não-específicas. Sendo assim, nas primeiras iterações, ele priorizou histórias que ajudariam a validar tais modelos, embora vários dados importantes ainda não existissem cadastrados, tais como habilidades, esportes, modalidades esportivas, atletas, entre outros.

A equipe de desenvolvimento optou por buscar formas de implementar as funcionalidades priorizadas criando mecanismos para contornar a dependência de dados para os quais ainda não haviam sido implementados cadastros. O objetivo disso foi assegurar que o requerente obtivesse feedback rápido para as funcionalidades que considerava mais críticas, o que acabou se revelando valioso, como foi possível observar nas seções anteriores.

O mecanismo criado para solucionar essa questão foi batizado de **Cadastrador** pela equipe de desenvolvimento. O Cadastrador era um módulo do sistema que podia ser executado à parte, sempre que necessário. Ele limpava diversas tabelas do banco de dados e as preenchia com valores fixos e fictícios que permitiam a utilização das funcionalidades. Assim, a equipe de desenvolvimento assegurava a existência dos cadastros necessários já na primeira iteração, embora os dados fossem ainda

simplificados e fictícios. Isso permitiu ao requerente utilizar as funcionalidades na ordem em que foram priorizadas.

Passada a primeira iteração, o requerente continuou demonstrando pouco interesse em implementar formulários de cadastro, o que gerou preocupação no requerente de TI. Na reunião de planejamento da terceira iteração, ele sugeriu então que diversos cadastros do sistema fossem implementados com o uso de planilhas em Excel, as quais poderiam ser carregadas no sistema fazendo algumas alterações no Cadastrador. A idéia foi aceita por todos, e a partir da quarta iteração o requerente já estava providenciando o preenchimento das planilhas seguindo um formato acordado com a equipe de desenvolvimento.

No início da quarta iteração o requerente de TI exerceu pressão para que os cadastros de maior volume fossem priorizados, especialmente aqueles que pudessem ter necessidade de inclusão, alteração e exclusão após a carga inicial em produção. A preocupação dele se baseava no fato de que nem a equipe de desenvolvimento, nem a equipe de esportes e nem mesmo a equipe de TI poderiam fazer modificações diretamente na base de dados de produção do comitê. Apesar do alerta, o requerente não priorizou os formulários de cadastro.

Durante a quinta iteração, as planilhas já estavam em pleno uso dentro do comitê e vários de seus departamentos estavam colaborando através do preenchimento das mesmas. Tal processo fluiu de forma positiva, embora alguns problemas tenham sido identificados. Havia dificuldades de compreensão da nomenclatura usada nas planilhas, já que o modelo conceitual de aferições estava sendo refinado em paralelo. Isso às vezes causava conflitos de nomenclaturas que dificultavam o preenchimento das planilhas. Apesar disso, o resultado foi positivo. Em retrospectiva realizada ao final da décima

quarta iteração, o uso das planilhas foi apontado como um dos aspectos mais positivos do projeto.

A implementação dos primeiros cadastros foi finalmente priorizada para a décima quinta iteração. Os dois cartões priorizados neste sentido foram:

- Cadastro de Equipe Esportiva
- Cadastro de Perfil Esportivo

A análise destas histórias revela duas questões. A primeira é que o projeto foi capaz de entregar funcionalidades relevantes para o requerente durante catorze iterações (aproximadamente sete meses) apesar de não haver sido implementado nenhum formulário de cadastro em todo este período. A segunda é o teor destes primeiros cadastros priorizados.

Os conceitos de Equipe Esportiva e Perfil Esportivo não existiam no início do projeto. Eles foram criados ao longo das inúmeras discussões sobre os modelos de aferição de habilidades, sobretudo o de habilidades específicas.

Nota-se com isso que a priorização tardia dos cadastros se beneficiou de todo o aprendizado das iterações anteriores para que os cadastros efetivamente implementados fizessem referência a aspectos realmente importantes e estáveis do sistema. Isso evitou re-trabalhos futuros e poupou tempo da equipe de desenvolvimento e do próprio requerente na hora de implementar os formulários de cadastros.

A utilização de planilhas ocorreu durante a maior parte do projeto. Em alguns casos, o requerente teve dificuldades para entregar as planilhas preenchidas no prazo acordado com a equipe de desenvolvimento. A ata da reunião de planejamento da décima sétima iteração, por exemplo, mostra a equipe de desenvolvimento pressionando o requerente para fornecer tais planilhas o mais brevemente possível. Isso levanta a

hipótese de que caso os formulários de cadastro tivessem sido implementados desde as primeiras iterações, o esforço poderia ter sido em vão, visto que o comitê não era capaz de gerar os dados a serem cadastrados com agilidade.

Novos cadastros foram priorizados para a décima nona iteração. Desta vez, as histórias priorizadas foram:

- Cadastro de Habilidades Específicas
- Cadastro de Habilidades Não-específicas

Mais uma vez, a implementação destes cadastros se beneficiou de todo o aprendizado anterior sobre os modelos de aferição de habilidades, que já estavam maduros nesta altura do projeto. Isso fez com que estes cadastros fossem implementados uma única vez, sem necessidades de re-trabalho que teriam sido significativos nas primeiras iterações, devido à grande quantidade de alterações efetuadas nos modelos conceituais de aferição.

Além dos cadastros mencionados anteriormente, apenas outros três menos importantes foram implementados. A contagem final de cadastros revela um número inferior ao imaginado no início do projeto. Alguns deixaram de fazer sentido em face das mudanças conceituais, enquanto outros foram eliminados por terem se mostrado desnecessários ou de pouco valor.

6.8 Relatórios

Assim como as funcionalidades de cadastro, os relatórios não foram implementados nas primeiras iterações do projeto. No início da décima segunda iteração foi decidido que os relatórios a serem desenvolvidos teriam formato PDF. Na décima nona iteração, foi priorizada a implementação do primeiro relatório. Importante notar que se trata de uma iteração que teve início logo após a entrada do sistema em

produção. Para este relatório, foi proposto que ele fosse implementado inicialmente de maneira mais simples, em HTML e, posteriormente, passasse a adotar também o formato PDF.

Para a elaboração dos relatórios, houve um grande cuidado de definir o formato visual dos mesmos antes de iniciar a implementação. Para isso, os relatórios foram desenhados inicialmente com o uso de planilhas em Excel. Estes desenhos foram discutidos inúmeras vezes com o requerente antes que qualquer relatório começasse a ser implementado. O objetivo neste caso foi usar protótipos para evitar re-trabalhos. Em retrospectiva no final da vigésima iteração, este modelo de validação e fechamento do formato visual dos relatórios foi apontado como um dos pontos mais positivos do projeto.

Para a vigésima primeira iteração foi priorizado que se faria um estudo para adotar uma ferramenta que fosse capaz de gerar os relatórios em PDF. Entretanto, ao final desta mesma iteração, o Requerente decidiu que não haveria necessidade de gerar os relatórios em PDF e eles permaneceriam sendo implementados em HTML. O requerente acabou gostando dos relatórios em HTML, pois eles facilitavam a cópia das informações.

Freqüentemente, o requerente tinha que copiar as informações de um relatório para uma planilha ou documento do Word, com o objetivo de preparar relatórios internos ao comitê. Utilizando os relatórios em HTML, bastava marcar as partes do relatório que interessavam, copiar e colar no Excel ou no Word. Essa facilidade, aliada a facilidade de implementar os relatórios em HTML e o fato de haver outras histórias mais prioritárias, fez com que o requerente eliminasse completamente as histórias que cuidariam de implementar os relatórios em PDF.

Para a vigésima terceira iteração, foram priorizados apenas relatórios. Depois desta iteração, outros relatórios foram implementados esporadicamente nas iterações seguintes. Além deles, foram implementados também acertos e pequenas alterações nos relatórios previamente implementados. Mais uma vez, o feedback rápido permitiu que o requerente identificasse pequenos erros e aspectos que precisavam ser incorporados para melhorar as informações apresentadas nos relatórios. Enquanto alguns destes aspectos se ocupavam basicamente do formato visual dos mesmos, outros tinham relação direta com o conteúdo apresentado. Os relatórios também se beneficiaram da maturidade das funcionalidades, em função das mudanças nos modelos conceituais que foram feitas nas iterações iniciais.

6.9 Histórico de Ciclos Passados

O Sistema de Aferições é usado semestralmente durante um período de dois meses que recebe o nome de **ciclo**. Fora do ciclo, o sistema é usado basicamente pelo pessoal da área de esportes ou por atletas do comitê que tenham interesse em ver informações históricas armazenadas no sistema.

Desde o início do projeto, o requerente e a equipe de desenvolvimento sabiam que haveria a necessidade de implementar funcionalidades específicas para o tratamento do histórico das informações do sistema, visto que o comportamento esperado é diferente daquele que os usuários encontram quando utilizam o sistema durante o ciclo. Tais funcionalidades não foram priorizadas nas primeiras iterações. Nem mesmo foi priorizada a criação de uma infra-estrutura que pudesse acolher as funcionalidades associadas ao tratamento do histórico do sistema.

O feedback recebido pelo requerente nas primeiras iterações, que levou a inúmeras alterações nos modelos conceituais de aferição, fez com que ele priorizasse

funcionalidades relativas às aferições e deixasse o histórico em segundo plano, situação na qual ele permaneceu até a vigésima primeira iteração, embora desde a terceira já houvesse uma entidade do sistema, chamada Ciclo, associada a todas as partes do sistema que fossem sensíveis ao tempo, tais como Aferições, Planos de Treinos, entre outros.

Para a vigésima primeira iteração, foi priorizada uma história que permitiria ao usuário navegar pela aferição de habilidades do ciclo anterior. Depois disso, o problema do tratamento de histórico só voltou a ser discutido na vigésima quinta iteração, já bem próximo ao final do projeto que teve um total de 27 iterações. Nessa iteração, a equipe de desenvolvimento e o requerente fizeram um estudo detalhado para definir como seria implementado o histórico nas duas últimas iterações do projeto. A partir deste estudo, foram priorizadas as primeiras histórias para a vigésima sexta e as últimas para a iteração seguinte.

Toda a estrutura de armazenamento de histórico foi implementada na vigésima sexta iteração. Foram priorizadas as últimas histórias relativas a histórico para a vigésima sétima e todas elas envolviam a alteração de relatórios que já haviam sido criados previamente e, eventualmente, a criação de novos relatórios. Na vigésima sétima iteração também foram feitos ajustes em funcionalidades ligadas ao histórico que já haviam sido implementadas na iteração anterior.

Deve-se notar que o primeiro tratamento de uma questão relativa ao histórico só ocorreu na vigésima primeira iteração, quando o sistema já havia entrado em produção e grande parte das funcionalidades já estava madura. Assim como ocorreu no caso dos relatórios, a implementação do histórico também se beneficiou da maturidade das

funcionalidades, permitindo que o processo fosse concluído com rapidez nas duas iterações finais do projeto.

Finalmente, é válido observar também que para implementar as funcionalidades associadas ao histórico, a equipe de desenvolvimento foi levada a alterar a arquitetura do sistema, pois a mesma não havia sido preparada para armazenar informações históricas. Nesta arquitetura não havia, por exemplo, tabelas destinadas ao armazenamento de informações históricas e nem procedimentos destinados a preenchê-las.

A implementação do histórico foi bem sucedida apesar da aparente fragilidade de uma arquitetura que não havia sido preparada para contemplá-lo. Através do uso da refatoração, a equipe de desenvolvimento conseguiu adaptar a arquitetura, inclusive fazendo alterações significativas nas bases de dados que já continham informações de produção armazenadas durante o primeiro ciclo de uso do sistema.

Este episódio demonstra que é possível trabalhar com uma arquitetura evolutiva e utilizar a refatoração freqüente para viabilizá-la, pois a utilização desta prática ao longo de todas as iterações do projeto foi importante para tornar a arquitetura simples e fácil de ser alterada. Sem isso, a implementação das funcionalidades do histórico poderia ter sido mais demorada e, eventualmente, inviabilizada.

Finalmente, vale notar também que a equipe modelou uma solução simples para o problema do histórico. Na realidade, duas alternativas foram consideradas no início das discussões, as quais envolveram todos os membros da equipe de desenvolvimento. Depois de dois dias de debates intensos e modelagens em conjunto utilizando quadro branco, a equipe chegou a uma terceira alternativa que se mostrou mais eficaz que as duas que vinham sendo debatidas até então. Além disso, se mostrou mais simples e

rápida de ser implementada. Neste caso, nota-se que a proximidade física entre os membros da equipe, o uso intenso do diálogo e a modelagem conjunta foram úteis mais uma vez para simplificar as soluções e acelerar a implementação das funcionalidades.

6.10 Documentação

No início do projeto, a equipe de desenvolvimento e o Comitê Olímpico fizeram uma priorização dos artefatos que deveriam ser gerados ao longo do projeto. Foram priorizados os seguintes itens:

- Diagrama de Classes;
- Javadoc;
- Plano de Testes de Aceitação;
- Manual do Usuário;
- Tutorial;
- Histórias;
- Modelo de Dados e
- Atas de Reuniões.

Estes artefatos eram produzidos de forma incremental, assim como ocorria com o código. Portanto, traziam novas informações a cada iteração, de modo a refletir as novas histórias implementadas no sistema.

No início de cada iteração, a equipe e o requerente produziam apenas os cartões contendo as histórias da iteração. Os desenvolvedores implementavam as funcionalidades até que nos últimos dois dias da iteração o redator técnico começasse a atualizar as documentações para refletir o que foi introduzido no código ao longo da iteração.

Ao atualizar os artefatos no final da iteração, a equipe evitava problemas de sincronização da documentação com o código. Caso as documentações fossem geradas no início da iteração, alterações no código para refletir ajustes solicitados pelo requerente demandariam alterações na documentação, tornando tais mudanças mais custosas. Deixando a atualização dos documentos para o final da iteração, os mesmos passavam a refletir um código mais estável o que minimizava a necessidade de alterações na documentação para refletir ajustes nas funcionalidades.

6.11 Análise de cada prática do XP no projeto

Nesta seção, iremos analisar a influência de cada prática do XP neste projeto.

Cliente Presente

Antes de iniciar o projeto, a consultoria responsável pelo desenvolvimento do Sistema de Aferições sugeriu que a equipe de desenvolvimento fosse alojada na sede do Comitê Olímpico, na mesma sala do requerente. Entretanto, o comitê não conseguiu disponibilizar um espaço para esta finalidade. Assim, a consultoria montou um escritório próximo à sede do comitê de modo que o requerente pudesse visitar a equipe de desenvolvimento com freqüência, bastando atravessar a rua.

Infelizmente, o mesmo não foi possível em relação ao requerente de TI, pois toda a área de Tecnologia da Informação do comitê ficava localizada em outro estado da Rússia. Para que o requerente de TI se encontrasse pessoalmente com a equipe, era necessário tomar um vôo com duração de uma hora, o que ele fazia uma vez a cada iteração.

As iterações sempre se iniciavam em uma terça-feira e terminavam em uma segunda-feira. Isso permitia que o requerente de TI tomasse um vôo na segunda-feira de manhã para acompanhar o encerramento da iteração e a validação das funcionalidades.

Passava uma noite na cidade da sede do comitê, e acompanhava o planejamento da iteração durante a terça, de modo que pudesse tomar um vôo de volta ao final do dia.

O relacionamento com o requerente que ficava próximo à equipe era mais freqüente. Ele se encontrava com os desenvolvedores quase diariamente, o que permitia diálogos que duravam alguns minutos na maioria das vezes, chegando a algumas horas em situações mais delicadas.

No início do projeto, as visitas do requerente não seguiam um planejamento bem definido. Ele visitava a equipe de forma *ad hoc* atendendo ao pedido dos desenvolvedores. Depois das primeiras iterações isso mudou. O líder da equipe de desenvolvimento passou a utilizar a reunião de planejamento no início de cada iteração para montar uma agenda de encontros com o requerente. Tal agenda indicava os dias da iteração nos quais o requerente visitaria a equipe, bem como o horário das visitas. O objetivo era fazer com que o requerente já alocasse espaço na sua agenda para a equipe e permitir que a equipe soubesse exatamente quando poderia contar com a presença do requerente para tirar dúvidas.

A proximidade física teve dois benefícios principais: permitiu que a equipe evitasse assumir premissas, fazendo validações freqüentes com o requerente; além disso, ajudou a estreitar os laços de confiança entre a equipe e o requerente, o que foi importante para assegurar a colaboração mútua entre as partes.

Durante as iterações, o desenvolvimento das histórias seguia uma ordem bem definida. A cada nova história, os desenvolvedores executavam as seguintes etapas:

1. A história era discutida em conjunto por todos os desenvolvedores que buscavam identificar o maior número possível de detalhes. Nessa fase, normalmente eles identificavam questões em aberto que eram anotadas

para serem perguntadas ao requerente assim que ocorresse a próxima visita dele.

2. Os desenvolvedores ajudavam o analista de testes a identificar testes de aceitação para a história, os quais eram anotados e validados mais tarde com o requerente. Este, por sua vez, eventualmente acrescentava outros testes de aceitação para a história.
3. Sempre que uma funcionalidade envolvia a criação ou alteração de uma tela do sistema (o que ocorria na maioria dos casos), a equipe desenhava a tela no quadro branco e validava com o requerente. Em seguida, o designer desenhava a tela utilizando ferramentas de tratamento de imagens, como o Photoshop. Este desenho tinha mais detalhes que aquele feito no quadro e permitia que o requerente fizesse ajustes visuais antes de a funcionalidade ser implementada. Feito isso, a mesma tela era desenhada em HTML (*Hypertext Markup Language*) puro, sem qualquer elemento dinâmico e mais uma vez era mostrada para a validação do requerente.
4. Finalmente, a equipe começava a implementar os aspectos dinâmicos da funcionalidade, adicionando código ao sistema.

Esse processo assegurava que os testes de aceitação fossem planejados antes da implementação da funcionalidade, o que levava os desenvolvedores a identificar cedo todos os fluxos aceitáveis e todas as exceções que poderiam ser geradas e, portanto, teriam que ser tratadas. Assim, quando a implementação começava, já era direcionada para cuidar dos fluxos aceitáveis, bem como para tratar as exceções, evitando que os usuários fossem surpreendidos por bugs do sistema caso tentassem executar operações

indevidas. Tais operações eram previstas no sistema e tratadas de modo que o usuário pudesse sempre receber uma mensagem adequada quando tentasse executá-las, ao invés de uma mensagem de erro genérica.

Além disso, a utilização de protótipos cada vez mais refinados era útil para permitir que o cliente visualizasse detalhes da funcionalidade de forma rápida, permitindo que ele pedisse ajustes e estes fossem incorporados com baixo custo. Ainda assim, era comum o requerente identificar ajustes após a funcionalidade ser completamente implementada. Entretanto, tais ajustes eram mínimos depois de todos os que eram efetuados durante a apresentação dos protótipos.

Cada um dos tipos de protótipos citados anteriormente costumava consumir uma média de poucos minutos até um máximo de uma ou duas horas para serem produzidos. Além disso, costumavam ser validados pelo requerente em poucos minutos.

Sem proximidade física, teria sido difícil utilizar este processo de trabalho, na medida em que as validações tenderiam a demorar mais para acontecer. Além disso, o uso de canais de comunicação menos ricos que os diálogos pessoais tornaria mais lenta a convergência das soluções.

Apesar de não ter tido acesso permanente ao requerente, visto que ele não estava permanentemente na sala em que ocorria o desenvolvimento, a proximidade dos escritórios permitiu um bom uso da prática de ter o cliente presente. Já no caso do requerente de TI, as relações eram um pouco mais complicadas devido à distância física.

Havia contato diário por telefone e email, mas esses canais freqüentemente levavam a dificuldades de entendimento e pequenos atritos no relacionamento. A distância também fazia com que o requerente de TI tivesse menor visibilidade do que

ocorria no dia-a-dia da equipe de desenvolvimento. Essa foi uma das razões pelas quais os relacionamentos de confiança eram menos fortes com o requerente de TI. A percepção geral dos desenvolvedores foi de que a confiança em uma prestação de serviço, como o desenvolvimento de um software, está profundamente associada à proximidade das partes e a visibilidade permanente do cliente sobre o que o prestador de serviço está fazendo diariamente pelo avanço do projeto.

Jogo do Planejamento

Como explicado anteriormente, o projeto teve 27 iterações, cada uma com duas semanas de duração, as quais se iniciavam em uma terça-feira e terminavam em uma segunda-feira. No primeiro dia de cada iteração havia uma reunião de planejamento que era usada para que o requerente priorizasse as histórias a serem implementadas na iteração. No último dia, o requerente validava todas as funcionalidades e percorria todo o plano de testes de aceitação elaborado e refinado ao longo da iteração que estava sendo finalizada.

Nas reuniões de planejamento, o requerente selecionava um conjunto de cartões contendo as histórias consideradas candidatas para serem implementadas na iteração. Às vezes, criava novos cartões em função do aprendizado obtido nas iterações anteriores. Em outras situações eliminava cartões que passassem a ser considerados desnecessários.

Os desenvolvedores estimavam os cartões candidatos. Isso era feito em conjunto, de modo que todos os desenvolvedores pudessem opinar sobre a quantidade de esforço necessária para cada cartão. Além disso, a estimativa era feita na presença do requerente e do requerente de TI, de modo que a equipe pudesse tirar dúvidas sobre os detalhes da funcionalidade. Finalmente, todos os cartões eram limitados a um máximo

de três dias de trabalho de um par de desenvolvedores. Cartões estimados acima deste número eram repassados de volta para o requerente para que fossem divididos.

De posse das estimativas, o requerente selecionava um conjunto de cartões que pudesse lhe gerar o máximo de benefício dentro do orçamento disponibilizado pela equipe de desenvolvimento. O orçamento era representado pela velocidade da equipe na iteração anterior. Para descobrir este valor, a equipe registrava seu progresso diariamente no **quadro de acompanhamento diário,** que foi criado por ela mesma e é explicado mais adiante na seção 6.13.

Feita a priorização, os cartões da iteração eram posicionados em um mural contendo três colunas: não iniciados, em andamento e finalizados (TELES, 2004, p.254). A ordem em que eram colocados no mural refletia a ordem definida pelo requerente para o desenvolvimento dos mesmos. Tal ordenação era respeitada pelos desenvolvedores ao longo da iteração, de modo que as funcionalidades mais importantes para o requerente fossem sempre as primeiras implementadas.

Ao final da reunião de planejamento, o líder da equipe de desenvolvimento escrevia uma ata contendo todos os assuntos discutidos na reunião, bem como a indicação dos cartões priorizados. Estas atas foram utilizadas para coletar inúmeras informações apresentadas neste estudo de caso.

A finalização de uma iteração era feita com a presença do requerente e do requerente de TI. Ambos utilizavam as funcionalidades implementadas na iteração e percorriam o plano de testes de aceitação para assegurar que todas as histórias tivessem sido implementadas corretamente. Nestes momentos, os requerentes normalmente pediam ajustes de última hora. Alguns deles eram implementados de imediato pelos

desenvolvedores, enquanto outros mais demorados acabavam se transformando em cartões para iterações seguintes.

Na reunião de finalização a equipe de desenvolvimento também fazia uma retrospectiva (TELES, 2004, p.265) com a participação dos requerentes. Trata-se de uma técnica estruturada para avaliação da iteração, de modo a indicar os aspectos da iteração que funcionaram bem e aqueles que precisariam ser melhorados. Ao final da retrospectiva, faz-se a priorização dos principais pontos que necessitam de melhorias e todos propõem ações para solucionar os problemas na iteração que se inicia. Tais ações ficam registradas no quadro branco de modo que a equipe possa recordar-se delas ao longo da iteração e colocá-las em prática. Este mecanismo se revelou útil para melhorar continuamente o sistema e a forma como a equipe se estruturava para implementá-lo.

Stand Up Meeting

A equipe de desenvolvimento realizava a reunião de stand up meeting diariamente por volta das 9:20h da manhã. Ela durava em torno de 10 a 20 minutos, nos quais se discutia tudo o que havia sido feito no dia anterior e se priorizava as atividades do dia que se iniciava. Além disso, o stand up meeting também era usado para que os desenvolvedores atualizassem o Quadro e Acompanhamento Diário.

Estas reuniões se mostraram úteis e foram feitas diariamente, durante todo o projeto. Muitos problemas eram resolvidos rapidamente nestas reuniões. Além disso, os desenvolvedores conseguiam ter uma boa noção do todo a partir dos relatos de seus colegas.

Programação em Par

A programação em par foi utilizada durante todo o projeto, embora tivesse havido resistências e dificuldades no início. Dentro da consultoria que implementou o

sistema de aferições, havia pessoas que não apoiavam a programação em par e acreditavam que ela colaboraria para diminuir a velocidade do projeto. Por isso, sugeriram que todas as práticas do XP fossem adotadas, com exceção da programação em par. A sugestão foi rejeitada pelo *coach* (BECK, 2000) (responsável técnico) da equipe de desenvolvimento.

Os desenvolvedores não estavam habituados a trabalhar em pares, portanto estranharam no início. Na verdade, não estavam habituados a trabalhar com o XP e tinham duvidavam sobre o seu funcionamento. Para contornar esta situação, o coach pediu que a equipe desse um crédito de seis semanas para o XP. Ou seja, durante seis semanas a equipe utilizaria as práticas do XP sem restrições. Em seguida, todos juntos fariam uma análise das práticas e decidiriam se continuariam a usá-las ou se adotariam outras estratégias de desenvolvimento. O coach assumiu o compromisso de adotar a estratégia que fosse definida pela equipe ao final deste período, desde que a equipe ao menos experimentasse as práticas do XP durante essas seis semanas.

Apesar do desconforto inicial, os desenvolvedores aprenderam rapidamente a trabalhar em pares e a se adaptar a seus colegas. Isso foi facilitado pelo fato de que todos os desenvolvedores se conheciam antes deste projeto e já tinham trabalhado juntos durante anos em projetos anteriores. Assim, os laços de amizade contribuíram para que eles alcançassem rapidamente um bom relacionamento ao trabalharem em par.

O uso da programação em par parece ter contribuído para que a equipe produzisse menos bugs. Além disso, os pares avançavam rapidamente de forma consistente. O líder de projeto, que estava entre as pessoas que se opuseram inicialmente à programação em par, se transformou em defensor da idéia após observar os resultados da primeira iteração. Ao longo do tempo, ninguém mais levantou dúvidas

sobre a programação em par e ela permaneceu em uso durante todo o projeto. Passadas as seis semanas de experimentação solicitadas pelo coach, a equipe de desenvolvimento decidiu manter a programação em par, bem como todas as demais práticas do XP.

Desenvolvimento Orientado a Testes

O desenvolvimento orientado a testes normalmente representa a parte técnica mais difícil do XP, especialmente para pessoas habituadas a trabalhar dentro da forma tradicional de desenvolvimento, isto é, implementar e depois testar. Com esta equipe não foi diferente. Os desenvolvedores não possuíam experiência, nem conhecimentos sobre como implementar testes automatizados.

Desde o início, eles conseguiram criar testes automatizados para as classes que faziam acesso ao banco de dados. Entretanto, não conseguiram fazer o mesmo para as demais classes do sistema. Portanto, houve deficiência na automação dos testes. Felizmente, o analista de testes executava testes manuais com boa freqüência, o que evitava uma incidência elevada de bugs.

A deficiência nos testes fez com que a equipe produzisse um número de bugs considerado significativo a cada iteração. Em média, ela produziu mais de dez bugs por iteração. Tais bugs costumavam ser detectados com rapidez pelo analista de testes, mas é provável que o número de bugs fosse menor e o trabalho de analista de testes menos intenso caso se tivesse conseguido implementar testes automatizados em todas as partes do sistema.

Refatoração

A equipe aprendeu rapidamente a utilizar a refatoração em seu dia-a-dia. Isso era facilitado pela utilização de um ambiente de desenvolvimento que disponibilizava

diversos tipos de refatorações de forma automatizada (IntelliJ 3.0), diminuindo a chance de que as refatorações produzissem erros.

A deficiência nos testes automatizados, entretanto, fez com que as refatorações fossem desprovidas de um mecanismo de proteção essencial. Por isso, houve situações em que refatorações geraram bugs que não foram detectados rapidamente. Porém, na maioria dos casos, o suporte à refatoração embutido no IntelliJ assegurou que as refatorações ocorressem de forma segura.

A arquitetura do sistema evoluiu de forma consistente com a utilização da refatoração. De um modo geral, a equipe de desenvolvimento foi capaz de adaptá-la rapidamente às necessidades das funcionalidades que iam surgindo no sistema.

Durante um dia de trabalho, era comum os desenvolvedores executam uma grande quantidade de pequenas refatorações à medida que evoluíam. Poucas eram as situações em que os desenvolvedores faziam grandes refatorações que consumiam muito tempo e geravam grandes impactos na arquitetura. Mesmo nestes casos, a equipe se surpreendeu diversas vezes com a rapidez com que algumas refatorações foram executadas.

Código Coletivo

A prática de código coletivo foi usada durante todo o projeto e não apresentou problemas. Tornar o código coletivo permitiu que a equipe avançasse com agilidade, na medida em que não era necessário esperar por outros desenvolvedores sempre que havia a necessidade de editar um arquivo do sistema. Além disso, o revezamento entre as diversas funcionalidades permitiu que os desenvolvedores obtivessem vivência em todos os aspectos do projeto.

Código Padronizado

Nos primeiros dias do desenvolvimento a equipe estabeleceu um padrão para o código do sistema. Isso foi facilitado pela utilização da prática de código coletivo e da programação em par. Ambas ajudaram a assegurar que os desenvolvedores aderissem aos padrões. Por sua vez, a padronização do código ajudou os desenvolvedores a manter o código coletivo e a trabalhar em pares ao longo do desenvolvimento.

Design Simples

A cada iteração, a equipe de desenvolvimento implementava novas características na arquitetura do sistema que fossem suficientes para comportar apenas as funcionalidades da iteração. Ou seja, mesmo que cartões de iterações futuras fossem conhecidos, a equipe não criava mecanismos para sustentar a construção futura dos mesmos.

Isso se mostrou útil visto que os modelos de aferição e planejamento de treinos sofreram fortes mudanças ao longo do projeto. Se a equipe tivesse investido muito cedo na criação de uma infra-estrutura para suportar cartões futuros, grande parte do esforço acabaria se perdendo e o comitê demoraria mais para receber as funcionalidades consideradas prioritárias.

Integração Contínua

O processo de integração contínua era apoiado pela automação de builds e pela utilização de um repositório. Os builds eram automatizados com a utilização da ferramenta Ant[7], enquanto o repositório era de responsabilidade do CVS[8]. Além disso, a equipe mantinha uma máquina exclusivamente para integração.

Antes de iniciar qualquer atividade de desenvolvimento, os pares recuperavam todos os arquivos do projeto a partir do repositório (*check out*) armazenando-os em um

[7] http://ant.apache.org/
[8] https://www.cvshome.org/

diretório local de suas estações de trabalho. Em seguida implementavam alterações, adições e remoções no código. De tempos em tempos, os pares iam até a máquina de integração para integrar o código que estavam editando com o código gerado pelos demais desenvolvedores que já estivessem armazenados no repositório.

Este processo funcionou bem durante todo o projeto. Às vezes ocorriam conflitos de integração, mas tais situações eram raras. Quando ocorriam, costumavam ser corrigidos com rapidez porque todos os desenvolvedores se sentavam próximos uns dos outros. Assim, os pares chamavam seus colegas para ajudar a solucionar os conflitos rapidamente, o que normalmente dava bons resultados.

Metáfora

No início do projeto, a equipe e o comitê desenvolveram algumas metáforas para se referirem a aspectos chaves dos processos de aferição. Em particular, pode-se citar a elaboração do conceito de gavetas para cada atleta, dentro das quais armazenavam-se fichas contendo os resultados das aferições, bem como os planos de treinos. Esta metáfora era usada nas conversas com o requerente e as classes do sistema eram nomeadas de acordo com ela.

O uso de metáforas se revelou útil para facilitar a comunicação dos conceitos mais importantes do projeto. Para criar tais metáforas, a equipe procurou imaginar como os processos modelados para o sistema seriam executados no dia-a-dia caso não existisse computador. Daí surgiu a idéia de utilizar gavetas, fichas etc.

Ritmo Sustentável

A equipe de desenvolvimento trabalhou de 9h às 18h durante praticamente todos os dias do projeto. Houve apenas cinco ocasiões onde isso não aconteceu. Nestes casos, houve necessidade de permanecer no escritório até mais tarde e em duas situações foi

preciso que parte dos desenvolvedores fossem ao escritório no fim-de-semana. De um modo geral isso ocorreu apenas quando o sistema estava sendo colocado em produção e teve como objetivo dar suporte à equipe de TI do comitê para que fosse capaz de instalar o sistema corretamente nos servidores de produção.

Releases Curtos

O Comitê Olímpico utiliza os serviços de duas empresas terceirizadas na área de Tecnologia da Informação. Uma cuida da manutenção de todos os sistemas do comitê, enquanto a outra é responsável por toda a infra-estrutura de hardware.

Para que um projeto entre em produção, é necessário acionar estas duas empresas e fazer o sistema ser homologado por elas. Na prática isso significa fazer com que o sistema passe por um conjunto de formalidades trabalhosas e demoradas. Por isso, a prática de releases curtos não foi adotada neste projeto, embora a equipe de desenvolvimento quisesse utilizá-la.

Houve uma entrada em produção na décima nona iteração. Depois, foram feitos acertos e novas funcionalidades foram adicionadas nas iterações seguintes. Estas entraram em produção mais rapidamente. A impossibilidade de utilizar a prática de releases curtos foi prejudicial por ter privado a equipe de desenvolvimento do feedback de uma comunidade maior de usuários que poderiam ajudar a refinar ainda mais o sistema.

6.12 Quadro de Acompanhamento Diário

O quadro de acompanhamento diário é uma ferramenta de monitoramento do projeto que foi criada pela equipe de desenvolvimento ao longo deste projeto. Como já vimos, o processo de planejamento do Extreme Programming envolve os seguintes conceitos:

- História;
- Cartão;
- Estimativa;
- Dia ideal;
- Dia real;
- Velocidade e
- Tarefas extras.

Uma **história** é uma pequena funcionalidade que pode ser desenvolvida por uma dupla de desenvolvedores em poucos dias. Um software é formado por um conjunto de histórias, as quais são escritas pelo cliente, com suas próprias palavras, em pequenos **cartões**.

Cada história é estimada pela equipe de desenvolvimento, que registra a **estimativa** no canto superior esquerdo dos cartões (mais tarde, o realizado é colocado no campo superior direito). Desta forma, observando cada cartão, o cliente é capaz de identificar a história e seu respectivo custo na forma de tempo de desenvolvimento.

As estimativas utilizam como unidade o conceito de **dia ideal**. Se uma história é estimada em um dia ideal, isso significa que uma dupla de desenvolvedores (devido ao uso da programação em par no XP) será capaz de desenvolver a história em um dia de trabalho, desde que não sejam interrompidos para executar outras atividades. Portanto, um dia ideal representa um dia de trabalho no qual o par pode se dedicar integralmente apenas ao desenvolvimento de histórias, sem se preocupar em atender telefonemas, participar de reuniões, corrigir bugs etc (BECK, 2000).

Ao fazer uma estimativa, o desenvolvedor projeta um mundo ideal, no qual ele possa se abstrair de interrupções e dedicar-se plenamento ao desenvolvimento da funcionalidade. Portanto, a estimativa não leva em conta fatores externos, nem qualquer tipo de interrupção. O que se busca é o melhor caso. Entretanto, infelizmente o

desenvolvedor vive em um **dia real**, no qual existem interrupções que afetam a quantidade de histórias produzidas.

Para lidar com isso, utiliza-se o conceito de **velocidade** da iteração. Um dia real é composto por horas dedicadas ao desenvolvimento de histórias e horas dedicadas a **tarefas extras**, as quais representam um overhead para o projeto. A velocidade indica a quantidade de horas efetivamente úteis, ou seja, que realmente levaram à implementação de novas histórias. Portanto, a velocidade procura indicar quantos dias ideais estiveram contidos dentro de uma determinada iteração.

Para compreender estes conceitos, pode-se utilizar um exemplo. Suponhamos uma iteração de duas semanas e uma equipe de quatro desenvolvedores. Neste caso:

1 iteração = 2 semanas = 10 dias úteis

4 desenvolvedores = 2 pares

1 par / dia trabalhando exclusivamente em histórias = **1 dia ideal**

2 pares / dia trabalhando exclusivamente em histórias = 2 dias ideiais

10 dias úteis x 2 dias idéias = 20 dias ideais

A cada iteração deste projeto, se tudo correr de maneira perfeita, a equipe será capaz de implementar histórias que somem um máximo de 20 dias ideais. Este é o limite máximo da iteração e só é alcançado caso não existam tarefas extras ao longo da mesma.

Se em uma determinada iteração a equipe tiver consumido 4 dias de um par para executar atividades extras, isso significará que a quantidade de dias ideais da iteração terá sido apenas 16, que representa o resultado de 20 (limite máximo) – 4 (tarefas

extras). O valor 16, ou seja, a quantidade de dias ideais da iteração representa a sua velocidade.

As histórias são estimadas em número de dias ideais. Por sua vez, a velocidade indica a quantidade de dias ideais de uma iteração. Portanto, este valor é utilizado pelo cliente para indicar a quantidade de histórias que serão alocadas a cada iteração. Ele poderá alocar histórias até que a soma de suas estimativas alcance a velocidade da iteração.

A velocidade não é um valor constante. Uma iteração pode ter uma quantidade maior ou menor de tarefas extras em relação a outra. Para que o cliente e a equipe possam se planejar adequadamente, é necessário que a equipe seja capaz de fornecer a velocidade da iteração que se inicia com a maior precisão possível.

A equipe verifica quantos dias ideais foram obtidos na iteração anterior e assume que esta quantidade se repetirá na iteração corrente. Entretanto, para fazer isso, é necessário que a equipe seja capaz de medir a velocidade de cada iteração. Em outras palavras, é necessário que ela monitore o tempo dedicado às funcionalidades e o tempo dedicado às tarefas extras.

O **quadro de acompanhamento diário** é uma ferramenta que pode ser utilizada de maneira simples para monitorar os acontecimentos da iteração. Ele é dividido em duas partes: histórias e tarefas extras, como ilustrado na figura 4. A parte associada às histórias é utilizada para registrar o tempo gasto no desenvolvimento de cada história da iteração, enquanto a outra parte é usada para indicar o tempo alocado às atividades extras.

A tabela contém as seguintes colunas:

- Estimativa – utilizada apenas na parte associada às histórias. Armazena a quantidade de dias ideais estimados para cada história da iteração. As histórias são listadas nas linhas da tabela em ordem de prioridade.
- Dias da iteração – considerando-se uma iteração de duas semanas, haverá dez dias úteis de trabalho. Para cada dia, do primeiro ao décimo, cria-se uma coluna.
- Total – indica a quantidade total de tempo gasta com uma determinada história ou tarefa extra.

No início da iteração, o quadro de acompanhamento diário se parece com o exemplo da figura 6.1. Apenas as histórias são preenchidas e suas respectivas estimativas.

História	Estimativa	1	2	3	4	5	6	7	8	9	10	Total	Desvio
1. Listar vôos entre duas cidades	3												
2. Ordenar por preço	1,75												
3. Ordenar por números de conexões	3												
4. Ordenar por número de escalas	1,75												
5. Buscar tarifa de um vôo	2												
6. Buscar tarifa mais barata de um trecho	2,5												
7. Buscar 10 tarifas mais baratas de um trecho	2												
8. Marcar assento de um passageiro	1												
Total	17												
Tarefas Extras													
Total													

Figura 6.1: quadro no início da iteração.

Ao longo da iteração, a equipe atualiza o quadro diariamente, de modo que, no meio da iteração, o quadro irá se parecer com o exemplo da figura 6.2.

História	Estimativa	1	2	3	4	5	6	7	8	9	10	Total	Desvio
1. Listar vôos entre duas cidades	3	1	0,5	1								2,5	-0,5
2. Ordenar por preço	1,75	1	0,75									1,75	0
3. Ordenar por números de conexões	3		0,25	1	0,75	1						3	0
4. Ordenar por número de escalas	1,75				1	0,5						1,5	-0,25
5. Buscar tarifa de um vôo	2					0,5						0,5	-1,5
6. Buscar tarifa mais barata de um trecho	2,5												
7. Buscar 10 tarifas mais baratas de um trecho	2												
8. Marcar assento de um passageiro	1												
Total	17	2	1,5	2	1,75	2						9,25	-7,75
Tarefas Extras													
Reunião de padronização do banco de dados			0,5									0,5	
Ajustes no link da internet					0,25							0,25	
Total		0	0,5	0	0,25	0						0,75	

Figura 6.2: quadro no meio da iteração.

Ao final da iteração, o quadro irá se assemelhar aos exemplos das figuras 6.3 ou 6.4.

História	Estimativa	1	2	3	4	5	6	7	8	9	10	Total	Desvio
1. Listar vôos entre duas cidades	3	1	0,5	1								2,5	-0,5
2. Ordenar por preço	1,75	1	0,75									1,75	0
3. Ordenar por números de conexões	3		0,25	1	0,75	1	0,5					3,5	0,5
4. Ordenar por número de escalas	1,75				1	0,5						1,5	-0,25
5. Buscar tarifa de um vôo	2					0,5	0,5	1				2	0
6. Buscar tarifa mais barata de um trecho	2,5							1	1	0,5		2,5	0
7. Buscar 10 tarifas mais baratas de um trecho	2								0,75	1,5	1	3,25	1,25
8. Marcar assento de um passageiro	1											0	-1
Total	17	2	1,5	2	1,75	2	1	2	1,75	2	1	17	0
Tarefas Extras													
Reunião de padronização do banco de dados			0,5									0,5	
Ajustes no link da internet					0,25							0,25	
Falta de luz							1					1	
Deployment em produção									0,25			0,25	
Correção do bug #28											1	1	
Total		0	0,5	0	0,25	0	1	0	0,25	0	1	3	

Figura 6.3: a quantidade de dias ideais permaneceu estável, mas a equipe não implementou todas as funcionalidades.

Na figura 6.3, a equipe herdou uma velocidade de 17 dias ideais, valor que acabou se repetindo ao final da iteração. Entretanto, algumas funcionalidades consumiram mais esforço que o previsto. O desvio entre a estimativa e o realizado também é armazenado na tabela, na extrema direita. Essa informação é útil para que a

equipe estude as funcionalidades onde houve erros de estimativa e procure compreender o que os gerou, de modo a melhorar as estimativas futuras.

História	Estimativa	1	2	3	4	5	6	7	8	9	10	Total	Desvio
1. Listar vôos entre duas cidades	3	1	0,5	1								2,5	-0,5
2. Ordenar por preço	1,75	1	0,75									1,75	0
3. Ordenar por números de conexões	3		0,25	1	0,75	1	0,5					3,5	0,5
4. Ordenar por número de escalas	1,75				1	0,5						1,5	-0,25
5. Buscar tarifa de um vôo	2					0,5	0,5	1				2	0
6. Buscar tarifa mais barata de um trecho	2,5								1	1		2	-0,5
7. Buscar 10 tarifas mais baratas de um trecho	2								0,75	1		1,75	-0,25
8. Marcar assento de um passageiro	1										1	1	0
Total	17	2	1,5	2	1,75	2	1	1	1,75	2	1	**16**	**-1**
Tarefas Extras													
Reunião de padronização do banco de dados			0,5									0,5	
Ajustes no link da internet					0,25							0,25	
Falta de luz							1	1				2	
Deployment em produção									0,25			0,25	
Correção do bug #28											1	1	
Total		0	0,5	0	0,25	0	1	1	0,25	0	1	**4**	

Figura 6.4: a quantidade de dias ideais foi alterada, mas a equipe implementou todas as funcionalidades.

No exemplo da figura 6.4, a velocidade cai de 17 para 16 porque a quantidade de tarefas extras cresceu em relação à iteração anterior. Entretanto, a equipe errou nas estimativas, desta vez, para cima. Assim, apesar de a velocidade ter diminuído, a equipe foi capaz de implementar todas as histórias previstas.

O quadro de acompanhamento diário é desenhado a cada iteração em um quadro branco, de modo que seja permanentemente visível e acessível por todos os membros da equipe. A atualização é feita de forma rápida durante o stand up meeting. Cada par escreve no quadro a quantidade de tempo gasta no dia anterior com cada história ou atividade extra. Para isso, utiliza-se a regra de que o máximo de tempo alocado a uma história ou atividade extra em um dia é 1 dia ideal. Por sua vez, o mínimo é 0,25 dia ideal que equivale a duas horas de trabalho de um par.

Fazendo isso, o quadro não fornece precisão absoluta sobre o que foi feito na iteração, mas o processo de registro da informação é simplificado para diminuir ou

eliminar resistências a sua utilização. O objetivo é facilitar para que se obtenha, ao menos, um bom retrato da iteração, mesmo que a precisão seja ligeiramente comprometida.

O preenchimento do quadro durante o *stand up meeting* também é utilizado como forma de assegurar que a equipe mantenha a disciplina em conjunto. Quando um desenvolvedor possui uma ficha para preencher tudo o que faz durante o dia e o tempo alocado a cada atividade, por exemplo, é comum que ele se esqueça de preencher ou simplesmente se recuse a fazê-lo. Por outro lado, quando todos os desenvolvedores criam o hábito de preencherem o quadro, juntos, manter a disciplina torna-se mais fácil. Isso se comprovou no projeto, visto que os desenvolvedores efetivamente atualizaram o quadro todos os dias, durante todo o projeto.

Ao final da iteração, a equipe pode registrar as informações do quadro para que possam ser referenciadas futuramente. Isso pode ser feito com o uso de uma planilha eletrônica ou um editor de texto, por exemplo. Assim, os membros do projeto são capazes, entre outras coisas, de verificar todas as histórias implementadas no passado, o tempo efetivamente gasto em cada uma, os desvios em relação às estimativas e as tarefas extras realizadas. Este estudo de caso se baseou, em grande parte, nas informações coletadas no quadro de acompanhamento diário, que foram registradas a cada iteração, nas atas das reuniões do jogo do planejamento.

Retrospectivas

As informações extraídas do quadro ao final da iteração atendem a vários propósitos. O mais básico é indicar a velocidade que será adotada na iteração seguinte. Entretanto, o quadro também é usado para fornecer subsídios à retrospectiva.

Ela representa um processo estruturado de reflexão da equipe de desenvolvimento, executado com a participação do cliente, sempre que possível. A primeira etapa da retrospectiva consiste em enumerar e analisar os pontos positivos da iteração para que eles sejam lembrados pela equipe na iteração seguinte e sirvam de inspiração para manter as práticas que vêem funcionando.

Em seguida, a equipe enumera e analisa os pontos da iteração que precisam melhorar. Todas as pessoas indicam os problemas que vivenciaram e as conseqüências dos mesmos. A equipe prioriza em conjunto os pontos a melhorar, de modo a indicar quais os dois ou três mais críticos (não mais que isso). Para cada um destes problemas, os participantes da retrospectiva propõem e discutem ações que possam resolvê-los na iteração seguinte (TELES, 2004).

As informações do quadro de acompanhamento diário são importantes durante a retrospectiva, pois geram pistas para identificar os problemas mais críticos, como por exemplo uma iteração com excesso de atividades extras, ou uma história que teve um desvio muito acentuado em relação ao estimado, ou um conjunto de histórias que acabou não sendo implementado. A retrospectiva é um mecanismo de melhoria contínua que foi utilizado com sucesso ao longo deste projeto

6.13 Considerações finais

O Sistema de Aferições é formado por 70 mil linhas de código, 1.214 classes e 5.728 métodos que foram gerados por uma equipe de 4 desenvolvedores durante um período de 55 semanas. Houve aproximadamente 550 dias úteis disponíveis para a equipe de desenvolvimento, dos quais 322 foram efetivamente utilizados para a implementação de funcionalidades. Os demais 228 dias foram consumidos em tarefas

extras ao longo do projeto. Portanto, apenas 58% do tempo foi efetivamente consumido na implementação de novas funcionalidades.

Ao final do projeto, o sistema passou por um período de seis meses de garantia. Neste período, caso algum erro fosse detectado, a consultoria que desenvolveu o Sistema de Aferições seria responsável por fazer correções. No total, a equipe de desenvolvimento teve que corrigir apenas três bugs ao longo dos seis meses de garantia, sendo que o primeiro só foi identificado depois de quatro meses e meio de utilização do sistema em produção.

O sistema foi utilizado no primeiro ciclo por 97% do seu público alvo. Este número foi considerado alto pelo comitê, visto que outros sistemas sazonais utilizados pelos atletas possuem taxa de utilização média de apenas 60% do público alvo. Diversos atletas fizeram comentários sobre o sistema e manifestaram a opinião de que o Sistema de Aferições era “fácil e bonito”, ao contrário de outros sistemas que os atletas tinham dificuldades de utilizar e acabavam deixando de lado.

O Comitê Olímpico demonstrou elevada satisfação com o resultado final e elogiou a equipe de desenvolvimento inúmeras vezes ao longo do projeto e mais enfaticamente no final do mesmo. Na última retrospectiva realizada, o requerente manifestou a opinião de que, após este projeto, estava convencido da necessidade de que ele acompanhasse a equipe ao longo de todo o projeto e tivesse a oportunidade de alterar o escopo ao longo do tempo de modo a melhorar cada vez mais as funcionalidades, tornando-as mais intuitivas para os usuários.

O resultado demonstrou que a utilização das práticas do XP contribuiu para que o sistema solucionasse as necessidades reais do comitê e fizesse isso de maneira

elegante e fácil de ser utilizada. Entretanto, não foi possível fazer inferências em relação à influência da utilização do XP sobre o prazo e o custo do projeto.

Antes de iniciar o projeto, o comitê forçou a consultoria a trabalhar com um prazo excessivamente curto para a implementação do projeto. Com o tempo, o comitê percebeu a necessidade de alterar o escopo para incorporar mudanças no modelo conceitual de aferições e planejamento dos treinos, bem como outras mudanças menos significativas. Desta forma, o escopo da solução final acabou sendo bastante diferente do original, apesar de o problema a ser resolvido ter se mantido estável ao longo de todo o projeto.

Os refinamentos consumiram um tempo que não havia sido previsto no cronograma inicial. Portanto, o projeto consumiu mais tempo que o imaginado originalmente e, conseqüentemente, custou mais caro. Porém, é difícil traçar um paralelo entre o resultado final e o previsto no que se refere a custo e prazo visto que o escopo foi profundamente alterado.

É possível levantar a hipótese de que a flexibilidade do escopo, normalmente encontrada em projetos XP, tenha sido responsável pela elevação no prazo e no custo do projeto. Por outro lado, caso o escopo original tivesse sido mantido até o final, é possível que o sistema fosse entregue no prazo e no custo previsto, porém não fosse capaz de atender às necessidades dos seus usuários.

Se ele se revelasse de pouca utilidade, o fato de alcançar o prazo e orçamento previsto teria sido irrelevante, pois os problemas dos usuários continuariam sem solução. Esperamos que outros trabalhos futuros possam ser mais conclusivos em relação à influência das práticas do XP nos prazos e custos dos projetos de software.

7 CONCLUSÃO

Este trabalho fez uma análise dos problemas que tradicionalmente afetam os projetos de software, tais como atrasos, gastos superiores aos orçamentos e funcionalidades que não solucionam os problemas dos usuários. O trabalho propôs a adoção das práticas e valores do XP como uma alternativa viável para a resolução destes problemas em diversos projetos de software.

Através da revisão de literatura, procurou-se estabelecer bases teóricas significativas para a adoção de cada uma das práticas e valores do XP. Foi mostrado que embora tais práticas sejam controversas em um primeiro momento, existem razões sólidas pelas quais podem funcionar de fato.

O estudo de caso apresentou um projeto que consumiu mais de um ano de trabalho com uma equipe de quatro desenvolvedores. Tal projeto utilizou todas as práticas do Extreme Programming. A avaliação dos resultados mostrou que o projeto foi capaz de gerar um conjunto de funcionalidades que atenderam às necessidades dos usuários de forma adequada.

A equipe foi capaz de conduzir o projeto com baixo nível de estresse e o relacionamento com os usuários se revelou positivo. Foi possível estabelecer uma forte relação de confiança entre a equipe de desenvolvimento e seus usuários. Além disso, o sistema apresentou elevada integridade, depois de sofrer inúmeros aprimoramentos com base no feedback gerado ao final de cada iteração.

No total, o projeto consumiu 70 mil linhas de código que foram geradas por 4 desenvolvedores durante um período de 55 semanas. Houve aproximadamente 550 dias úteis disponíveis para a equipe de desenvolvimento, com aproveitamento de 58% deste

tempo para a implementação de novas funcionalidades. Durante o período de seis meses de garantia após a conclusão do projeto, foram identificados apenas três bugs.

O sistema, que tem utilização sazonal, foi usado no primeiro ciclo por 97% do seu público alvo. Trata-se de um número elevado, visto que a outros sistemas sazonais do cliente têm taxa de utilização média de apenas 60% do público alvo. Os usuários do sistema o consideraram “fácil e bonito”, o que gerou o elevado grau de utilização observado.

Devido à intensidade das mudanças que ocorreram no escopo do projeto, não foi possível estabelecer os efeitos da adoção do XP sobre o prazo e o custo do projeto. Sendo assim, espera-se que trabalhos futuros sejam capazes de obter outros dados de modo a comparar com os obtidos neste estudo de caso, em particular no que se refere ao prazo e ao custo.

REFERÊNCIAS

AMBLER, Scott W. **Writing robust Java code:** the AmbySoft Inc. coding standards for Java. 2000. Disponível em: http://www.ambysoft.com/javaCodingStandards.pdf. Acesso em: 23/12/2004.

ASTELS, David. **Test-driven development:** a practical guide. 1. ed. Upper Saddle River, NJ: Prentice Hall PTR, 2003. 562 p.

BECK, Kent. **Extreme Programming explained:** embrace change. 1. ed. Reading, MA: Addison-Wesley, 2000. 190 p.

BECK, Kent. **Test-driven development:** by example. 1. ed. Boston: Addison-Wesley, 2003. 240 p.

BECK, Kent; ANDRES, Cynthia. **Extreme Programming explained:** embrace change. 2. ed. Upper Saddle River: Addison-Wesley, 2005. 189 p.

BECK, Kent; FOWLER, Martin. **Planning Extreme Programming**. 1. ed. Boston: Addison-Wesley, 2001. 139 p.

BOEHM, Barry; TURNER, Richard. **Balancing agility and discipline:** a guide for the perplexed. 1. ed. Boston: Addison-Wesley, 2003. 266 p.

BROOKS, Frederick P. **The mythical man-month:** essays on software engineering, 20th anniversary edition. 2. ed. Reading, MA: Addison-Wesley, 1995. 322 p.

BROOKS, Frederick P. **No silver bullet:** essences and accidents of Software Engineering. IEEE. Computer, v. 20, n. 4, 1987. p. 10-19.

BRYANT, Antony,'It's engineering Jim ... but not as we know it': Software Engineering - solution to the software crisis, or part of the problem? In: International Conference on Software Engineering, 22., 2000, Limerick, Ireland. **Anais...** New York, NY: ACM Press, 2000. p. 78-87.

BUHRER, Koni. **From craft to science: searching for first principles of software development**. The Rational Edge, 2000. Disponível em: http://www-106.ibm.com/developerworks/rational/library/content/RationalEdge/dec00/FromCrafttoScienceDec00.pdf. Acesso em: 23/12/2004.

COCKBURN, Alistair. **Agile software development**. 1. ed. Boston: Addison-Wesley, 2002. 278 p.

CONSTANTINE, Larry L. **The peopleware papers:** notes on the human side of software. 1. ed. Upper Saddle River: Prentice Hall PTR, 2001. 346 p.

DEMARCO, Tom. **Slack:** getting past burnout, busywork, and the myth of total efficiency. 1. ed. New York: Broadway Books, 2001. 227 p.

DEMARCO, Tom; BOEHM, Barry. **The agile methods fray**. IEEE Computer, v. 35, n. 6, 2002. p. 90-92.

DEMARCO, Tom; LISTER, Timothy R. **Peopleware:** productive projects and teams 2nd ed. 2. ed. New York, NY: Dorset House Pub. Co, 1999. 245 p.

DEMARCO, Tom; LISTER, Timothy R. **Peopleware:** productive projects and teams. 1. ed. New York, NY: Dorset House Pub. Co, 1987. 188 p.

DIJKSTRA, Edsger W. **The humble programmer**. ACM. Communications of the ACM, v. 15, n. 10, 1972. p. 859-866.

DRUCKER, Peter. **Desafio gerenciais para o século XXI**. 1. ed. São Paulo: Pioneira, 1999. 168 p.

EISCHEN, Kyle. **Software development:** an outsider's view. IEEE. Computer, v. 35, n. 5, 2002. p. 36-44.

EMAM, Khaled E. **Finding success in small software projects**. Cutter Consortium Agile Project Management. Executive Report, v. 4, n. 11, 2003.

FOWLER, Martin. **Refactoring:** improving the design of existing code. Upper Saddle River, NJ: Addison-Wesley, 2000. 421 p.

HIGHSMITH, James A. **Agile software development ecosystems**. 1. ed. Boston: Addison-Wesley, 2002. 404 p.

HUNT, Andrew; THOMAS, David. **The pragmatic programmer:** from journeyman to master. 1. ed. Upper Saddle River: Addison-Wesley, 2000. 321 p.

HUNT, Andrew; THOMAS, David. **Pragmatic unit testing:** in Java with JUnit. 1. ed: The Pragmatic Programmers, 2003. 176 p.

ISBELL, Douglas; HARDIN, Mary; UNDERWOOD, Joan. **Mars Climate orbiter team finds likely cause of loss**. NASA Jet Propulsion Laboratory, 1999. Disponível em: http://mars.jpl.nasa.gov/msp98/news/mco990930.html. Acesso em: 23/12/2004.

JACOBSON, Ivar; BOOCH, Grady; RUMBAUGH, James. **The Unified software development process:** the complete guide to the Unified Process from the original designers. 1. ed. Reading, MA: Addison-Wesley, 1999. 463 p.

JEFFRIES, Ron; ANDERSON, Ann; HENDRICKSON, Chet. **Extreme Programming installed**. 1. ed. Boston: Addison-Wesley, 2001. 265 p.

JOHNSON, Jim. **"ROI, It's your job"**. Published Keynote Third International Conference on Extreme Progrmming, Alghero, Italy, May 26-29, 2002. Disponível em: http://www.xp2003.org/talksinfo/johnson.pdf. Acesso em: 28/12/2004.

KRUTCHTEN, Philippe. **The nature of software: what's so special about software engineering**. The Rational Edge, 2001. Disponível em: http://www-106.ibm.com/developerworks/rational/library/4700.html. Acesso em: 23/12/2004.

MASI, Domenico de. **O ócio criativo**. 1. ed. Rio de Janeiro: Sextante, 2000. 336 p.

MCBREEN, Pete. **Software craftsmanship:** the new imperative. 1. ed. Upper Sadle River: Addison-Wesley, 2002. 187 p.

POPPENDIECK, Mary; POPPENDIECK, Tom. **Lean software development:** an agile toolkit. 1. ed. Upper Saddle River, NJ: Addison-Wesley, 2003

PRESSMAN, Roger S. **Software engineering:** a practitioner's approach. 4. ed. New York: McGraw-Hill, 1997. 885 p.

SENGE, Peter M. **A quinta disciplina:** arte e prática da organização que aprende. 12. ed. São Paulo: Best Seller, 2002. 491 p.

SMITH, Adam. **A riqueza das nações:** investigação sobre a sua natureza e suas causas. São Paulo: Nova Cultura, 1996. 479 p.

SORID, Dan. **Human error doomed Mars Climate Orbiter**. Space.com, 1999. Disponível em: http://www.space.com/news/orbiter_error_990930.html. Acesso em: 23/12/2004.

TEIXEIRA, Sérgio. **O caçador de defeitos**. Exame, n. 814, 2004.

TELES, Vinícius Manhães. **Extreme Programming:** aprenda como encantar seus usuários desenvolvendo software com agilidade e alta qualidade. 1. ed. São Paulo: Novatec, 2004. 316 p.

THE STANDISH GROUP INTERNATIONAL, Inc. **Extreme chaos**. The Standish Group International, Inc, 2001. Disponível em: http://www.standishgroup.com/sample_research/PDFpages/extreme_chaos.pdf. Acesso em: 23/12/2004.

TOFFLER, Alvin. **A terceira onda:** a morte do industrialismo e o nascimento de uma nova civilização. 26. ed. Rio de Janeiro: Record, 2001. 491 p.

WEINBERG, Gerald M. **The psychology of computer programming**. 1. ed. New York: Van Nostrand Heinhold Company, 1971. 288 p.

WILLIAMS, Laurie; KESSLER, Robert. **Pair programming illuminated**. 1. ed. Boston, MA: Addison-Wesley, 2003. 265 p.

WILLIAMS, Laurie; KESSLER, Robert; CUNNINGHAM, Ward, et al. **Strengthening the case for pair programming**. IEEE Software, v. 17, n. 4, 2000. p. 19-25.

YOURDON, Edward. **Death march**. 2. ed. Upper Saddle River, NJ: Prentice Hall PTR, 2004. 230 p.